

Tema 1. Introducción a la programación

1.1.-Un programa

1.1. Un programa informático

Un programa informático es un conjunto de instrucciones que permiten a un dispositivo, como un ordenador, realizar tareas específicas. Estas instrucciones están escritas en un lenguaje de programación y traducidas a lenguaje de máquina para que el hardware pueda ejecutarlas. Los programas se desarrollan para resolver problemas o automatizar procesos, siguiendo un ciclo que incluye la planificación, codificación, prueba y mantenimiento. A lo largo del proceso de programación, se utilizan herramientas y entornos que facilitan la creación y gestión del software

1. La programación

- **Definición:** Es el **proceso** por el cual se desarrolla un **programa**, haciendo uso de herramientas, entre otras: un **lenguaje de programación** mediante el que se indican las instrucciones al dispositivo y un **traductor** que sea capaz de *traducirlo* al lenguaje máquina. Este lenguaje máquina lo *entiende* el microprocesador del dispositivo (ordenador, móvil, tablet,...)
- El **Ciclo de vida** para el desarrollo de un programa es: Entender el problema, Recopilar requisitos, Planificar, Diseñar, Programar, Probar, Desplegar, Mantener.

2. Ordenador (o cualquier otro dispositivo)

- **Máquina** electrónica, analógica o digital, dotada de una memoria y de métodos de tratamiento de la información, capaz de resolver problemas matemáticos y lógicos mediante la utilización de programas informáticos.
- Un ordenador **ejecuta programas**, que son un conjunto de instrucciones representadas mediante un lenguaje de programación y datos que se ejecutan de forma secuencial y que a partir de unos datos de entrada producen una salida. Para ejecutar esos programas el ordenador sigue esta estructura básica:

3. ¿Qué es un programa o software?

El **software**, de acuerdo con el IEEE: “es el conjunto de los programas de cómputo, procedimientos, reglas, documentación y datos asociados, que forman parte de las operaciones de un sistema de computación”.

Dicho en otras palabras, son todos los programas o aplicaciones incluidas en un dispositivo y que le permiten realizar tareas específicas.

El software le da instrucciones al hardware de la forma como debe realizar una tarea, por esta razón, todos los programas que usamos en un dispositivo son software, por ejemplo:

- Navegador web como Google Chrome o Mozilla Firefox.
- Sistemas operativos como Windows, Mac OS, Linux, Android, entre otros.
- Antivirus.
- Aplicaciones de ofimática como Microsoft Word.
- Sistemas empresariales como un BPMS, ERP, CRM, entre otros.

3.1. Tipos de software

- **De sistema** (Sistema operativo, drivers -controladores-)
- **De aplicación** (Suite ofimática, Navegador, Edición de imagen, ...)
- **De desarrollo** (Editores, compiladores, interpretes, ...)

Los drivers son los controladores de dispositivos.

4. Relación Hardware-Software

La relación entre el **software** y el **hardware** se pueden describir de la siguiente forma:

- **Disco duro:** almacena de forma permanente los archivos ejecutables y los archivos de datos.
- **Memoria RAM:** almacena de forma temporal el código binario de los archivos ejecutables y los archivos de datos necesarios.
- **CPU:** lee y ejecuta instrucciones almacenadas en memoria RAM, así como los datos necesarios.
- **E/S:** estos dispositivos recogen nuevos datos desde la entrada, muestran los resultados, leen/guardan a disco, etc.

El disco duro se considera un periférico de E/S (Entrada/Salida).

La CPU se llama también UCP (en inglés), procesador o microprocesador.

5. Algoritmos

Como decíamos, la programación es el proceso que se utiliza para la creación de programas que se ejecutan en dispositivos con capacidad de cómputo. Estos programas son creados para satisfacer unas necesidades o resolver problemas.

Para que este proceso tenga éxito, se ha de analizar el problema que se quiere satisfacer y describir cada paso que se va a realizar, es decir, se ha de diseñar el algoritmo (secuencia de pasos) que se va a seguir para llegar a la solución.

Algoritmo: En términos de programación, un algoritmo es una secuencia de pasos lógicos que permiten solucionar un problema.

Una vez se tenga el algoritmo, se podrá pasar a su codificación, es decir, escribir ese algoritmo a código fuente mediante un lenguaje de programación y por último se generará el programa que se ejecutará en el ordenador para poder depurarse antes de darlo por finalizado.

5.1 Características de los algoritmos

Según *Joyanes* en su libro “Fundamentos de la programación”, las características que debe tener cualquier algoritmo son:

- **Preciso:** se debe indicar el orden de realización de cada paso.
- **Definido:** si se sigue un algoritmo dos veces con las mismas entradas, se debe obtener el mismo resultado.
- **Finito:** todo algoritmo debe terminar en algún momento.

Los algoritmos son **independientes del lenguaje en el que se implementan** y del dispositivo en el que se ejecutan. Aprender a programar, no es aprender un lenguaje de programación, si no realizar algoritmos correctos que resuelvan un problema.

5.2 Pseudocódigo y diagramas de flujo

El **pseudocódigo** se puede considerar como un lenguaje intermedio entre el lenguaje humano y el lenguaje de programación y las palabras reservadas de este. También permite la representación de las estructuras de control y la asignación de manera muy fácil.

Supongamos que queremos resolver un problema, sobre "*cómo realizar el mantenimiento de una lámpara*".

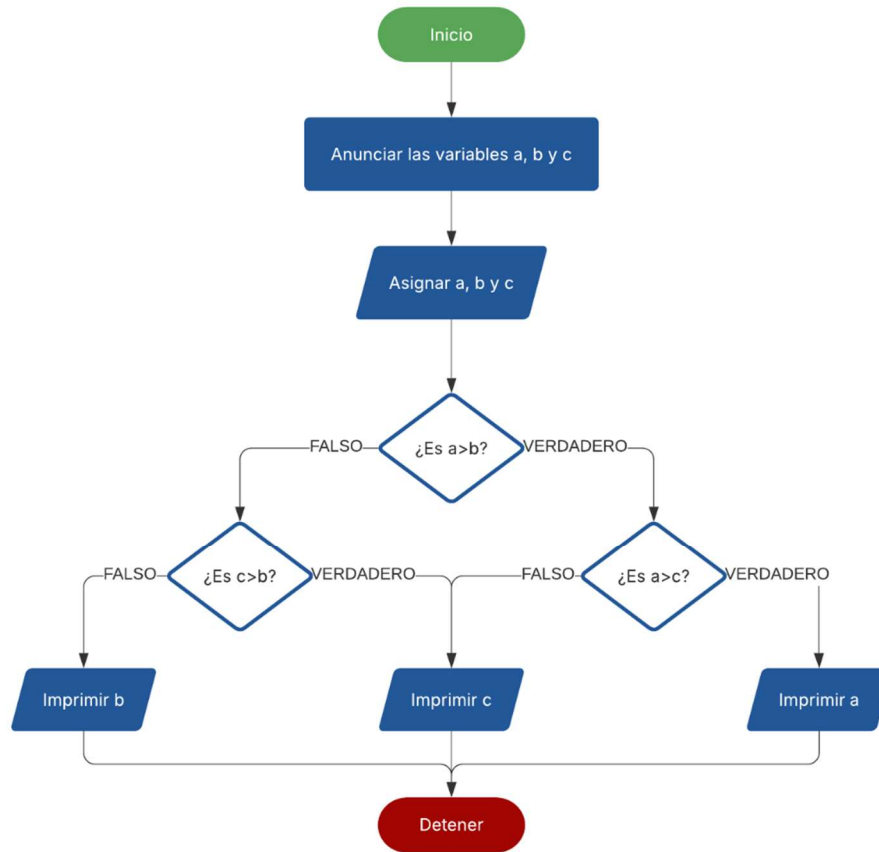
```
Si la lampara funciona entonces
    fin. # (1)
Si no
    Si la lampara NO está enchufada entonces
        Enchufarla.
    Si el foco está quemado entonces
        Reemplazar el foco.
    Si sigue sin funcionar entonces
        Comprar nueva lámpara.
fin. # (1)
```

1. Con la palabra *fin*, **finaliza** el programa.

Los **diagramas de flujo** son representaciones gráficas de la secuencia de operaciones que se realizan dentro de un algoritmo. Se representan mediante un conjunto de formas unidas por flechas. Para indicar el inicio del diagrama, se representa en un óvalo la palabra “inicio”. Una secuencia de operaciones se representa mediante una secuencia (lo más detallada posible) de rectángulos de arriba-abajo o derecha-izquierda. Un rombo representa una operación condicional con dos posibles caminos a seguir.

Ejemplo de diagrama de flujo de programación

System Templates | July 2, 2025



6. Lenguajes de programación

Un **lenguaje de programación** es un lenguaje formal que especifica una serie de instrucciones que pueden ser usadas para generar diversos tipos de datos, controlar el flujo de ejecución y representar datos. Los lenguajes de programación permiten a los programadores especificar de manera precisa las instrucciones que un ordenador debe seguir para llevar a cabo una tarea.

Este punto se estudiará de forma general en el módulo de entornos de desarrollo y particularmente (kotlin y python) en las siguientes unidades.

1.1.1.-Pseudocódigo

1.1.1. El pseudocódigo

El pseudocódigo es una forma de representar algoritmos de forma que sean fácilmente entendibles por cualquier persona, independientemente de su formación en programación. Se trata de un lenguaje de programación simplificado que utiliza expresiones y estructuras de control propias de la programación estructurada.

1. Pseudocódigo. Características

- Lenguaje cercano a un lenguaje de programación cuyo objetivo es el desarrollo de algoritmos fácilmente interpretables por un programador.
- Es independiente del lenguaje de programación en el que vayamos a realizar posteriormente nuestra aplicación.
- Debe utilizar un conjunto limitado de expresiones, pero no existe una sintaxis estandarizada.
- Pueden escribir algoritmos que tengan una solución finita y que comiencen desde un único punto de partida.

2. Elementos de un programa en pseudocódigo

A pesar de que no existe una norma rígida que establezca cómo realizar la escritura de programas en pseudocódigo, es recomendable seguir una serie de recomendaciones que permitan transcribir el programa al lenguaje de programación que va a usarse con la mayor facilidad.

A la hora de realizar programas en pseudocódigo, podemos utilizar los siguientes elementos:

- Inicio
- Fin
- Escribe "un texto a escribir"
- Lee X
- Si (condición) entonces
- Si (condición) entonces ... Sino
- Según (valor) entonces ... opcion1: ... opcion2: ...
- Mientras (condición) hacer
- Para X en (1...N) hacer
- Operadores matemáticos básicos: +, -, *, /, // y % (módulo).
- Operadores relacionales: == (igual), >, <, >=, <= y != (distinto).
- Operadores lógicos: and, or y not (negación).
- La asignación de valores a una variable la realizaremos con el símbolo =

2.1. Inicio y Fin

Todo algoritmo va a empezar por un paso o instrucción `Inicio` y va a terminar con la palabra reservada `Fin`.

