Introducción a la programación con Python

Bienvenida

Objetivos

- Conocer el alcance del módulo.
- Conocer los requerimientos del módulo.
- Conocer las herramientas que utilizaremos a lo largo del módulo.

Descripción

Durante este módulo se enseñan los aspectos más básicos y fundamentales para comenzar a programar software, que corresponden a los algoritmos y a los controles de flujo. Estos conocimientos son necesarios para programar en cualquier lenguaje de programación, no solo en Python. Luego se introducen contenidos más específicos sobre la programación utilizando Python.

¿Qué se requiere para este módulo?

Para este y el resto de los módulos, se trabajará utilizando Python versión 3.7.1 o superior mediante Anaconda, y un editor de texto.

Anaconda está disponible para Windows, macOS y Linux, por lo que se requiere cualquiera de estos sistemas operativaos. Su instalación se verá durante el desarrollo del módulo.

Introducción a la programación

Objetivos

- Entender en qué consiste la programación.
- Conocer el rol de un programador dentro de una organización.
- Conocer la definición del algoritmo y su importancia dentro de la programación.
- Conocer las entidades que forman parte de un diagrama de flujo.
- Conocer la relación entre pseudocódigo y diagramas de flujo.
- Conocer la importancia de la lógica independiente del lenguaje.
- Conocer la importancia del desarrollo del pensamiento lógico.

¿Qué es programar?

Cuando hablamos de programar, nos imaginamos un hacker sentado frente a un computador escribiendo código en una pantalla negra con letras verdes. Pero en realidad, este es un concepto mucho más amplio. En términos generales, consiste en crear un programa o software. Este proceso de creación es complejo, involucrando varias etapas, y quizá sea por esta complejidad que también se le llama a este proceso "Desarrollar", y a los programadores "Desarrolladores".

Programar es más que escribir código

La mayor parte del trabajo de un programador consiste en generar soluciones a problemas que surgen dentro de una organización. Para ésto, el programador busca descomponer el problema en varias partes y posteriormente implementar éstas soluciones.

Las soluciones propuestas por un programador se conocen como algoritmos, que se expresan en código. El objetivo de un programador es desarrollar e implementar soluciones mediante los algoritmos. El código es simplemente una de las formas en que se materializa la solución.

¿Qué es un algoritmo?

Un algoritmo no es más que es una serie de pasos finitos y ordenados para resolver un problema.

Ejemplo de algoritmo

Por ejemplo, si deseamos ensamblar un mueble es necesario seguir todos los pasos del manual de manera secuencial. Otro ejemplo típico es crear un origami a partir de un trozo de papel, o una receta de cocina.

Para poder indicar cómo preparar panqueques a otra persona, tenemos que indicar claramente todos los pasos que debe hacer de manera secuencial. El omitir un paso afecta el resultado.

RECETA DE PANQUEQUES	ALGORITMO
Ingredientes: - 1 taza de harina - 1 taza de leche - 1 huevo Preparación - Mezcle todos los ingredientes hasta tener una mezcla homogénea. - Vierta una porción de la mezcla en una sartén precalentada, esparciendo hasta tener una capa delgada de masa. - Espere 1 minuto y de vuelta la masa. - Espere otro minuto y retire el panqueque con la espátula. - Repite el proceso hasta terminar la mezcla.	1. Agregar 1 taza de harina en un bowl. 2. Agregar 1 taza de leche a la harina. 3. Agregar 1 huevo a los ingredientes previos. 4. Revolver y mezclar los 3 ingredientes. 5. Precalentar el sartén. 6. Agregar parte de la mezcla hasta cubrir el sartén y esparcir una capa delgada. 7. Esperar 1 minuto. 8. Dar vuelta la masa. 9. Esperar otro minuto. 10. Retirar el panqueque. 11. Repetir pasos del 6 al 10 hasta terminar la mezcla.

Escribiendo un algoritmo

Cuando generamos un algoritmo en el computador, debemos especificar paso a paso qué es lo que el computador debe realizar. Esto es similar a cuando deseamos hacer un origami, debemos seguir pasos.

Si el algoritmo utilizado para construir nuestro programa omite algún paso, o no sigue el orden correcto, nuestro programa fallará, o no cumplirá con el resultado esperado. Es por ello que saber el algoritmo, o pasos, para crear nuestro programa, es de suma importancia.

Formas de escribir un algoritmo

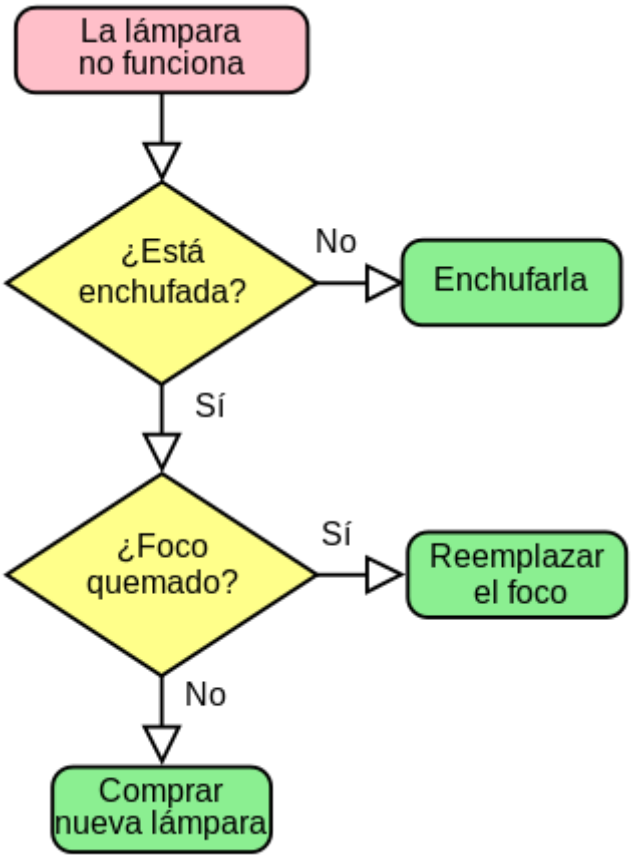
Existen distintas formas de implementar un algoritmo:

- Representarlo mediante un diagrama de flujo.
- Escribirlo en pseudocódigo.
- Escribirlo directamente en un lenguaje de programación.

Diagrama de flujo

El diagrama de flujo es una representación gráfica de los pasos detallados en un algoritmo. Permite visualizarlo y de esta forma reducir su complejidad.

Si queremos arreglar una lámpara, nuestro proceso algorítmico conlleva los siguientes pasos.

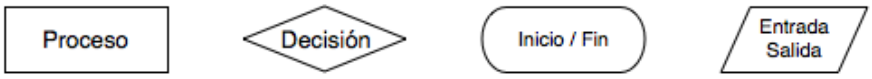


En algunos diagramas, como en el anterior, el fin es implícito; Al llegar al último paso, se supone que el algoritmo concluye. Esta distinción es necesaria para destacar que existen procesos donde no hay fines explícitamente declarados.

Símbolos de un diagrama de flujo

Cada uno de los símbolos en un diagrama de flujo representa alguna acción a realizar en el algoritmo.

- **Inicio y fin del programa**: Por defecto todo algoritmo debe tener una serie de elementos finitos. Éstos se declaran mediante el inicio y el fin del programa.
- **Datos de entrada y salida**: Cada paso que se genera dentro del algoritmo implica tomar algún dato y devolver otro dato.
- **Procesos (la instrucción que le damos a la máquina)**: Definición de los pasos a seguir en un algoritmo.
- **Decisiones**: Eventualmente el flujo lógico podrá encausar los siguientes pasos en base a la resolución de una decisión.



Los diagramas de flujo son importantes para un programador

Existen dos habilidades importantes que tendremos que aprender:

1. Crear diagramas de flujo.
2. Implementar código a partir de un diagrama de flujo.

Pseudocódigo

Otra forma de representar un algoritmo es mediante el pseudocódigo, que consiste en* una serie de frases que describen el flujo:

```
Algoritmo Suma
    Leer valor1
    Leer Valor2
    Mostrar valor1 + valor2
FinAlgoritmo
```

En pseudocódigo se utiliza la instrucción `leer` para especificar que el usuario tiene que ingresar un valor y `mostrar` para imprimir el valor en la pantalla.

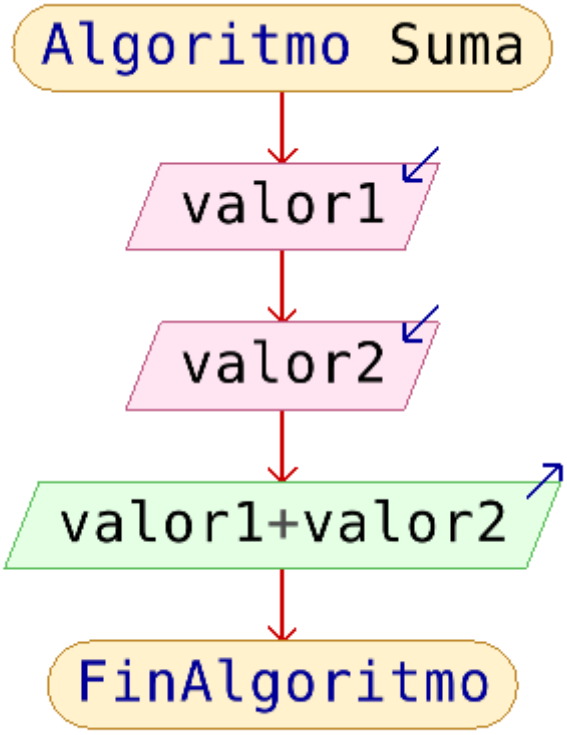
Escribiendo pseudocódigo

Si bien la especificación del pseudocódigo no es formal, existen algunos programas que nos permiten escribir pseudocódigo y ejecutarlo, como **PSEINT**.

Se puede experimentar con pseudocódigo descargando el programa de forma gratuita en la página oficial http://pseint.sourceforge.net/. *Algunas capturas de pantalla fueron obtenidas utilizando este programa*.

De pseudocódigo a diagrama de flujo

Transformar el pseudocódigo a un diagrama de flujo es sencillo y muy lineal, solo hay que reemplazar las instrucciones específicas por símbolos correspondientes y poner atención hacia dónde apuntan las flechas.



Todo diagrama de flujo puede transformase a pseudocódigo y viceversa, así como a un lenguaje de programación, aunque ésto no siempre será tan directo.