Ejemplo:

```
Inicio
    Escribe "¿Cómo te llamas?"
    Lee nombre
    Escribe "Hola, " + nombre
Fin
```

2.2. Variables

- Una variable va a ser básicamente un contenedor de información al que le vamos a asignar un nombre en minúsculas.
- Podrá contener los siguientes tipos de datos: Cadena de caracteres (se representa con comillas dobles "Una cadena"), números (enteros y decimales) y valores lógicos (verdadero/falso).
- Su valor podrá ser modificado a lo largo del algoritmo.
- El tipo de datos que contiene no vamos a especificarlo explícitamente simplemente al asignar un valor, estaremos definiendo de forma implícita su tipo de datos.
- Podemos concatenar el valor de una variable a una cadena de caracteres con el símbolo +.
- No será necesario realizar conversiones de tipos de datos para trabajar (ya veréis posteriormente lo necesario que esto es en la programación). Se trata de simplificar al máximo la construcción y manejo de las variables en un algoritmo.

Ejemplo:

```
Inicio
    num1 = 10
    Escribe "Introduce un número: "
    Lee num2

    suma = num1 + num2
    Escribe "La suma de " + num1 + " + " + num2 + " es " + suma

    iva = 0.21
    Escribe "Introduce un precio: "
    Lee precio

    Escribe "El precio con IVA es " + (precio * iva)
Fin
```

2.3. Estructuras de control

El pseudocódigo utiliza las estructuras de control propias de la programación estructurada. * Estructuras de control secuencial. * Estructuras de control condicional. * Estructuras de control iterativa.

Para la construcción de un algoritmo vamos a simplificar estas estructuras lo máximo posible para su mejor entendimiento, ya posteriormente según el lenguaje de programación, veremos todas las opciones que nos proporciona.

2.3.1. Estructuras de control secuencial.

Describen bloques de instrucciones que son ejecutadas en orden de aparición (secuencialmente). Los bloques pueden estar delimitados por las expresiones Inicio-Fin o bien estar contenidos en otras estructuras.

```
Inicio
  Instrucción1
  ...
  InstrucciónN
Fin
```

2.3.2. Estructuras de control condicional.

La estructura de control condicional, nos permite ejecutar instrucciones de forma alternativa o selectiva, es decir encauza el flujo de ejecución hacia un bloque de instrucciones u otro en función de la evaluación que se realiza sobre una condición determinada.

El bloque o secuencia de instrucciones que ejecutará debe estar tabulado y acabará cuando esa indentación finalice, es decir, vuelva a encontrar una instrucción a la misma altura de la expresión Si (condición) entonces.

Vamos a utilizar: simple, doble, múltiple, anidados.

2.3.2.1 Condicional simple:

Establece un conjunto de instrucciones que se ejecutarán si se cumple una condición que retornará un valor lógico.

Ejemplo:

```
Si (condición) entonces
  Instrucción1
  ...
  InstrucciónN
```

2.3.2.2 Condicional doble:

Añade otro bloque de instrucciones que se ejecuta en caso de que no se cumpla la condición.

Ejemplo:

```
Si (condición) entonces
  Instrucción1
  ...
  InstrucciónN
Sino
  Instrucción1
  ...
  InstrucciónN
```


2.3.2.3 Condicional múltiple:

Permite definir multiple bloques de instrucciones que se ejecutarán en función de la opción que sea verdadera:

```
Según valor_selector entonces
  opcion1:
    Instrucción1
  ...
  InstrucciónI
  opcion2:
    InstrucciónJ
  ...
  InstrucciónN
```

2.3.2.3 Estructuras anidados:

Permite ejecutar diferentes bloques de instrucciones mediante el anidamiento de diferentes estructuras de control, cuyas condiciones son excluyentes.

Ejemplo:

```
Si (condición) entonces
  Instrucciones1
Sino
  Si (condición) entonces
    InstruccionesI
  Sino
    Si (condición) entonces
      InstruccionesJ
    Sino
      InstruccionesN
```

2.3.3. Estructuras de control iterativa.

La estructura de control iterativa permite que un bloque de instrucciones sea ejecutado mientras se cumpla una condición.

Solo vamos a contemplar dos tipos de bucles: Mientras y Para.

2.3.3.1. Estructura iterativa Mientras.

Iteración con salida al principio (Mientras): primero se evalúa la condición y en caso de cumplirse ejecuta el bloque de instrucciones. Las instrucciones contenidas deben actuar sobre los valores usados en la condición para evitar bucles infinitos.



En algunos lenguajes suele existir otra estructura similar que podrías asemejarse a Hacer ... Hasta (condición) en donde primero se ejecutan las instrucciones y antes de proseguir se evalúa la condición, por tanto, siempre se ejecutará el bloque de instrucciones una vez.

Ejemplos:

```
Mientras (condición) hacer
    Instrucción1
    ...
    InstrucciónN
```

```
Mientras (cont > 0) hacer
    Escribe cont
    cont = cont - 1
```



Actividad: ¿Cuál es el resultado del algoritmo anterior?

2.3.3.2. Estructura iterativa Para.

Ejecutará el bloque de instrucciones un número determinado de veces. Hace uso de una variable que irá incrementando o decrementando su valor de uno en uno en función de un rango de valores.

Ejemplos:

```
Para i en (1...N) hacer
    Instrucción1
    ...
    InstrucciónN
Para i en (N...0) hacer
    Instrucción1
    ...
    InstrucciónN
Inicio
    suma = 0
    Para i en (1...10) hacer
        suma = suma + 1
    Escribe "La suma de los primeros 10 números enteros es " + suma
Fin
```



Actividad: ¿Cuál es el resultado del algoritmo anterior?



Actividad: Realiza un algoritmo que lea dos números y muestre cuál es el mayor.

```
Inicio
    Lee num1
    Lee num2

    Si (num1 > num2) Entonces
        Escribe num1 + " es mayor que " + num2
    Sino
        Escribe num2 + " es mayor que " + num1

Fin
```



Actividad: Muestra la relación entre dos números que introduce el usuario.

```

Inicio
  Lee num1
  Lee num2

  Si (num1 == num2) entonces
    Escribe num1 + " es igual que " + num2
  Sino
    Si (num1 > num2) entonces
      Escribe num1 + " es mayor que " + num2
    Sino
      Escribe num2 + " es mayor que " + num1

Fin

```



Actividad: Lee un número, si es mayor que 0 muestra la serie decrementando su valor hasta 0. Por ej: 7 => 7 6 5 4 3 2 1 0

```

Inicio
  Lee num

  Si (num > 0) entonces
    Escribe num + " => "
    Mientras (num >= 0) hacer
      Escribe num + " "
      num = num - 1

Fin

```

```

Inicio
  Lee num

  Si (num > 0) entonces
    Escribe num + " => "
    Para i en (num...0) hacer
      Escribe i + " "

Fin

```



Actividad: Lee un número, si es mayor que 0 muestra la serie decrementando su valor hasta 0. Usa el separador coma. Por ej: 7 => 7, 6, 5, 4, 3, 2, 1, 0

```

Inicio
  Lee num

  Si (num > 0) entonces
    Escribe num + " => "

    Mientras (num >= 0) hacer
      Escribe num
      Si (num != 0) entonces
        Escribe ", "
      num = num - 1

Fin

```



Actividad: Lee un número, si es mayor que 0 muestra la serie decrementando su valor hasta 0 (*usa el bucle Para y separador coma*).

Por ej: 7 => 7, 6, 5, 4, 3, 2, 1, 0

```
Inicio
  Lee num

  Si (num > 0) entonces
    Escribe num + " => "

    Para i en (num...0) hacer
      Escribe i
      Si (i != 0) entonces
        Escribe ", "

Fin
```



Ejercicio 1: Lee un número hasta que el número esté en el rango 1-10

```
Inicio
  Escribir "Introduce un número entre 1 y 10:"
  Leer numero

  Mientras numero < 1 O numero > 10 Hacer
    Escribir "Número fuera de rango. Intenta nuevamente:"
    Leer numero
  FinMientras

  Escribir "Número válido: ", numero
  Escribir "Correcto!"

Fin
```

... otra forma (bucle que dijimos que no íbamos a ver repite-hasta)

```
Inicio
  Repetir
    Escribir "Introduce un número entre 1 y 10:"
    Leer numero
  Hasta Que numero >= 1 Y numero <= 10
  Escribir "Número válido: ", numero
  Escribir "Correcto!"

Fin
```

```
Introduce un número: 15
Inténtalo otra vez! (1-10): 0
Inténtalo otra vez! (1-10): 5
Correcto!
```



Ejercicio 2: Lee dos números y crea la serie que los une de 1 en 1...

Inicio

Escribir "Introduce un número:"

Leer num1

Escribir "Introduce otro:"

Leer num2

Si num1 <= num2 Entonces

Para i desde num1 hasta num2 Hacer

Escribir i

FinPara

Sino

Para i desde num2 hasta num1 Hacer

Escribir i

FinPara

FinSi

Fin

Introduce un número: 4

Introduce otro: 8

4-5-6-7-8

Introduce un número: 12

Introduce otro: 3

3-4-5-6-7-8-9-10-11-12



Ejercicio 3: Lee 3 números y dame los números ordenados de menor a mayor.

Inicio

Escribir "Introduce el primer número:"

Leer A

Escribir "Introduce el segundo número:"

Leer B

Escribir "Introduce el tercer número:"

Leer C

Si A > B Entonces

Intercambiar A y B

FinSi

Si A > C Entonces

Intercambiar A y C

FinSi

Si B > C Entonces

Intercambiar B y C

FinSi

Escribir "Tus números son: ", A, ", ", B, ", ", C

Fin

... sin técnica de intercambio se complicaría un poco

```

Inicio
  Escribir "Introduce el primer número:"
  Leer A
  Escribir "Introduce el segundo número:"
  Leer B
  Escribir "Introduce el tercer número:"
  Leer C

  Si A <= B Y B <= C Entonces
    Escribir Tus números son A, B, C
  Sino Si A <= C Y C <= B Entonces
    Escribir Tus números son A, C, B
  Sino Si B <= A Y A <= C Entonces
    Escribir Tus números son B, A, C
  Sino Si B <= C Y C <= A Entonces
    Escribir Tus números son B, C, A
  Sino Si C <= A Y A <= B Entonces
    Escribir Tus números son C, A, B
  Sino
    Escribir Tus números son C, B, A
  FinSi
Fin

```

```

Dame 3 números:
14
7
10
Tus números son 7 10 14

```

* Práctica 1.1: Instala python

P1.1 - Mi primer programa - Windows

A continuación, ofrecemos una guía paso a paso para aquellos usuarios principiantes interesados en aprender Python con Windows.

1. Configurar el entorno de desarrollo

Si eres un usuario principiante y no estás familiarizado con Python, te recomendamos [instalar Python desde Microsoft Store](#). La instalación a través de Microsoft Store utiliza el intérprete de Python3 básico, pero controla el establecimiento de la configuración del valor PATH para el usuario actual (lo que evita la necesidad de contar con acceso de administrador) y, además, proporciona actualizaciones automáticas. Resulta especialmente útil si te encuentras en un entorno educativo o en un departamento de una organización que restringe los permisos o el acceso administrativo en la máquina.

2. Instalar Python

Para instalar Python con Microsoft Store:

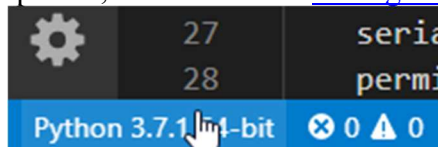
1. Ve al menú **Inicio** (icono de Windows de la esquina inferior izquierda), escribe "Microsoft Store" y selecciona el vínculo para abrir Store.
2. Una vez que lo hayas abierto, selecciona **Buscar** en el menú superior derecho y escribe "Python". Selecciona la versión de Python que quiera usar en los resultados de la opción Aplicaciones. Se recomienda usar la más reciente, a menos que tenga una razón para no hacerlo (por ejemplo, alinearse con la versión que se usó en un proyecto existente en el que planea trabajar). Una vez que haya determinado qué versión quiere instalar, seleccione **Obtener**.
3. Una vez que Python haya completado el proceso de descarga e instalación, abre Windows PowerShell mediante el menú **Inicio** (icono de Windows de la esquina inferior izquierda). Cuando PowerShell esté abierto, escribe `python --version` para confirmar que Python3 está instalado en la máquina.
4. La instalación de Microsoft Store de Python incluye **PIP**, el administrador de paquetes estándar. PIP te permite instalar y administrar paquetes adicionales que no forman parte de la biblioteca estándar de Python. Para confirmar que también dispones de PIP para instalar y administrar paquetes, escribe `pip --version`.

3. Instalar Visual Studio Code

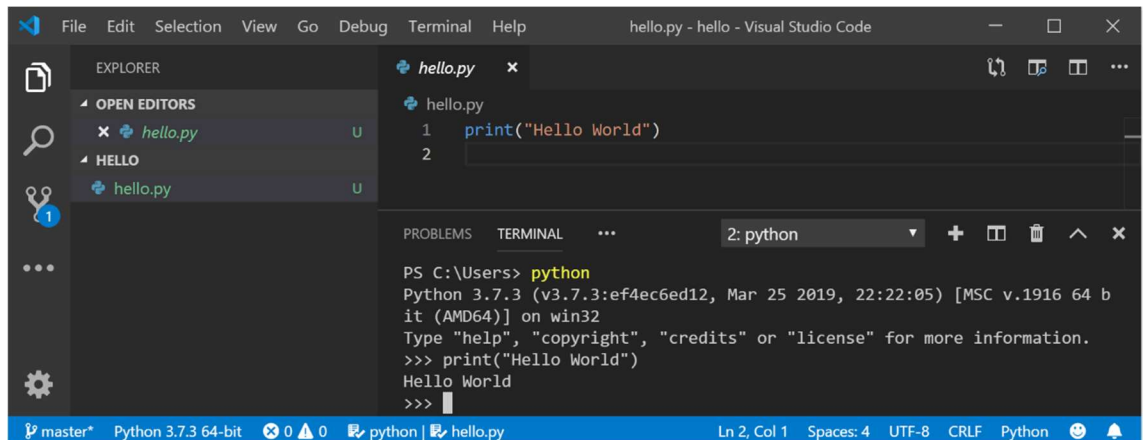
Al usar VS Code como editor de texto/entorno de desarrollo integrado (IDE), puedes aprovechar [IntelliSense](#) (una ayuda de finalización de código), el [detector de errores](#) (permite evitar que se produzcan errores en el código), el [soporte técnico de depuración](#) (ayuda a buscar errores en el código después de ejecutarlo), los [fragmentos de código](#) (plantillas para pequeños bloques de código reutilizables) y las [pruebas unitarias](#) (para probar la interfaz del código con distintos tipos de entrada).

VS Code también contiene un [terminal integrado](#) que te permite abrir una línea de comandos de Python con el símbolo del sistema de Windows, PowerShell o cualquier otra herramienta que prefieras, y establece un flujo de trabajo sin interrupciones entre el editor de código y la línea de comandos.

1. Para instalar VS Code, descarga VS Code para Windows: <https://code.visualstudio.com>.
2. Una vez instalado VS Code, también debes instalar la extensión de Python. Para instalar la extensión de Python, puedes seleccionar el [vínculo para VS Code de Marketplace](#) o abrir VS Code y buscar **Python** en el menú de extensiones (Control + Mayús + X).
3. Python es un lenguaje interpretado y, para ejecutar el código de Python, debes indicar a VS Code el intérprete que debe usar. Se recomienda usar la versión más reciente de Python, a menos que tenga una razón específica para elegir alguna diferente. Después de instalar la extensión de Python, selecciona un intérprete de Python 3. Para ello, abre la **paleta de comandos** (Control + Mayús + P) y empieza a escribir el comando **Python: Select Interpreter** para buscarlo y, luego, selecciónalo. También puedes usar la opción **Select Python Environment** (Seleccionar entorno de Python) en la barra de estado inferior si está disponible (es posible que ya se muestre un intérprete seleccionado). El comando presenta una lista de los intérpretes disponibles que VS Code puede buscar automáticamente, incluidos los entornos virtuales. Si no ves el intérprete que quieres, consulta [Configuración de los entornos de Python](#).



4. Para abrir el terminal en VS Code, selecciona **Ver > Terminal**, o bien usa el acceso directo **Control + `** (mediante el carácter de tilde aguda). El terminal predeterminado es PowerShell.
5. En el terminal de VS Code, simplemente escribe el comando `python` para abrir Python.
6. Para probar el intérprete de Python, escribe `print("Hello World")`. Python devolverá la instrucción "Hola mundo".



4. Instalar GIT (opcional)

Si planeas colaborar con otras personas en el código de Python u hospedar el proyecto en un sitio de código abierto (como GitHub), VS Code admite el [control de versiones con GIT](#). La pestaña Control de código fuente de VS Code realiza un seguimiento de todos los cambios y tiene comandos GIT comunes (agregar, confirmar, enviar cambios e incorporar cambios) integrados directamente en la interfaz de usuario. Primero, debes instalar GIT para alimentar el panel de control de código fuente.

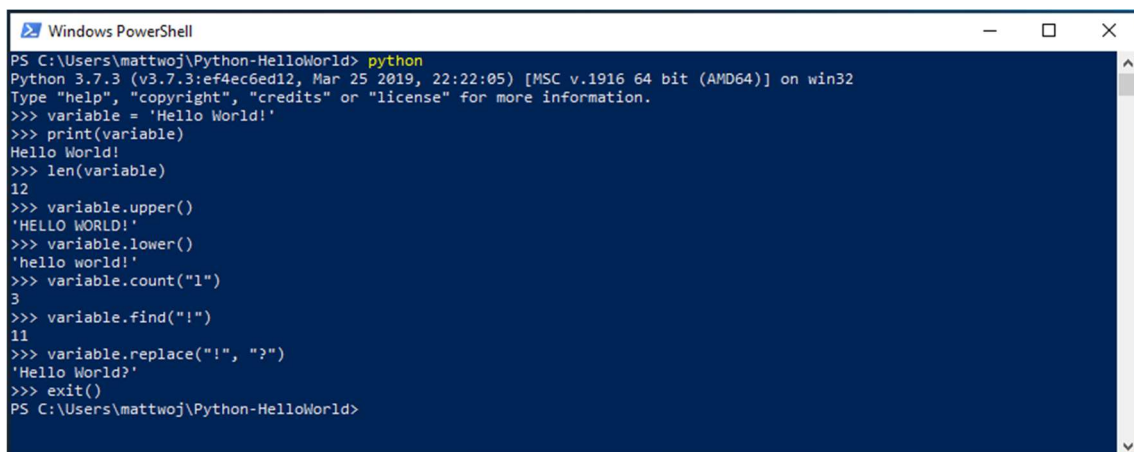
1. Descarga e instala GIT para Windows desde el [sitio web git-scm](#).
2. Se incluye un asistente para instalación que te formulará una serie de preguntas sobre la configuración de la instalación de GIT. Te recomendamos que uses todas las opciones de configuración predeterminadas, a menos que tengas un motivo concreto para cambiar algo.
3. Si nunca has trabajado con GIT, las [guías de GitHub](#) pueden resultarte de ayuda para empezar.

5. Tutorial de Hola mundo para algunos aspectos básicos de Python

Python, según su creador Guido van Rossum, es un "lenguaje de programación de alto nivel y su filosofía de diseño básico trata sobre la legibilidad del código y una sintaxis que permite a los programadores expresar conceptos en unas pocas líneas de código".

Python es un lenguaje interpretado. A diferencia de los lenguajes compilados, en los que el código que escribes debe traducirse en código máquina para que lo ejecute el procesador del equipo, el código de Python se pasa a un intérprete y se ejecuta directamente. Solo tienes que escribir el código y ejecutarlo. Probémoslo.

1. Con la línea de comandos de PowerShell abierta, escribe `python` para ejecutar el intérprete de Python 3. (Algunas instrucciones prefieren usar el comando `py` o `python3` y también deberían funcionar). Sabrá que se ha ejecutado correctamente porque se mostrará un aviso `>>>` con tres símbolos de "mayor que" .
2. Hay varios métodos integrados que permiten realizar modificaciones en las cadenas de Python. Crea una variable con `variable = 'Hello World!'`. Presiona Entrar para que se muestre una nueva línea.
3. Imprime la variable con `print(variable)`. Se mostrará el texto "Hello World!".
4. Averigua la longitud (el número de caracteres que se usan) de la variable de cadena con `len(variable)`. Se mostrará que se usan 12 caracteres. (Ten en cuenta que el espacio en blanco se cuenta como un carácter en la longitud total).
5. Convierte la variable de cadena en letras mayúsculas: `variable.upper()`. Convierte la variable de cadena en letras minúsculas: `variable.lower()`.
6. Cuenta el número de veces que se usa la letra "l" en la variable de cadena: `variable.count("l")`.
7. Busca un carácter específico en la variable de cadena. En este caso, buscaremos el signo de exclamación con `variable.find("!")`. Se mostrará que el signo de exclamación se encuentra en el carácter undécimo de la cadena.
8. Reemplaza el signo de exclamación por un signo de interrogación: `variable.replace("!", "?")`.
9. Para salir de Python, puedes escribir `exit()` o `quit()`, o seleccionar Control-Z.



```
Windows PowerShell
PS C:\Users\mattwoj\Python-HelloWorld> python
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 22:22:05) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> variable = 'Hello World!'
>>> print(variable)
Hello World!
>>> len(variable)
12
>>> variable.upper()
'HELLO WORLD!'
>>> variable.lower()
'hello world!'
>>> variable.count("l")
3
>>> variable.find("!")
11
>>> variable.replace("!", "?")
'Hello World?'
>>> exit()
PS C:\Users\mattwoj\Python-HelloWorld>
```

Lo que acabas de ver, son algunos de los métodos de modificación de cadenas integrados de Python. Ahora intenta crear un archivo de programa de Python y ejecutarlo con VS Code.

6. Tutorial Hola mundo para usar Python con VS Code

El equipo de VS Code ha elaborado el excelente tutorial [Introducción a Python](#) en el que se explica cómo crear un programa Hola mundo con Python, ejecutar el archivo de programa, configurar y ejecutar el depurador e instalar paquetes como *matplotlib* y *NumPy* para crear un trazado gráfico dentro de un entorno virtual.