Ventajas del pseudocódigo

El pseudocódigo permite pensar en términos independientes al lenguaje de programación y permite concentrarnos en describir lo que estamos tratando de hacer y los pasos necesarios en lugar de cómo lograrlo.

Para programar aprenderemos que es vital saber cómo comunicarle al computador la secuencia de pasos a seguir. Escribir pseudocódigo y diagramas de flujo ayuda a reforzar esta habilidad.

¿Por qué es importante que sepamos esto?

Es importante porque los algoritmos y diagramas de flujo son **independientes del lenguaje**, es decir, podemos implementar un mismo algoritmo en diferentes lenguajes de programación.

Un algoritmo corresponde a una solución lógica y pone a prueba nuestro desarrollo del pensamiento lógico.

Enfrentándose a un problema

El desarrollo del pensamiento lógico es una habilidad **imprescindible** al momento de aprender a programar y está directamente relacionado con nuestra manera de solucionar problemas. Antes de escribir código, debemos abstraernos del código y pensar en el problema en los siguientes pasos:

1. Analizar el problema
2. Descomponer el problema en partes
3. Resolver el problema

Cuanto más desarrollemos nuestro pensamiento lógico, más rápido obtendremos soluciones a problemas cotidianos en programación y nuestros programas harán más a menudo lo que esperamos.

El pensamiento lógico y nuestra capacidad de abstracción son habilidades que desarrollaremos a lo largo del módulo.

Introducción a Python

Objetivos

- Conocer Python basics
- Conocer la posición de Python dentro de los lenguajes más conocidos.
- Conocer las ventajas de Python por sobre otros lenguajes.
- Conocer el ecosistema de empresas que utilizan Python.
- Conocer las distintas versiones de Python y cuál utilizaremos en el curso.
- Revisar el problema de versiones de Python.
- Conocer un administrador de versiones de Python.
- Instalar un administrador de versiones de Python.

¿Qué es Python?

Python es un lenguaje de programación muy flexible y potente. Fue creado por Guido Van Rossum en 1991, con su primera versión 1.0 lanzada en 1994. Dado la claridad y deliberada simpleza de su sintáxis, Python se ha transformado en uno de los lenguajes con mayores tasas de adopción y demanda tanto en la industria.

Python permite:

- Construir de forma sencilla aplicaciones web con manejos de bases de datos.
- Hacer análisis de datos y visualización de éstos.
- Realizar web-scrapping (Captura de datos de una página web).
- Crear videojuegos.
- Crear aplicaciones de escritorio.

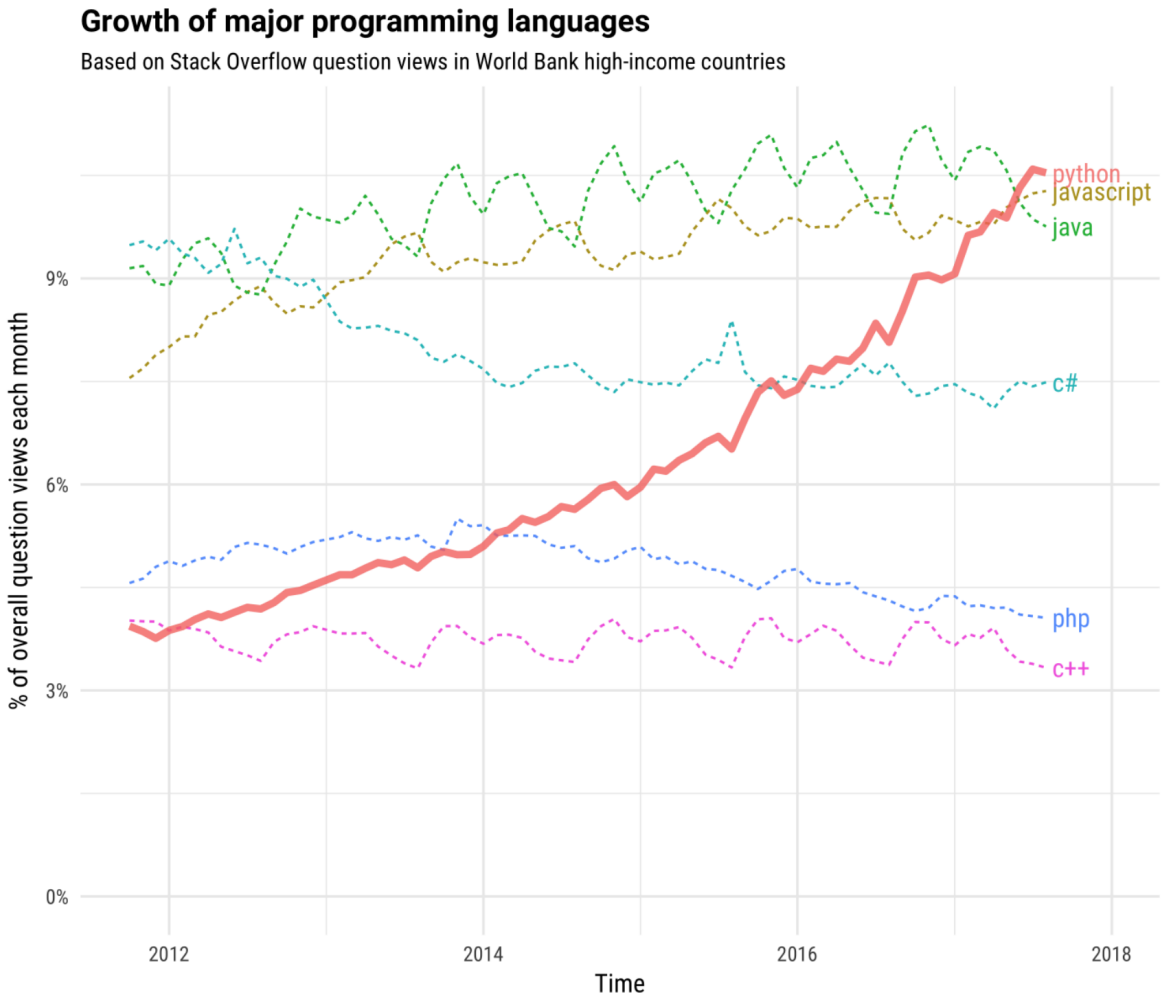
¿Es Python una buena elección para comenzar a programar?