1. Abre PowerShell y crea una carpeta vacía denominada "hello", navega a esta carpeta y ábrela en VS Code: ****Consola**Copiar**

```
mkdir hello
cd hello
code .
```

2. Una vez que se abra VS Code y se muestre la nueva carpeta *Hello* en la ventana **Explorador** del lado izquierdo, abra una ventana de línea de comandos en el panel inferior de VS Code. Para ello, presione **Control + `** (mediante el carácter de tilde aguda) o seleccione **Ver > Terminal**. Al iniciar VS Code en una carpeta, esa carpeta se convierte en tu "área de trabajo". VS Code almacena la configuración específica de esa área de trabajo en. `vscode/settings.json`, que es independiente de la configuración de usuario que se almacena globalmente. 3. Continúa con el tutorial en la documentación de VS Code: [Creación de un archivo de código fuente de Hola mundo de Python](#).

Fuente

- [Introducción a Python](#)
- [Introducción para principiantes](#)

1.2.-Practica con un lenguaje

1.2. Practica con un lenguaje

Vamos a iniciar con la práctica de un lenguaje de programación, en este caso, Python. Vemos los bloques de un programa y las estructuras básicas de un programa en Python.

1. Bloques de un programa

1.1. En Kotlin

Los programas están compuestos por un conjunto de bloques

- [Main, punto de entrada](#): Inicio del flujo de ejecución de un programa
- [Paquetes e import](#): Agrupa clases, e incorpora clases para su uso, respectivamente
- [Funciones](#): Bloques de código con nombre que pueden ser reutilizados

1.2. En python

En Python, un programa es un conjunto de módulos. Un módulo es un archivo que contiene código Python. Puede contener definiciones de funciones, clases y variables, así como ejecutar código. A continuación, se muestra un ejemplo de un módulo simple y los bloques que lo componen.

```
# Importación de módulos

from file_two import function_three

print("File one __name__ is set to: {}".format(__name__))


# Definición de funciones

def main():
    print("Function main is executed")
    print("Cuerpo principal")


if __name__ == "__main__":
    # Por aquí pasa cuando es ejecutado, python file_one.py
    main()
else:
    # Por aquí pasa cuando es importado, import file_one
    print("File one executed when imported")
```

2. Comenzando con Python

A continuación, se presenta una introducción a Python, un lenguaje de programación de alto nivel. Python es un lenguaje de programación interpretado, lo que significa que el código se ejecuta línea por línea. Python es un lenguaje de programación versátil y fácil de aprender, que se utiliza en una amplia variedad de aplicaciones, desde el desarrollo web hasta la ciencia de datos.

2.1. Introducción a Python: Características del lenguaje

Python es un lenguaje de programación de alto nivel. Las principales características de Python son las siguientes:

- **Es multiparadigma**, ya que soporta la programación imperativa, programación orientada a objetos y funcional.
- **Es multiplataforma**: Se puede encontrar un intérprete de Python para los principales sistemas operativos como *Windows*, *Linux* y *Mac OS*. Además, se puede reutilizar el mismo código en cada una de las plataformas.
- **Es dinámicamente tipado**: Es decir, el tipo de las variables se decide en tiempo de ejecución.
- **Es fuertemente tipado**: No se puede usar una variable en un contexto fuera de su tipo. Si se quisiera, habría que hacer una conversión de tipos.
- **Es interpretado**: El código no se compila a lenguaje máquina.

El hecho de que Python sea **interpretado** quiere decir que **hace falta un intérprete** que permita ejecutar un programa o script escrito en Python sin necesidad de compilarlo.

2.2. El intérprete de Python

Cuando instalas Python correctamente (en cualquier sistema operativo) ocurren, entre otras, dos cosas: se añade el comando `python` (o `python3`, en caso de que instales la versión 3.x de Python) al path y se instala el intérprete de Python correspondiente.

En el intérprete de Python podemos escribir expresiones e instrucciones que este interpretará y ejecutará.

Puedes probar, por ejemplo, a escribir `2 + 3`. El resultado debe ser el siguiente:

```
>>>2 + 3
5
```

O ejecutar la instrucción `print(';Hola mundo!')`:

```
>>>print(';Hola mundo!')
;Hola mundo!
```

Para salir del intérprete basta con ejecutar la instrucción `quit()`.

No obstante, aunque esta forma de escribir código puede ser útil para aprender y en casos muy puntuales, no es la habitual a la hora de escribir un programa o script en Python.

2.3. Primer programa en Python

Normalmente, los programas en Python se escriben en archivos con la extensión `.py`. Estos archivos se pasan al intérprete de Python para que los interprete y ejecute.

Vamos a verlo con un ejemplo. Crea con un editor de texto un fichero llamado `suma.py` con el siguiente contenido:

```
suma = 2 + 3
print(suma)
```

A continuación, abre un terminal, sitúate en el directorio en el que creaste el archivo `suma.py` y ejecuta lo siguiente:

```
$ python3 suma.py
```

En el terminal verás que aparece el número 5 como resultado de ejecutar el programa anterior. ¿Qué ha ocurrido aquí? Básicamente que el intérprete de Python ha leído y ejecutado las líneas de código que hemos escrito en el fichero `suma.py`.

Esta es la manera más común de crear y ejecutar programas en Python.

3. Variables, literales y constantes

En esta sección vamos a ver cómo se definen las variables en Python, qué son los literales y cómo se definen las constantes.

3.1. Valores (literales) y tipos

Un *valor* o *literal* es una de las cosas básicas que utiliza un programa, como una letra o un número. Hasta ahora hemos visto valores o literales como 1, 2, y `¡Hola, mundo!`

Esos valores pertenecen a *tipos* diferentes: 2 es un entero (`int`), y `¡Hola, mundo!` es una *cadena* (string), que recibe ese nombre porque contiene una “cadena” de letras. Tú (y el intérprete) podéis identificar las cadenas porque van encerradas entre comillas.

En python existen los siguientes tipos básicos de datos:

- * **int**: Números enteros. Por ejemplo, 2, 4, 20. Los números enteros son aquellos que no tienen parte decimal. Por ejemplo, 2, 4, 20. Aunque en Python no hay un límite en el tamaño de los enteros, en la práctica, el tamaño de un entero está limitado por la memoria de la computadora.
- * **float**: Números decimales. Por ejemplo, 5.0, 1.6, 3.14159. Los números decimales son aquellos que tienen parte decimal. Por ejemplo, 5.0, 1.6, 3.14159. Los números decimales en Python se representan con un punto decimal.
- * **str**: Cadenas de texto. Por ejemplo, 'Hola', 'Python', '3.14159'. Las cadenas de texto son secuencias de caracteres. Por ejemplo, 'Hola', 'Python', '3.14159'. Las cadenas de texto en Python se pueden definir con comillas simples ('...') o dobles ("...").
- * **bool**: Valores booleanos. Por ejemplo, True, False. Los valores booleanos son aquellos que representan la verdad o la falsedad. En Python, los valores booleanos son True y False. Los valores booleanos se utilizan en expresiones lógicas y de comparación.
- * **None**: Representa la ausencia de valor. None es un valor especial en Python que se utiliza para representar la ausencia de valor.

Vamos a usar el comando `python` para iniciar el intérprete.

```
$python
>>>print(4)
4
```

La sentencia `print` además de funcionar con cadenas, también funciona con enteros.

Si no estás seguro de qué tipo de valor estás manejando, el intérprete te lo puede decir.

```
>>>type('¡Hola, mundo!')
<class 'str'>
>>>type(17)
<class 'int'>
```

No es sorprendente que las cadenas pertenezcan al tipo `str` y los enteros pertenezcan al tipo `int`. De manera menos obvia, los números con un punto decimal pertenecen a un tipo llamado `float`, porque estos números se representan en un formato llamado *punto flotante*.

```
>>>type(3.2)
>>><class 'float'>
```

¿Qué ocurre con valores como 17 y 3.2? Parecen números, pero van entre comillas como las cadenas.

```
>>>type('17')
<class 'str'>
>>>type('3.2')
<class 'str'>
```

Pues son cadenas.

Cuando escribes un entero grande, puede que te sientas tentado a usar comas o puntos para separarlo en grupos de tres dígitos, como en 1,000,000. Eso no es un entero válido en Python, pero en cambio sí que resulta válido algo como:

```
>>> print(1,000,000)
1 0 0 # Imprime 3 numeros
```

Bien, ha funcionado. ¡Pero eso no era lo que esperábamos!. Python interpreta 1,000,000 como una secuencia de enteros separados por comas, así que lo imprime con espacios en medio.

Éste es el primer ejemplo que hemos visto de un error semántico: el código funciona sin producir ningún mensaje de error, pero no hace su trabajo “correctamente”.

En este caso deberíamos usar salida formateada, como se verá más adelante:

```
print("{:,}".format(1000000))
```

3.2. Variables

Una de las características más potentes de un lenguaje de programación es la capacidad de manipular *variables*. Una variable es un nombre que se refiere a un valor, un literal.

Una *sentencia de asignación* crea variables nuevas y les da valores:

```
>>> mensaje = 'Y ahora algo completamente diferente'
>>> n = 17
>>> pi = 3.1415926535897931
```

Este ejemplo hace tres asignaciones. La primera asigna una cadena a una variable nueva llamada `mensaje`; la segunda asigna el entero `17` a `n`; la tercera asigna el valor (aproximado) de π a `pi`.

Para mostrar el valor de una variable, se puede usar la sentencia `print`:

```
>>> print(n)
17
>>> print(pi)
3.141592653589793
```

El tipo de una variable es el tipo del valor al que se refiere.

```
>>> type(mensaje)
<class 'str'>
>>> type(n)
<class 'int'>
>>> type(pi)
<class 'float'>
```

3.3. Constantes en Python

A diferencia de otros lenguajes, **en Python no existen las constantes**.

Entendemos como *constante* una variable que, una vez asignado un valor, este no se puede modificar. Es decir, que a la variable no se le puede asignar ningún otro valor una vez asignado el primero.

Se puede simular este comportamiento, siempre desde el punto de vista del programador y atendiendo a convenciones propias, pero no podemos cambiar la naturaleza mutable de las variables.

No obstante, sí que es cierto que el propio Python define una serie de valores constantes en su propio namespace. Los más importantes son:

- **False:** El valor *false* del tipo `bool`.
- **True:** El valor *true* del tipo `bool`.
- **None:** El valor del tipo `NoneType`. Generalmente `None` se utiliza para representar la ausencia de valor de una variable.

4. Operadores, expresiones y sentencias en Python

Para introducirse en cualquier lenguaje es importante saber la diferencia entre operador, expresión y sentencia, ya que son las formas básicas que componen la estructura de cualquier programa.

4.1. Operador

Un **operador** es un carácter o conjunto de caracteres que actúa sobre una, dos o más **variables** y/o **literales** para llevar a cabo una **operación** con un **resultado** determinado. Veremos la definición de variable y literales en los siguientes puntos.

Ejemplos de operadores comunes son los operadores aritméticos + (suma), - (resta) o * (producto), aunque en Python existen otros operadores.

4.1.1 Orden de las operaciones

Cuando en una expresión aparece más de un operador, el orden de evaluación depende de las *reglas de precedencia*. Para los operadores matemáticos, Python sigue las convenciones matemáticas. El acrónimo *PEMDSR* resulta útil para recordar esas reglas:

- Los Paréntesis tienen el nivel superior de precedencia, y pueden usarse para forzar a que una expresión sea evaluada en el orden que se quiera. Dado que las expresiones entre paréntesis son evaluadas primero, $2 * (3-1)$ es 4, y $(1+1) ** (5-2)$ es 8. Se pueden usar también paréntesis para hacer una expresión más sencilla de leer, incluso si el resultado de la misma no varía por ello, como en $(minuto * 100) / 60$.
- La Exponenciación (elevar un número a una potencia) tiene el siguiente nivel más alto de precedencia, de modo que $2**1+1$ es 3, no 4, y $3*1**3$ es 3, no 27.
- La Multiplicación y la División tienen la misma precedencia, que es superior a la de la *S*uma y la *R*esta, que también tienen entre sí el mismo nivel de precedencia. Así que $2*3-1$ es 5, no 4, y $6+4/2$ es 8, no 5.
- Los operadores con igual precedencia son evaluados de izquierda a derecha. Así que la expresión $5-3-1$ es 1 y no 3, ya que $5-3$ se evalúa antes, y después se resta 1 de 2.

En caso de duda, añade siempre paréntesis a tus expresiones para asegurarte de que las operaciones se realizan en el orden que tú quieres.

4.1.2. Operaciones con cadenas

El operador + funciona con las cadenas, pero no realiza una suma en el sentido matemático. En vez de eso, realiza una *concatenación*, que quiere decir que une ambas cadenas, enlazando el final de la primera con el principio de la segunda. Por ejemplo:

```
>>> primero = 10
>>> segundo = 15
>>> print(primeros+segundo)
25
>>> primero = '100'
>>> segundo = '150'
>>> print(primeros + segundo)
100150
```

La salida de este programa es 100150.

El operador * también trabaja con cadenas multiplicando el contenido de una cadena por un entero. Por ejemplo:

```
>>> primero = 'Test '
>>> second = 3
>>> print(primeros * second)
Test Test Test
```

Veremos más sobre los operadores en los siguientes puntos.

4.2. Expresión

Una expresión es una unidad de código que **devuelve un valor** y está formada por una combinación de operandos (variables y literales) y operadores. Los siguientes son ejemplos de expresiones (cada línea es una expresión diferente):

```
5 + 2          # Suma del número 5 y el número 2
a < 10         # Compara si el valor de la variable a es menor que
10
b is None      # Compara si la identidad de la variable b es None
3 * (200 - c)  # Resta a 200 el valor de c y lo multiplica por 3
```

4.3. Sentencia

Por su parte, una sentencia o declaración es una instrucción que define una acción. Una sentencia puede estar formada por una o varias expresiones, aunque no siempre es así.

En definitiva, las sentencias son las instrucciones que componen nuestro programa y determinan su comportamiento.

Ejemplos de sentencias son la asignación `=` o las instrucciones `if`, `if ... else ...`, `for` o `while` entre otras.

Una sentencia está delimitada por el carácter `Enter` (`\n`).

4.3.1. Sentencias de más de una línea

Normalmente, las sentencias ocupan una sola línea. Por ejemplo:

```
a = 2 + 3 #Asigna a la variable <a> el resultado de 2 + 3
```

Sin embargo, aquellas sentencias que son muy largas pueden ocupar más de una línea ([la guía de estilo PEP 8](#), recomienda una longitud de línea máxima de 72 caracteres).

Para dividir una sentencia en varias líneas se utiliza el carácter `\`. Por ejemplo:

```
a = 2 + 3 + 5 + \
7 + 9 + 4 + \
6
```

Además de la separación explícita (la que se realiza con el carácter `\`), en Python la continuación de línea es implícita siempre y cuando la expresión vaya dentro de los caracteres `()`, `[]` y `{}`.

Por ejemplo, podemos inicializar una lista del siguiente modo:

```
a = [1, 2, 7,  
3, 8, 4,  
9]
```

No te preocupes si no sabes lo que es una lista o no entiendes lo que hace el ejemplo anterior. Lo importante es que comprendas que lo anterior es una sentencia multi-línea ya que está comprendida entre los caracteres `[]`.

4.3.2. Bloques de código (Indentación)

Lo último que veremos sobre sentencias en esta introducción a Python es cómo se pueden agrupar en bloques de código.

Un bloque de código es un grupo de sentencias relacionadas bien delimitadas. A diferencia de otros lenguajes como JAVA o C, en los que se usan los caracteres `{ }` para definir un bloque de código, en Python se usa la indentación o sangrado.

El sangrado o indentación consiste en mover un bloque de texto hacia la derecha insertando espacios o tabuladores al principio de la línea, dejando un margen a la izquierda.

Esta es una de las principales características de Python.

Un bloque comienza con un nuevo sangrado y acaba con la primera línea cuyo sangrado sea menor. De nuevo, la guía de estilo de Python recomienda usar los espacios en lugar de las tabulaciones para realizar el sangrado. Yo suelo utilizar 4 espacios.

Configura tu IDE de desarrollo para que use los espacios en lugar de los tabuladores para el sangrado. Establece el número de espacios a 4 ó 2.

Veamos todo esto con un ejemplo:

```
def suma_numeros(numeros):      # Bloque 1  
    suma = 0                    # Bloque 2  
    for n in numeros:           # Bloque 2  
        suma += n               # Bloque 3  
        print(suma)             # Bloque 3  
    return suma                 # Bloque 2
```

Como te decía en la sección anterior, no hace falta todavía que entiendas lo que hace el ejemplo. Simplemente debes comprender que en la línea 1 se define la función `suma_numeros`. El cuerpo de esta función está definido por el grupo de sentencias que pertenecen al bloque 2 y 3. A su vez, la sentencia `for` define las acciones a realizar dentro de la misma en el conjunto de sentencias que pertenecen al bloque 3.

5. Petición de información al usuario

A veces necesitaremos que sea el usuario quien nos proporcione el valor para una variable, a través del teclado. Python proporciona una función interna llamada `input` que recibe la entrada desde el teclado. Cuando se llama a esa función, el programa se detiene

y espera a que el usuario escriba algo. Cuando el usuario pulsa `Retorno` o `Intro`, el programa continúa y `input` devuelve como **una cadena** aquello que el usuario escribió.

```
>>>entrada = input()
Cualquier cosa ridícula
>>>print(entrada)
Cualquier cosa ridícula
```

Antes de recibir cualquier dato desde el usuario, es buena idea escribir un mensaje explicándole qué debe introducir. Se puede pasar una cadena a `input`, que será mostrada al usuario antes de que el programa se detenga para recibir su entrada:

```
>>>nombre = input('¿Cómo te llamas?\n')
¿Cómo te llamas?
Chuck
>>>print(nombre)
Chuck
```

La secuencia `\n` al final del mensaje representa un *newline*, que es un carácter especial que provoca un salto de línea. Por eso la entrada del usuario aparece debajo de nuestro mensaje.

Si esperas que el usuario escriba un entero, puedes intentar convertir el valor de retorno a `int` usando la función `int()`:

```
>>> prompt = '¿Cual es la velocidad de vuelo de una golondrina sin
carga?\n'
>>> velocidad = input(prompt)
¿Cual es la velocidad de vuelo de una golondrina sin carga?
17
>>> int(velocidad)
17
>>> int(velocidad) + 5
22
```

Pero si el usuario escribe algo que no sea una cadena de dígitos, obtendrás un error:

```
>>>velocidad = input(prompt)
¿Cual es la velocidad de vuelo de una golondrina sin carga?
¿Te refieres a una golondrina africana o a una europea?
>>>int(velocidad)
ValueError: invalid literal for int()
```

Veremos cómo controlar este tipo de errores más adelante.

6. Comentarios en Python

Como cualquier otro lenguaje de programación, Python permite escribir comentarios en el código. Para añadir un comentario a tu código simplemente comienza una línea con el carácter `#`:

```
# Esta línea es un comentario**
a = 5
# Resultado de multiplicar a por 2
print(a * 2)
```

Los comentarios son ignorados por el intérprete de Python. Solo tienen sentido para los programadores.

6.1. Comentarios de varias líneas

Para escribir comentarios que ocupan varias líneas, simplemente escribe cada una de las líneas anteponiendo el carácter #:

```
# Este comentario ocupa  
# 2 líneas
```

6.2. Docstrings

Los docstrings son un tipo de comentarios especiales que se usan para documentar un módulo, función, clase o método. En realidad, son la primera sentencia de cada uno de ellos y se encierran entre tres comillas simples o dobles.

Los docstrings son utilizados para generar la documentación de un programa. Además, suelen utilizarlos los entornos de desarrollo para mostrar la documentación al programador de forma fácil e intuitiva.

```
def suma(a, b):  
    """Esta función devuelve la suma de los parámetros a y b"""  
    return a + b
```

7. Palabras reservadas de Python

Python tiene una serie de palabras clave **reservadas**, por tanto, **no pueden usarse como nombres de variables, funciones, etc.**

Estas palabras clave se utilizan para definir la sintaxis y estructura del lenguaje Python.

La lista de palabras reservadas es la siguiente:

and, as, assert, break, class, continue, def, del, elif, else, except, False, finally, for, from, global, if, import, in, is, lambda, None, nonlocal, not, or, pass, raise, return, True, try, yield, while y with

8. Convenciones de nombres en Python

A la hora de nombrar una variable, una función, un módulo, una clase, etc. en Python, siempre se siguen las siguientes reglas y recomendaciones:

- Un identificador puede ser cualquier combinación de letras (mayúsculas y minúsculas), números y el carácter guión bajo (_).
- Un identificador no puede comenzar por un número.
- A excepción de los nombres de clases, es una convención que todos los identificadores se escriban en minúsculas, separando las palabras con el guión bajo. Ejemplos: contador, suma_enteros.

- Es una convención que los nombres de clases sigan la notación *Camel Case*, es decir, todas las letras en minúscula a excepción del primer carácter de cada palabra, que se escribe en mayúscula. Ejemplos: `Coche`, `VehiculoMotorizado`.
- No se pueden usar como identificadores las palabras reservadas.
- Como recomendación, usa identificadores que sean expresivos. Por ejemplo, `contador` es mejor que simplemente `c`.
- Python diferencia entre mayúsculas y minúsculas, de manera que `variable_1` y `Variable_1` son dos identificadores totalmente diferentes.

9. Depuración

En este punto, el error de sintaxis que es más probable que cometas será intentar utilizar nombres de variables no válidos, como `class` y `yield`, que son palabras clave, o `odd~job` y `US$`, que contienen caracteres no válidos.

Si pones un espacio en un nombre de variable, Python cree que se trata de dos operandos sin ningún operador:

```
>>> bad name = 5
SyntaxError: invalid syntax
>>> month = 09
      File "<stdin>", line 1
        month = 09
                ^
SyntaxError: invalid token
```

Para la mayoría de errores de sintaxis, los mensajes de error no ayudan mucho. Los mensajes más comunes son `SyntaxError: invalid syntax` y `SyntaxError: invalid token`, ninguno de los cuales resulta muy informativo.