La respuesta es **sí**.

La velocidad de la tasa de adopción se explica por varios factores directamente relacionados con el diseño de Python. Dada su simpleza sintáctica y similitud al inglés, ha logrado convencer a expertos de otras áreas que no sea programación (llámese matemáticas, lingüística, biología) a programar y diseñar rutinas para agilizar su trabajo. Dado el énfasis en la comunidad de Pythonistas, la cantidad de librerías y contribuciones lo han posicionado como una excelente primera alternativa para resolver tareas.

¿Qué tan relevante es Python en la industria?

En los últimos años, Python se ha caracterizado por ser el lenguaje de programación con una mayor tasa de adopción en comparación a sus principales competidores. Esto acorde a resultados de una encuesta de StackOverflow, uno de los principales foros de programación a nivel mundial, quienes procesaron los datos de sus usuarios.



Áreas donde se utiliza Python

Existen dos grandes áreas donde Python es uno de los principales competidores:

- **Desarrollo Web**: Páginas como YouTube, Instagram y Google implementan Python en sus servicios. El lenguaje ofrece librerías como Django y Flask que permiten desarrollar servicios webs complejos dentro de un marco de trabajo sencillo.
- **Ciencia de Datos y Aprendizaje de Máquinas**: Dada la deliberada simpleza sintáctica de Python, existe un gran desarrollo de librerías de análisis y preprocesamiento de datos por parte de la academia e industria para agilizar las rutinas de análisis. Librerías como Scikit-Learn y TensorFlow dominan la implementación de modelos predictivos en distintas áreas.

Python en Acción

¿Qué crees que hace la siguiente expresión?

```
for i in range(3):
    print("hip hip hooray!")
```

¿Cuál será el resultado de la siguiente expresión?

```
sum([1, 2, 3, 4])
```

Descargando e instalando Anaconda

Python y sus múltiples versiones

En este módulo trabajaremos con la versión 3.7 de Python, la última gran versión de Python que se liberó al público. Una de las características más importantes de esta versión es que **no es compatible con versiones anteriores** como la 2.x. Por tanto, es casi imposible correr código desarrollado en Python 2 en Python 3, sin tener que modificar su contenido.

¿Cómo podemos manejar distintas versiones de Python?

Una alternativa es desarrollar **ambientes virtuales**, los cuales funcionan como un contenedor aislado donde se puede declarar una versión específica de Python y librerías específicas. Mediante éstos, nos aseguramos que cualquier problema existente en el ambiente virtual no afecte a la máquina en su totalidad.

Para trabajar con una versión específica de Python, se debe trabajar con el administrador de versiones adecuado. Existen varios administradores de versiones de Python, como `pyenv`, `virtualenv` y `anaconda`. Para efectos prácticos, nosotros trabajaremos con `anaconda`, un administrador orientado al trabajo en ciencia de datos.

La instalación de Anaconda es fácil, sólo debemos ingresar a la página https://www.anaconda.com/download/ y seguir las instrucciones de instalador gráfico.

Utilizando Python

Objetivos

- Conocer y utilizar Python desde la consola para ejecutar código.
- Ejecutar scripts de Python desde la consola.
- Leer la salida de un script ejecutado.
- Conocer las limitantes de Python para la ejecución de código.
- Conocer y utilizar iPython.

Formas de trabajar con Python

Existen variadas formas de utilizar Python:

1. La primera es mediante el ambiente REPL (Read, Evaluate, Print Loop), que permite escribir código directamente e ir probando los resultados de forma inmediata. Esto resulta muy útil para probar código, pero no permite escribir un programa completo.

2. La segunda forma, y la que se utiliza tradicionalmente, es escribir una serie de instrucciones en un editor de texto, y posteriormente ejecutarlo. Al archivo de texto generado se le llama "script".

En este capítulo aprenderemos a trabajar de las dos formas.

Trabajando con Python desde la consola

Para ingresar a Python desde la consola, debemos entrar al terminal. Para ello abriremos la aplicación *Command Prompt* en Windows o *Terminal* en Linux/OSX. Al abrirla veremos un signo peso $ o un signo porcentual \%. Esto se conoce como el *prompt*, un símbolo que indica que el terminal está listo para recibir instrucciones.