El runtime error (error en tiempo de ejecución) que es más probable que obtengas es un “use before def” (uso antes de definir); que significa que estás intentando usar una variable antes de que le hayas asignado un valor. Eso puede ocurrir si escribes mal el nombre de la variable:

```
>>> principal = 327.68
>>> interest = principle * rate
NameError: name 'principle' is not defined
```

Los nombres de las variables son sensibles a mayúsculas, así que `LaTeX` no es lo mismo que `latex`.

En este punto, la causa más probable de un error semántico es el orden de las operaciones. Por ejemplo, para evaluar $\frac{1}{2\pi}$, puedes sentirte tentado a escribir

```
>>> 1.0 / 2.0 * pi
```

Pero la división se evalúa antes, ¡así que obtendrás $\pi/2$, que no es lo mismo! No hay forma de que Python sepa qué es lo que querías escribir exactamente, así que en este caso no obtienes un mensaje de error; simplemente obtienes una respuesta incorrecta.

10. Mas sobre Operadores

Ya hablamos sobre los operadores, los operadores son símbolos reservados por el propio lenguaje que se utilizan para llevar a cabo operaciones sobre uno, dos o más elementos llamados operandos. Los operandos pueden ser variables, literales, el valor devuelto por una expresión o el valor devuelto por una función.

El ejemplo más típico que siempre viene a la mente es el operador suma, +, que se utiliza para obtener la suma aritmética de dos valores:

```
>>>9 + 1 # 9 y 1 son los operandos
10 # 10 es el resultado
```

10.1. Operador de concatenación de cadenas de caracteres

Una de las operaciones más básicas cuando se trabaja con cadenas de caracteres es la concatenación. Esto consiste en unir dos cadenas en una sola, siendo el resultado un nuevo *string*.

La forma más simple de concatenar dos cadenas en Python es utilizando el operador de concatenación +:

```
>>> hola = 'Hola'
>>> python = 'Pythonista'
>>> hola_python = hola + ' ' + python # concatenamos 3 strings
>>> print(hola_python)
Hola Pythonista
```

10.2. Operadores lógicos o booleanos

A la hora de operar con valores booleanos, tenemos a nuestra disposición los operadores `and`, `or` y `not`.

❑ **IMPORTANTE:** Las operaciones `and`, `or` y `not` realmente no devuelven `True` o `False`, sino que devuelven uno de los operandos como veremos en el cuadro de abajo.

A continuación, te muestro cómo funcionan los operadores booleanos (en orden de preferencia ascendente):

Operación	Resultado	Descripción
<code>a or b</code>	Si <code>a</code> se evalúa a falso, entonces devuelve <code>b</code> , si no devuelve <code>a</code>	Solo se evalúa el segundo operando si el primero es falso
<code>a and b</code>	Si <code>a</code> se evalúa a falso, entonces devuelve <code>a</code> , si no devuelve <code>b</code>	Solo se evalúa el segundo operando si el primero es verdadero
<code>not a</code>	Si <code>a</code> se evalúa a falso, entonces devuelve <code>True</code> , si no devuelve <code>False</code>	Tiene menos prioridad que otros operadores no booleanos

Ejemplos:

```
>>> x = True
>>> y = False
>>> x or y
True
>>> x and y
False
>>> not x
False
>>> x = 0
>>> y = 10
>>> x or y
10
>>> x and y
0
>>> not x
True
```

10.3. Operadores de comparación

Los operadores de comparación se utilizan, como su nombre indica, para comparar dos o más valores. El resultado de estos operadores siempre es `True` o `False`.

Operador	Descripción
----------	-------------

>	Mayor que. <code>True</code> si el operando de la izquierda es estrictamente mayor que el de la derecha; <code>False</code> en caso contrario.
>=	Mayor o igual que. <code>True</code> si el operando de la izquierda es mayor o igual que el de la derecha; <code>False</code> en caso contrario.
<	Menor que. <code>True</code> si el operando de la izquierda es estrictamente menor que el de la derecha; <code>False</code> en caso contrario.
<=	Menor o igual que. <code>True</code> si el operando de la izquierda es menor o igual que el de la derecha; <code>False</code> en caso contrario.
==	Igual. <code>True</code> si el operando de la izquierda es igual que el de la derecha; <code>False</code> en caso contrario.
!=	Distinto. <code>True</code> si los operandos son distintos; <code>False</code> en caso contrario.

Ejemplos:

```
>>> x = 9
>>> y = 1
>>> x < y
False
>>> x > y
True
>>> x == y
False
```


10.3.1. Consideraciones sobre los operadores de comparación

Los objetos de diferentes tipos, excepto los tipos numéricos, nunca se comparan igual. El operador `==` siempre está definido, pero para algunos tipos de objetos (por ejemplo, objetos de clase) es equivalente a [is](#).

Las instancias no idénticas de una clase normalmente se comparan como no iguales a menos que la clase defina el método `__eq__()`.

Las instancias de una clase no se pueden ordenar con respecto a otras instancias de la misma clase u otros tipos de objeto, a menos que la clase defina los métodos `__lt__()`, `__gt__()`.

Los operadores de comparación se pueden concatenar. Ejemplo:

```
# Las comparaciones siguientes son idénticas
>>> x = 9
>>> 1 < x and x < 20
True
>>> 1 < x < 20
True
```

10.4. Operadores aritméticos en Python

En cuanto a los operadores aritméticos, estos permiten realizar las diferentes operaciones aritméticas del álgebra: suma, resta, producto, división, ... Estos operadores Python son de los más utilizados. El listado completo es el siguiente:

Operador Descripción

+	Suma dos operandos.
-	Resta al operando de la izquierda el valor del operando de la derecha. Utilizado sobre un único operando, le cambia el signo.
*	Producto/Multiplicación de dos operandos.
/	Divide el operando de la izquierda por el de la derecha (el resultado siempre es unfloat).
%	Operador módulo. Obtiene el resto de dividir el operando de la izquierda por el de la derecha.
//	Obtiene el cociente entero de dividir el operando de la izquierda por el de la derecha.
**	Potencia. El resultado es el operando de la izquierda elevado a la potencia del operando de la derecha.

Ejemplos:

```
>>> x = 7
```

```

>>> y = 2
>>> x + y # Suma
9
>>> x - y # Resta
5
>>> x * y # Producto
14
>>> x / y # División
3.5
>>> x % y # Resto
1
>>> x // y # Cociente
3
>>> x ** y # Potencia
49

```

10.5. Operadores a nivel de bits

Los operadores a nivel de bits actúan sobre los operandos como si fueran una cadena de dígitos binarios. Como su nombre indica, actúan sobre los operandos bit a bit. Son los siguientes:

Operación Descripción

$x y$	or bit a bit de x e y.
$x ^ y$	or exclusivo bit a bit de x e y.
$x \& y$	and bit a bit de x e y.
$x \ll n$	Desplaza x n bits a la izquierda.
$x \gg n$	Desplaza x n bits a la derecha.
$\sim x$	not x. Obtiene los bits de x invertidos.

Supongamos que tenemos el entero 2 (en bits es 00010) y el entero 7 (00111). El resultado de aplicar las operaciones anteriores es:

```

>>> x = 2
>>> y = 7
>>> x | y
7
>>> x ^ y
5
>>> x & y
2
>>> x << 1
4
>>> x >> 1
1
>>> ~x
-3

```

10.6. Operadores de asignación

El operador de asignación se utiliza para asignar un valor a una variable. Como te he mencionado en otras secciones, este operador es el signo `=`.

Además del operador de asignación, existen otros operadores de asignación compuestos que realizan una operación básica sobre la variable a la que se le asigna el valor.

Por ejemplo, `x += 1` es lo mismo que `x = x + 1`. Los operadores compuestos realizan la operación que hay antes del signo igual, tomando como operandos la propia variable y el valor a la derecha del signo igual.

A continuación, aparece la lista de todos los operadores de asignación compuestos:

Operador	Ejemplo	Equivalencia
<code>+=</code>	<code>x += 2</code>	<code>x = x + 2</code>
<code>-=</code>	<code>x -= 2</code>	<code>x = x - 2</code>
<code>*=</code>	<code>x *= 2</code>	<code>x = x * 2</code>
<code>/=</code>	<code>x /= 2</code>	<code>x = x / 2</code>
<code>%=</code>	<code>x %= 2</code>	<code>x = x % 2</code>
<code>//=</code>	<code>x //= 2</code>	<code>x = x // 2</code>
<code>**=</code>	<code>x **= 2</code>	<code>x = x ** 2</code>
<code>&=</code>	<code>x &= 2</code>	<code>x = x & 2</code>
<code> =</code>	<code>x = 2</code>	<code>x = x 2</code>
<code>^=</code>	<code>x ^= 2</code>	<code>x = x ^ 2</code>
<code>>>=</code>	<code>x >>= 2</code>	<code>x = x >> 2</code>
<code><<=</code>	<code>x <<= 2</code>	<code>x = x << 2</code>

10.7. Operadores de pertenencia

Los operadores de pertenencia se utilizan para comprobar si un valor o variable se encuentran en una secuencia (`list`, `tuple`, `dict`, `set` o `str`).

Todavía no hemos visto estos tipos, pero son operadores muy utilizados.

Operador	Descripción
<code>in</code>	Devuelve <code>True</code> si el valor se encuentra en una secuencia; <code>False</code> en caso contrario.

Operador Descripción

`not in` Devuelve True si el valor no se encuentra en una secuencia; False en caso contrario.

A continuación, vemos unos ejemplos que son muy intuitivos:

```
>>> lista = [1, 3, 2, 7, 9, 8, 6]
>>> 4 in lista
False
>>> 3 in lista
True
>>> 4 not in lista
True
```

10.8. Operadores de identidad

Por último, los operadores de identidad se utilizan para comprobar si dos variables son, o no, el mismo objeto.

Operador Descripción

`is` Devuelve True si ambos operandos hacen referencia al mismo objeto; False en caso contrario.

`is not` Devuelve True si ambos operandos no hacen referencia al mismo objeto; False en caso contrario.

□ **Recuerda:** Para conocer la identidad de un objeto se usa la función `id()`.

Ejemplos:

```
>>> x = 4
>>> y = 2
>>> lista = [1, 5]
>>> x is lista
False
>>> x is y
False
>>> x is 4
True
```

10.9. Prioridad de los operadores en Python

Como ya dijimos, al igual que ocurre en las matemáticas, los operadores en Python tienen un orden de prioridad. Este orden es el siguiente, de menos prioritario a más prioritario: asignación; operadores booleanos; operadores de comparación, identidad y pertenencia; a nivel de bits y finalmente los aritméticos (con el mismo orden de prioridad que en las matemáticas).

Este orden de prioridad se puede alterar con el uso de los paréntesis `()`:

```
>>> x = 5
>>> y = 2
>>> z = x + 3 * y # El producto tiene prioridad sobre la suma
>>> z
11
>>> z = (x + 3) * y # Los paréntesis tienen prioridad
>>> z
16
```

Fuente

- [Pagina de Juan Jose Lozano Gomez sobre Python](#)
- [Python for Everybody](#)
- [Elementos de un programa de Python](#)

1.3.-Tipos de datos

1.3. Tipos de datos

En Python, como en otros lenguajes de programación, los datos se clasifican en distintos tipos según su naturaleza y el tipo de operaciones que se pueden realizar con ellos. Los tipos de datos primitivos simples son aquellos que no se pueden descomponer en partes más pequeñas y son los que se utilizan para representar los valores más básicos. Los tipos de datos primitivos compuestos son aquellos que se pueden descomponer en partes más pequeñas y son los que se utilizan para representar estructuras más complejas.