Ingresando a Python

Desde el terminal, podemos ingresar a Python interactivo. Para ello, escribiremos `python` y posteriormente daremos `Enter`.

Si lo hicimos de forma correcta, deberíamos ver que el prompt cambió. Esto es porque ahora estamos dentro de otro programa `python`.

```
[isz:~] isz% python
Python 3.6.5 |Anaconda, Inc.| (default, Apr 26 2018, 08:42:37)
[GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE_401/final)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Probando Python

Dentro de este programa podemos escribir instrucciones válidas para Python. Por ejemplo, podemos escribir `2 + 2` y ver como resultado `4`.

```
[isz:~] isz% python
Python 3.6.5 |Anaconda, Inc.| (default, Apr 26 2018, 08:42:37)
[GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE_401/final)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 2 + 2
4
>>>
```

¿Qué son los otros números que aparecen en la pantalla?

Antes de probar otras operaciones, revisemos algunos detalles que son relevantes:

- El prompt en este caso se señala con `>>>`.
- En el inicio se lee `Python 3.6.5 | Anaconda Inc.`, que hace referencia a la versión de Python ocupada, y a qué distribución específica pertenece. En este caso, nuestra consola está en Python 3.6.5, y fue administrada por el gestor de ambientes `Anaconda`.
- Los retornos y resultados de toda expresión válidamente ejecutada no se preceden con símbolo alguno. En este caso, la expresión ejecutada corresponde a `2 + 2`, y el resultado se muestra en la línea siguiente, sin prompt.

Realizando nuestras primeras pruebas

Ejecuta en Python las siguientes instrucciones:

- ¿Qué se obtiene al realizar la operación `5 / 2`?
- ¿Qué se obtiene al realizar la operación `'hola' + 'mundo'`?
- ¿Qué se obtiene al realizar la operación `sum([1, 2, 3])`?

Saliendo de Python

Para salir de Python, debemos escribir `quit()`. De esta forma volveremos al prompt original del terminal.

Trabajando con Python desde el editor de texto

Existen varios editores de texto gratuitos como Atom, Visual Studio Code y Sublime Text. Éste último es de pago, pero ofrece una versión gratuita.

Puedes escoger el editor que más te guste. Si no sabes cual escoger, te sugerimos utilizar *Visual Studio Code* ya que es el que utilizaremos para los ejemplos del módulo.

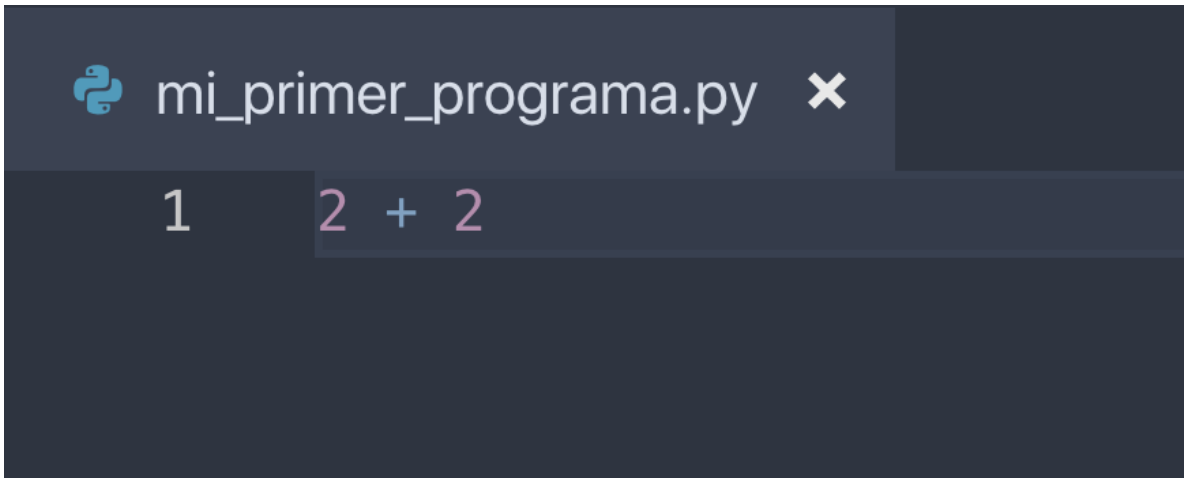
Extensión de los archivos `.py` (script)

Para escribir un programa desde cero en Python en un editor de texto, o `script`, solo necesitamos crear un archivo con extensión `.py`. En este archivo se deben escribir las líneas de código deseadas, y luego se debe *ejecutar* desde la terminal.

Nuestro primer programa en el editor

Para escribir nuestro primer programa en el editor de texto, vamos a seguir los siguientes pasos:

1. Abrir nuestro editor de texto.
2. Escribir una instrucción sencilla.
3. Guardar el archivo con extensión `.py`.



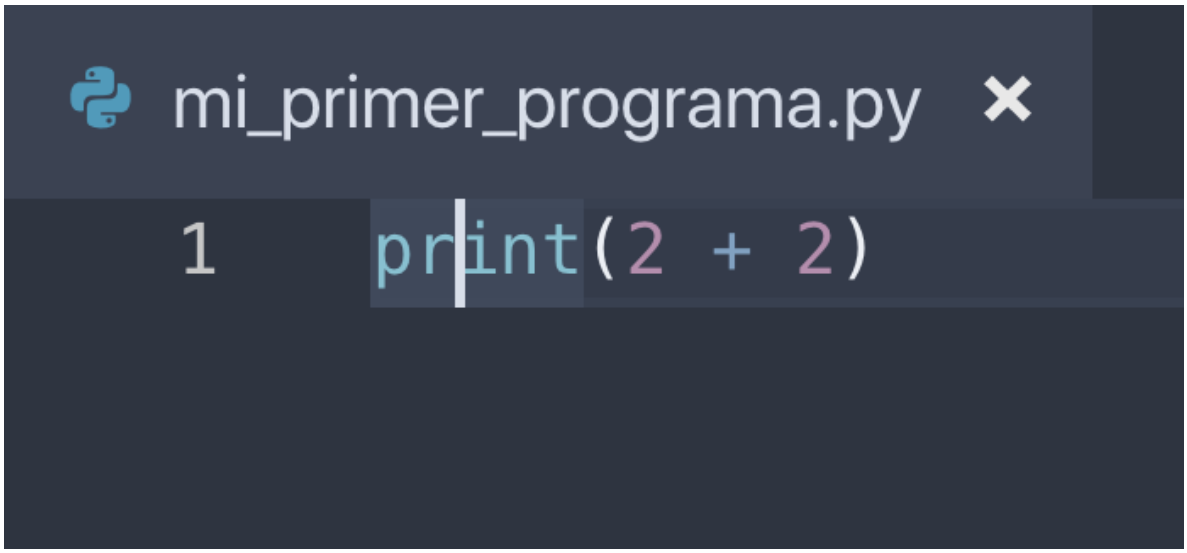
Ejecutando el programa

A continuación, debemos abrir un terminal en la misma ubicación donde se encuentra el archivo, y ejecutar `python nombre_archivo.py`

```
[isz:~/Desktop] isz% python mi_primer_programa.py
[isz:~/Desktop] isz%
```

Mostrando el resultado

Lamentablemente nuestro script no devuelve nada. Este comportamiento se debe a que cuando ejecutamos un script desde el terminal, por defecto Python no devuelve el resultado de cada línea. Si deseamos mostrar el resultado, debemos indicar a Python de manera explícita que muestre el resultado en pantalla. Esto lo logramos con la instrucción `print()`



Ejecutando el programa

Si ejecutamos nuevamente el script, ahora sí veremos el resultado en pantalla:

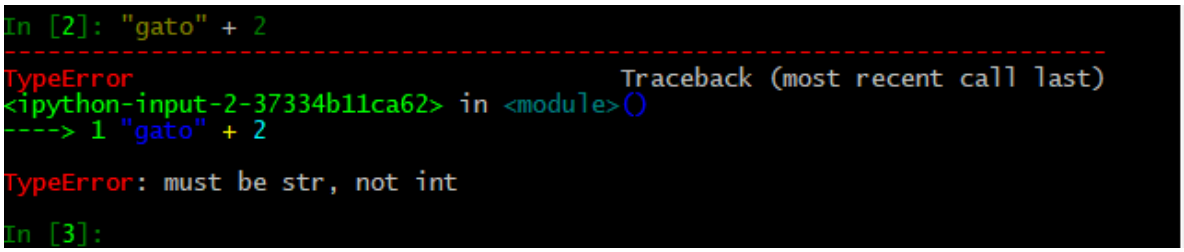
```
[isz:~] isz% cd ~/Desktop/
[isz:~/Desktop] isz% python mi_primer_programa.py
4
[isz:~/Desktop] isz%
```

Limitantes

Si bien Python se caracteriza por ser un lenguaje de programación de sintaxis simple, esto no significa que no tenga reglas para escribir el código. Por ejemplo, Python no nos permite sumar letras y números.

A lo largo del módulo y del curso, se irán viendo estas reglas en detalle según el contenido revisado. Una muy importante de recordar es respetar la identación al interior de controles de flujo y funciones, lo cual se verá más adelante.

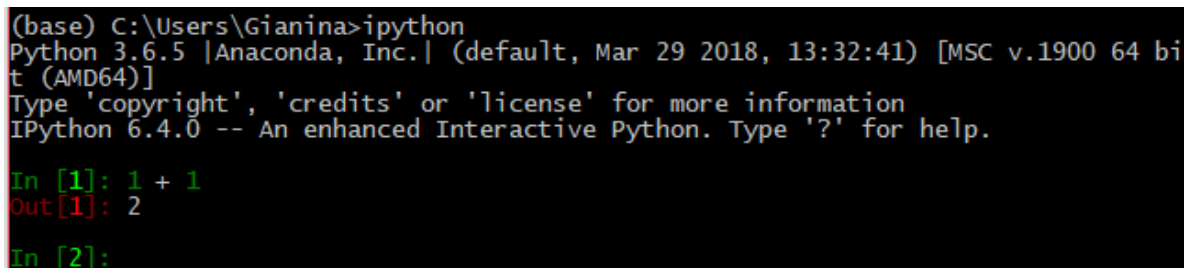
Siempre que Python no sea capaz de ejecutar una instrucción debido a un error de sintaxis, se mostrará un mensaje explicando el error.



Facilitar la escritura y evitar errores

Los editores de texto nos permiten facilitar la escritura del código, resaltando ciertas palabras que signifiquen instrucciones específicas con colores diferentes, o identando automáticamente cuando corresponda.

Otra opción, estando en la terminal, es utilizar el kernel `iPython` en lugar de `python`. Al igual que en un editor de texto, se resaltan distintos tipos de palabras o instrucciones en distintos colores, y también realiza identación de forma automática.



Elementos básicos de Python

Objetivos

- Definir comentarios, variables y constantes en Python.
- Conocer y utilizar operaciones aritméticas en Python.
- Manipular strings utilizando concatenación en Python.
- Manipular strings utilizando interpolación en Python.
- Manipular entrada y salida de datos utilizando `print` e `input` en Python.
- Diferenciar un error sintáctico de uno semántico.

Muchos de los términos utilizados en este capítulo están definidos en el glosario, por lo que se recomienda consultarlo en caso de existir duda sobre alguno.

Comentarios

Los comentarios son líneas de código que son ignoradas. Sirven para dar indicaciones y documentar nuestro código a otros usuarios.