1. Tipos de datos primitivos simples

Los tipos de datos primitivos simples son aquellos que NO se pueden descomponer en partes más pequeñas y son los que se utilizan para representar los valores más básicos. Los tipos de datos primitivos simples en Python son:

- **Números** (numbers): Secuencia de dígitos (pueden incluir el - para negativos y el . para decimales) que representan números. **Ejemplo** . 0, -1, 3.1415.
- **Cadenas** (strings): Secuencia de caracteres alfanuméricos que representan texto. Se escriben entre comillas simples o dobles. **Ejemplo** . 'Hola', "Adiós".
- **Booleanos** (boolean): Contiene únicamente dos elementos `True` y `False` que representan los valores lógicos verdadero y falso respectivamente.

2. Tipos de datos primitivos compuestos (contenedores)

Los tipos de datos primitivos compuestos son aquellos que se pueden descomponer en partes más pequeñas y son los que se utilizan para representar estructuras más complejas. Profundizaremos más adelante en ellos. Los tipos de datos primitivos compuestos en Python son:

- **Listas** (lists): Colecciones de objetos que representan secuencias ordenadas de objetos de distintos tipos. Se representan con corchetes y los elementos se separan por comas. **Ejemplo** . [1, "dos", [3, 4], True].
- **Tuplas** (tuples). Colecciones de objetos que representan secuencias ordenadas de objetos de distintos tipos. A diferencia de las listas son inmutables, es decir, que no cambian durante la ejecución. Se representan mediante paréntesis y los elementos se separan por comas. **Ejemplo** . (1, 'dos', 3)
- **Diccionarios** (dictionaries): Colecciones de objetos con una clave asociada. Se representan con llaves, los pares separados por comas y cada par contiene una clave y un objeto asociado separados por dos puntos. **Ejemplo** . {'pi':3.1416, 'e':2.718}.

3. Clase de un dato (`type()`)

La clase a la que pertenece un dato se obtiene con el comando `type()`

```
>>> type(1)
<class 'int'>
```

```
>>> type("Hola")
<class 'str'>
>>> type([1, "dos", [3, 4], True])
<class 'list'>
>>> type({'pi':3.1416, 'e':2.718})
<class 'dict'>
>>> type((1, 'dos', 3))
<class 'tuple'>
```

4. Números (clases `int` y `float`)

Secuencia de dígitos (pueden incluir el - para negativos y el . para decimales) que representan números. Pueden ser enteros (`int`) o reales (`float`).

```
>>> type(1)
<class 'int'>
>>> type(-2)
<class 'int'>
>>> type(2.3)
<class 'float'>
```

4.1. Operadores aritméticos

- Operadores aritméticos: + (suma), - (resta), * (producto), / (cociente), // (cociente división entera), % (resto división entera), ** (potencia).

Orden de prioridad de evaluación:

- 1 Funciones predefinidas
- 2 Potencias
- 3 Productos y cocientes
- 4 Sumas y restas

Se puede saltar el orden de evaluación utilizando paréntesis ().

```
>>> 2+3
5
>>> 5*-2
-10
>>> 5/2
2.5
>>> 5//2
2
>>> (2+3)**2
25
```

4.2. Operadores lógicos con números

Devuelven un valor lógico o booleano.

- Operadores lógicos: == (igual que), > (mayor que), < (menor que), >= (mayor o igual que), <= (menor o igual que), != (distinto de).

```
>>> 3==3
True
>>> 3.1<=3
False
>>> -1!=1
True
```

5. Cadenas (clase `str`)

Secuencia de caracteres alfanuméricos que representan texto. Se escriben entre comillas sencillas `'` o dobles `"`.

```
'Python'
"123"
'True'
# Cadena vacía
''
# Cadena con un espacio en blanco
' '
# Cambio de línea
'\n'
# Tabulador
'\t'
```

5.1. Acceso a los elementos de una cadena

Cada carácter tiene asociado un índice que permite acceder a él.

Cadena	P	y	t	h	o	n
Índice positivo	0	1	2	3	4	5
Índice negativo	-6	-5	-4	-3	-2	-1

- `c[i]` devuelve el carácter de la cadena `c` con el índice `i`.

El índice del primer carácter de la cadena es 0.

También se pueden utilizar índices negativos para recorrer la cadena del final al principio.

El índice del último carácter de la cadena es -1.

```
>>> 'Python'[0]
'P'
>>> 'Python'[1]
'y'
>>> 'Python'[-1]
'n'
>>> 'Python'[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```


5.2. Subcadenas

- `c[i:j:k]` : Devuelve la subcadena de `c` desde el carácter con el índice `i` hasta el carácter anterior al índice `j`, tomando caracteres cada `k`.

```
>>> 'Python'[1:4]
'yth'
>>> 'Python'[1:1]
''
>>> 'Python'[2:]
'thon'
>>> 'Python'[:-2]
'Pyth'
>>> 'Python'[: ]
'Python'
>>> 'Python'[0:6:2]
'Pto'
```

5.3. Operaciones con cadenas

- `c1 + c2` : Devuelve la cadena resultado de concatenar las cadenas `c1` y `c2`.
- `c * n` : Devuelve la cadena resultado de concatenar `n` copias de la cadena `c`.
- `c1 in c2` : Devuelve `True` si `c1` es una cadena contenida en `c2` y `False` en caso contrario.
- `c1 not in c2` : Devuelve `True` si `c1` es una cadena no contenida en `c2` y `False` en caso contrario.

```
>>> 'Me gusta ' + 'Python'
'Me gusta Python'
>>> 'Python' * 3
'PythonPythonPython'
>>> 'y' in 'Python'
True
>>> 'tho' in 'Python'
True
>>> 'to' not in 'Python'
True
```

5.4. Operaciones de comparación de cadenas

- `c1 == c2` : Devuelve `True` si la cadena `c1` es igual que la cadena `c2` y `False` en caso contrario.
- `c1 > c2` : Devuelve `True` si la cadena `c1` sucede a la cadena `c2` y `False` en caso contrario.
- `c1 < c2` : Devuelve `True` si la cadena `c1` antecede a la cadena `c2` y `False` en caso contrario.
- `c1 >= c2` : Devuelve `True` si la cadena `c1` sucede o es igual a la cadena `c2` y `False` en caso contrario.
- `c1 <= c2` : Devuelve `True` si la cadena `c1` antecede o es igual a la cadena `c2` y `False` en caso contrario.
- `c1 != c2` : Devuelve `True` si la cadena `c1` es distinta de la cadena `c2` y `False` en caso contrario.

Utilizan el orden establecido en el [código ASCII](#) .

```
>>> 'Python' == 'python'
False
>>> 'Python' < 'python'
True
>>> 'a' > 'Z'
True
>>> 'A' >= 'Z'
False
>>> '' < 'Python'
True
```

5.5. Funciones de cadenas

- `len(c)` : Devuelve el número de caracteres de la cadena `c`.
- `min(c)` : Devuelve el carácter menor de la cadena `c`.
- `max(c)` : Devuelve el carácter mayor de la cadena `c`.
- `c.upper()` : Devuelve la cadena con los mismos caracteres que la cadena `c` pero en mayúsculas.
- `c.lower()` : Devuelve la cadena con los mismos caracteres que la cadena `c` pero en minúsculas.
- `c.title()` : Devuelve la cadena con los mismos caracteres que la cadena `c` con el primer carácter en mayúsculas y el resto en minúsculas.
- `c.split(delimitador)` : Devuelve la lista formada por las subcadenas que resultan de partir la cadena `c` usando como delimitador la cadena `delimitador`. Si no se especifica el delimitador utiliza por defecto el espacio en blanco.

```
>>> len('Python')
6
>>> min('Python')
'P'
>>> max('Python')
'y'
>>> 'Python'.upper()
'PYTHON'
>>> 'A,B,C'.split(',')
['A', 'B', 'C']
>>> 'I love Python'.split()
['I', 'love', 'Python']
```

5.6. Cadenas formateadas (`format()`)

- `c.format(valores)`: Devuelve la cadena `c` tras sustituir los valores de la secuencia `valores` en los marcadores de posición de `c`. Los marcadores de posición se indican mediante llaves `{}` en la cadena `c`, y el reemplazo de los valores se puede realizar por posición, indicando en número de orden del valor dentro de las llaves, o por nombre, indicando el nombre del valor, siempre y cuando los valores se pasen con el formato `nombre = valor`.

```
>>> 'Un {} vale {} {}'.format('€', 1.12, '$')
'Un € vale 1.12 $'
>>> 'Un {2} vale {1} {0}'.format('€', 1.12, '$')
'Un $ vale 1.12 €'
>>> 'Un {moneda1} vale {cambio} {moneda2}'.format(moneda1 = '€', cambio
= 1.12, moneda2 = '$')
'Un € vale 1.12 $'
```

Los marcadores de posición, aparte de indicar la posición de los valores de reemplazo, pueden indicar también el formato de estos. Para ello se utiliza la siguiente sintaxis:

- `{:n}` : Alinea el valor a la izquierda rellenando con espacios por la derecha hasta los `n` caracteres.
- `{:>n}` : Alinea el valor a la derecha rellenando con espacios por la izquierda hasta los `n` caracteres.
- `{:^n}` : Alinea el valor en el centro rellenando con espacios por la izquierda y por la derecha hasta los `n` caracteres.
- `{:nd}` : Formatea el valor como un número entero con `n` caracteres rellenando con espacios blancos por la izquierda.
- `{:n.mf}` : Formatea el valor como un número real con un tamaño de `n` caracteres (incluido el separador de decimales) y `m` cifras decimales, rellenando con espacios blancos por la izquierda.