```
# Esta línea es un comentario
# Los comentarios son ignorados
# Pueden ser una línea nueva
print(2 + 2)  # O puede acompañár una línea de códido existente
# print 2 + 3 Si comentamos al principio todo la línea será ignorada
```

```
4
```

Comentarios en múltiples líneas

Existe otra forma de hacer comentarios multilínea que no es tan utilizada. Consiste en envolver todo el comentario entre tres comillas dobles al inicio y al final. Esta forma es comúnmente utilizada para documentar el código.

```
"""
Comentario multilínea:
Python
lo
ignorará
"""

print("hola")
```

```
hola
```

Indicación importante

A lo largo del módulo utilizaremos comentarios en algunos ejemplos para representar los valores mostrados en pantalla.

Esto facilitará la lectura de ejemplos como:

```
print(2 + 2) # 4
```

Introducción a variables

Podemos entender las variables como **contenedores que pueden almacenar valores**. Éstos valores que hay dentro del *contenedor* pueden *variar*, por ello su nombre **"variable"**. Una variable se compone de:

- Un nombre
- Un valor

Dentro de las convenciones de Python, se estipula que el nombre de una variable comienza con *minúscula*, y si se requiere utilizar más de una palabra, éstas deben estar unidas por un *guión bajo* o *underscore*. Los nombres de las variables **no pueden tener espacios**. Es importante también utilizar nombres que sean representativos del valor que se quiere almacenar. Por ejemplo, si queremos crear una variable de que almacene la cantidad de alumnos de un curso, su nombre podría ser `cant_alumnos_algebra`.

Además, una variable poseerá un **tipo de dato** asociado, según el valor que se le asigne. Los tipos de datos pueden ser, entre otros, `integer`, `string`, `double` o `bool`.

Asignando un valor a una variable

Integers

En Python -y en muchos otros lenguajes de programación- los números enteros se denominan 'enteros' y **Integers**. De tal manera, podemos decir que: La variable `entero` contiene un *Integer* y su valor es 27.

```
entero = 27
```

Como el valor de la variable puede cambiar, a la variable `entero` se le puede asignar un nuevo número sin problemas.

```
entero = 33
```

Strings

Una variable puede almacenar otros tipos de contenido. Por ejemplo, puede almacenar una palabra o una frase.

```
saludo = 'hola'
```

Esto se lee: La variable `saludo` contiene el valor `hola`.

A una palabra o frase se le conoce como *cadena de caracteres* o **String**. Un *String* puede estar formado por:

```
frase = 'Hola Mundo!' # Una o varias palabras
letra = 'x' # Simplemente un caracter.
```

¿Por qué los *Strings* se escriben entre comillas?

Python necesita algún mecanismo para diferenciar si el programador está haciendo referencia a una variable o a un *string*.

```
a = 'Esto es un string'
b = a
print(b)
```

```
Esto es un string
```

En la instrucción `b = a`, Python entiende que nos estamos refiriendo a la variable `a` y no a un caracter 'a'

¿Comillas simples o dobles?

Para trabajar con *Strings* podemos utilizar comillas simples ('') o dobles (""). Lo importante es que la comilla de apertura debe ser igual a la de cierre:

```
comillas_simples = 'Esto es un String'
comillas_dobles = "Esto también es un String"
```

El salto de línea

El salto de línea es un caracter especial que nos permite agregar un salto de línea dentro de un string. Este caracter es **\n**.

```
saltos = "hola\na\ntodos"
print(saltos)
```

```
hola
a
todos
```

El salto de línea requiere ser utilizado dentro de comillas dobles para indicar que el backslash no significa '' literalmente, si no que está escapando la `n` para transformarlo en un salto de línea.

Constantes

Las constantes también se componen de un *nombre* y de un *valor*. En otros lenguajes de programación, a diferencia de las variables, su valor **no** se puede modificar. Se utilizan para establecer valores que son de uso común y que no se debiesen modificar a lo largo del código. Por ejemplo, podríamos utilizar una constante para almacenar el valor del IVA.

Como Python no posee una declaración especial para crear una constante, una buena práctica es crear un archivo `.py` separado que contenga solo las constantes y sus valores asignados. Luego este archivo se puede importar dentro de nuestro código para hacer uso de las constantes declaradas.

Para diferenciar una constante de una variable, por convención el nombre de la constante se escribre *completamente en mayúsculas*.