```
>>> 'Hoy es {:^10}, mañana {:10} y pasado {:>10}'.format('lunes',
'martes', 'miércoles')
'Hoy es   lunes   , mañana martes       y pasado  miércoles'
>>> 'Cantidad {:5d}'.format(12)
'Cantidad    12'
>>> 'Pi vale {:8.4f}'.format(3.141592)
'Pi vale    3.1416'
```

6. Datos lógicos o booleanos (clase `bool`)

Contiene únicamente dos elementos `True` y `False` que representan los valores lógicos verdadero y falso respectivamente.

`False` tiene asociado el valor 0 y `True` tiene asociado el valor 1.

6.1. Operaciones con valores lógicos

- Operadores lógicos: `==` (igual que), `>` (mayor), `<` (menor), `>=` (mayor o igual que), `<=` (menor o igual que), `!=` (distinto de).
- `not b` (negación) : Devuelve `True` si el dato booleano `b` es `False` , y `False` en caso contrario.
- `b1 and b2` : Devuelve `True` si los datos booleanos `b1` y `b2` son `True`, y `False` en caso contrario.
- `b1 or b2` : Devuelve `True` si alguno de los datos booleanos `b1` o `b2` son `True`, y `False` en caso contrario.

6.2. Tabla de verdad

<code>x</code>	<code>y</code>	<code>not x</code>	<code>x and y</code>	<code>x or y</code>
<code>False</code>	<code>False</code>	<code>True</code>	<code>False</code>	<code>False</code>
<code>False</code>	<code>True</code>	<code>True</code>	<code>False</code>	<code>True</code>
<code>True</code>	<code>False</code>	<code>False</code>	<code>False</code>	<code>True</code>
<code>True</code>	<code>True</code>	<code>False</code>	<code>True</code>	<code>True</code>

```
>>> not True
False
>>> False or True
```

```
True
>>> True and False
False
>>> True and True
True
```

6.3. Conversión de datos primitivos simples

En algunos casos, es necesario convertir un dato de un tipo a otro. Las conversiones de datos en Python se pueden clasificar en dos tipos:

1. **Conversiones explícitas:** Son aquellas en las que se especifica manualmente el tipo al que se quiere convertir el dato.
2. **Conversiones implícitas:** Son aquellas en las que Python realiza la conversión de forma automática durante la ejecución del programa.

6.3.1. Conversiones explícitas

Las siguientes funciones permiten convertir un dato de un tipo a otro, siempre y cuando la conversión sea válida:

- **int()** convierte a entero.
- Ejemplo:
 - `int('12')` # 12
 - `int(True)` # 1
 - `int('c')` # Error
- **float()** convierte a número real (flotante).
- Ejemplo:
 - `float('3.14')` # 3.14
 - `float(True)` # 1.0
 - `float('III')` # Error
- **str()** convierte a cadena.
- Ejemplo:
 - `str(3.14)` # '3.14'
 - `str(True)` # 'True'
- **bool()** convierte a booleano.
- Ejemplo:
 - `bool('0')` # False
 - `bool('3.14')` # True
 - `bool('')` # False
 - `bool('Hola')` # True

6.3.2. Conversiones implícitas

Python realiza ciertas conversiones de tipos de datos de manera implícita durante las operaciones. Las conversiones implícitas que Python efectúa son las siguientes:

- **De int a float:** Si se realiza una operación aritmética entre un número entero y un número real, el entero se convierte automáticamente en un número real.

Ejemplo:

```
2 + 3.0 # 5.0 (int convertido a float)
```

- **De float a int:** Python no convierte automáticamente un `float` a `int` en una operación aritmética, ya que esto implicaría pérdida de precisión. Para convertir un `float` a `int`, se debe realizar una conversión explícita utilizando `int()`.
- **No hay conversión implícita de int a str ni de float a str:** Para concatenar un número con una cadena, se debe convertir explícitamente el número a cadena utilizando `str()`. Python no realiza esta conversión de manera implícita.

Ejemplo:

```
'Hola' + str(3) # 'Hola3'
```

- **No hay conversión implícita de str a int o float:** Si se necesita realizar operaciones aritméticas con una cadena que contiene un número, es necesario convertir la cadena explícitamente a `int` o `float` utilizando las funciones `int()` o `float()`.

Ejemplo:

```
2 + int('3') # 5 (conversión explícita de str a int)
2 + float('3.0') # 5.0 (conversión explícita de str a float)
```

En resumen:

- Las **conversiones implícitas** en Python ocurren principalmente en operaciones numéricas entre `int` y `float`.
- Para realizar operaciones con cadenas y números, es necesario realizar **conversiones explícitas** utilizando funciones como `str()`, `int()`, o `float()`.

Fuente

- [Aprende con Alf](#)
- [Pagina de Juan Jose Lozano Gomez sobre Python](#)
- [Documentación de Python](#)
- [Tipos de datos](#)

* Práctica 1.2: Primeros programas

P1.2 - Ejercicios

Ejercicio 1.2.1

Escribe un programa que pida el nombre del usuario para luego darle la bienvenida.

```
Escribe tu nombre: Juan
Hola, Juan.
```

Ejercicio 1.2.2

Escribe un programa para pedirle al usuario las horas de trabajo y el precio por hora y calcule el importe total del servicio.

```
Horas de trabajo: 6
Coste por hora: 10
Importe total: 60
```

Ejercicio 1.2.3

Suponiendo que se han ejecutado las siguientes sentencias de asignación:

```
ancho = 17
alto = 12.0
```

Para cada una de las expresiones siguientes, intenta adivinar el valor de la expresión y su tipo sin ejecutarlas en el intérprete:

1. `ancho / 2`
2. `ancho // 2`
3. `alto / 3`
4. `1 + 2 * 5`

Cuando termines comprueba con el intérprete si has acertado.

Ejercicio 1.2.4

Escribe un programa que le pida al usuario una temperatura en grados Celsius, la convierta a grados Fahrenheit e imprima por pantalla la temperatura convertida.

Ejercicio 1.2.5

Escribe un programa que pida el importe sin IVA de un artículo y el tipo de IVA a aplicar y calcule e imprima por pantalla el precio final del artículo.

Ejercicio 1.2.6

Escribe un programa que pida el importe final de un artículo y calcule e imprima por pantalla el IVA que se ha pagado y el importe sin IVA (suponiendo que se ha aplicado un tipo de IVA del 10%).

Ejercicio 1.2.7

Escribe un programa que solicite tres números al usuario y calcule e imprima por pantalla su suma.

Ejercicio 1.2.8

Escribir el programa del ejercicio 1.7 usando solamente dos variables diferentes.

Ejercicio 1.2.9

¿Es posible escribir el programa del ejercicio 1.7 sin usar variables? Intentalo.

Ejercicio 1.2.10

Escribir un programa que muestre por pantalla el resultado de la siguiente operación aritmética

$$\left(\frac{3+2}{2 \cdot 5} \right)^2$$

Ejercicio 1.2.11

Escribir un programa que lea un entero positivo, n , introducido por el usuario y después muestre en pantalla la suma de todos los enteros desde 1 hasta n . La suma de los n primeros enteros positivos puede ser calculada de la siguiente forma:

$$\text{suma} = \frac{n(n + 1)}{2}$$

Ejercicio 1.2.12

Escribir un programa que pida al usuario su peso (en kg) y estatura (en metros), calcule el índice de masa corporal y lo almacene en una variable, y muestre por pantalla la frase Tu índice de masa corporal es donde es el índice de masa corporal calculado redondeado con dos decimales.

Ejercicio 1.2.13

Escribir un programa que pida al usuario dos números enteros y muestre por pantalla los siguientes: "la división de n entre m da un cociente c y un resto r ", donde n y m son los números introducidos por el usuario, y c y r son el cociente y el resto de la división entera respectivamente.

Ejercicio 1.2.14

Una juguetería tiene mucho éxito en dos de sus productos: payasos y muñecas. Suele hacer venta por correo y la empresa de logística les cobra por peso de cada paquete así que deben calcular el peso de los payasos y muñecas que saldrán en cada paquete a demanda. Cada payaso pesa 112 g y cada muñeca 75 g. Escribir un programa que lea el número de payasos y muñecas vendidos en el último pedido y calcule el peso total del paquete que será enviado.

Ejercicio 1.2.15

Imagina que acabas de abrir una nueva cuenta de ahorros que te ofrece el 4% de interés al año. Estos ahorros debido a intereses, que no se cobran hasta finales de año, se te añaden al balance final de tu cuenta de ahorros. Escribir un programa que comience leyendo la cantidad de dinero depositada en la cuenta de ahorros, introducida por el usuario. Después el programa debe calcular y mostrar por pantalla la cantidad de ahorros tras el primer, segundo y tercer años. Redondear cada cantidad a dos decimales.

```
Calcula el interés: capital * (1 + interes)
```

Ejercicio 1.2.16

Una panadería vende barras de pan a 3.49€ cada una. El pan que no es el día tiene un descuento del 60%. Escribir un programa que comience leyendo el número de barras vendidas que no son del día. Después el programa debe mostrar el precio habitual de una barra de pan (establecido en el programa como una constante), el descuento que se le hace por no ser fresca y el coste final total de todas las barras no frescas.

Ejercicio 1.2.17

Escribir un programa que pregunte el nombre del usuario en la consola y un número entero e imprima por pantalla en líneas distintas el nombre del usuario tantas veces como el número introducido.

Ejercicio 1.2.18

Escribir un programa que pregunte el nombre completo del usuario en la consola y después muestre por pantalla el nombre completo del usuario tres veces, una con todas las letras minúsculas, otra con todas las letras mayúsculas y otra solo con la primera letra del nombre y de los apellidos en mayúscula. El usuario puede introducir su nombre combinando mayúsculas y minúsculas como quiera.

Ejercicio 1.2.19

Escribir un programa que pregunte el nombre del usuario en la consola y después de que el usuario lo introduzca muestre por pantalla "NOMBRE tiene n letras.", donde NOMBRE es el nombre de usuario en mayúsculas y n es el número de letras que tienen el nombre.

Ejercicio 1.2.20

Los teléfonos de una empresa tienen el siguiente formato prefijo-número-extension donde el prefijo es el código del país +34, y la extensión tiene dos dígitos (por ejemplo +34-913724710-56). Escribir un programa que pregunte por un número de teléfono con este formato y muestre por pantalla el número de teléfono sin el prefijo y la extensión.

Ejercicio 1.2.21

Escribir un programa que pida al usuario que introduzca una frase en la consola y muestre por pantalla la frase invertida.

Ejercicio 1.2.22

Escribir un programa que pida al usuario que introduzca una frase en la consola y una vocal, y después muestre por pantalla la misma frase pero con la vocal introducida en mayúscula.

Ejercicio 1.2.23

Escribir un programa que pregunte el correo electrónico del usuario en la consola y muestre por pantalla otro correo electrónico con el mismo nombre (la parte delante de la arroba @) pero con dominio ceu.es.

Ejercicio 1.2.24

Escribir un programa que pregunte por consola el precio de un producto en euros con dos decimales y muestre por pantalla el número de euros y el número de céntimos del precio introducido.

Ejercicio 1.2.25

Escribir un programa que pregunte al usuario la fecha de su nacimiento en formato dd/mm/aaaa y muestre por pantalla, el día, el mes y el año. Adaptar el programa anterior para que también funcione cuando el día o el mes se introduzcan con un solo carácter.

Ejercicio 1.2.26

Escribir un programa que pregunte por consola por los productos de una cesta de la compra, separados por comas, y muestre por pantalla cada uno de los productos en una línea distinta.

Ejercicio 1.2.27

Escribir un programa que pregunte el nombre de un producto, su precio y un número de unidades y muestre por pantalla una cadena con el nombre del producto seguido de su precio unitario con 6 dígitos enteros y 2 decimales, el número de unidades con tres dígitos y el coste total con 8 dígitos enteros y 2 decimales.

Ejercicio 1.2.28

Calcular el área de un triángulo a partir de tres lados

Ejercicio 1.2.29

Cálculo de un número aleatorio entre dos valores

Ejercicio 1.2.30

Escribir un programa que determine si un número es primo

Ejercicio 1.2.31

Mostrar todos los divisores de un número

Ejercicio 1.2.32

Calcular la serie de Fibonacci hasta un número dado