```
IVA = 0.18
NUMERO_PI = 3.14
```

String y métodos asociados

¿Qué obtendremos al "sumar" dos Strings?

```
a = 'HOLA'
b = ' MUNDO'
print(a + b)
```

```
HOLA MUNDO
```

Al asignar los valores utilizando comillas simples, Python los interpreta como String. Por ende, la instrucción dada por el caracter `+`, para este caso, es de *concatenar*, no de *sumar*. En programación, la acción de 'sumar' dos o más *Strings* se conoce como **concatenación**.

```
a = '7'
b = '3'
print(a + b)
```

```
73
```

Concatenación

Podemos obtener los mismo resultados utilizando concatenación e interpolación, sin embargo, se prefiere la interpolación debido a que es más rápida y presenta una sintaxis más amigable para el desarrollador.

```
nombre = 'Carlos'
apellido = 'Santana'
# Concatenación
print("Mi nombre es " + nombre + " " + apellido)
```

```
Mi nombre es Carlos Santana
```

Interpolación

Otra acción muy importante y ampliamente utilizada al momento de trabajar con Strings es la **interpolación**.

La interpolación es un mecanismo que nos permite introducir una variable (o un dato) dentro un String, sin necesidad de concatenarlo. Para interpolar simplemente tenemos que introducir la variable (o dato) utilizando la siguiente notación:

```
nombre = 'Carlos'
apellido = 'Santana'
# Interpolación
print("Mi nombre es {} {}".format(nombre, apellido))
```

```
Mi nombre es Carlos Santana
```

Count

Otro método que se puede aplicar a las variables de tipo string es **count**, que nos permite contar la cantidad de ocurrencias de un carácter específico dentro de un texto.

En este caso, al hacer:

```
print(count.nombre("a"))
```

obtenemos como resultado “1”, porque la letra “a” aparece 1 vez en el texto “Carlos”.

Len

Entrega la cantidad de letras o el “largo” del texto. Por tanto al ejecutar:

```
print(len(apellido))
```

se obtiene 7, que es la cantidad de letras de “Santana”.

Upper

Aplica letras mayúsculas a todo el texto:

```
print(apellido.upper())
```

Entrada de datos

Con frecuencia nuestro script necesitará interactuar con el usuario, ya sea para seleccionar una opción de un menú o para simplemente ingresar un valor sobre el cuál nuestro script va a operar:

Podemos capturar datos introducidos por un usuario ocupando la instrucción `input`. Se utiliza de la siguiente forma:

Ingresando datos con input

Antes de mostrar el valor, la consola quedará bloqueada hasta que ingresemos una secuencia de caracteres y presionemos la tecla enter.

```
Python 3.6.5 |Anaconda, Inc.| (default, Apr 26 2018, 08:42:37)
[GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE_401/final)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> nombre = input()
Carlos Santana
>>> nombre
'Carlos Santana'
>>>
```

Salida de datos

Cuando se escribe una expresión, el resultado de ésta se muestra inmediatamente en la pantalla. Sin embargo, si el resultado de la expresión es asignado a una variable, para acceder a este valor se requerirá ejecutar la función `print()` a la varianle creada. Otra opción es simplemente llamar a la variable, como en el ejemplo anterior.

Creando un programa con lo aprendido

Con esta simple entrada de datos, ya podemos crear programas sencillos. Por ejemplo, un programa que solicite tu nombre y luego te salude:

```
nombre = input()
print 'Hola {}!'.format(nombre)
```

Operando con variables Int

Los integers corresponden a números enteros. A éstos se les puede aplicar operaciones matemáticas como: suma, resta, división y multiplicación, respetando la precedencia de operaciones.

Respetar la precedencia de operaciones quiere decir que, si tenemos la expresión:

```
10 + 2 * 5
```

Se obtendrá como resultado **20**, ya que primero se resuelve la multiplicación `2 * 5 = 10` y luego se aplica la suma de 10 dando como resultado 20.

Se debe tener en cuenta que no se pueden concatenar directamente a un string, es decir, si tenemos la variable `cadena = "2"` e intentamos aplicar la suma:

```
print(cadena + 2)
```

Python arrojará un error de tipo *must be str, not int*.

Para poder aplicar la suma, primero debemos convertir la cadena a integer con la función `int(cadena)` y luego aplicar la suma. De esta forma, al escribir:

```
print(int(cadena) + 2)
```

sí se obtendrá el resultado de la suma.

Hay veces en la que vamos a querer asignar valores a variables por medio del usuario, y en este caso, la variable tomará el tipo de dato string.

Un ejemplo de esto es mediante el método **input**.

Al crear la variable **a** y asignandole un **input**: `a = input()` vemos que el prompt cambia, ya que esta esperando que el usuario asigne un valor, para el que ingresaremos el valor **6**.

Al solicitar el valor de **a** vemos que éste es "6". Por tanto, si queremos hacer la suma `a + 2` nos generará el mismo error anterior. Para solucionarlo, debemos aplicar el método `int()` a `"a"`.

argv

(En el editor de texto)

Otra manera para asignar valor a una variable es mediante **argv**, esto es cuando queremos ejecutar un script.

Ejemplo:

Crear el archivo `argumentos.py`.

Lo primero, es importar el módulo que nos permite rescatar argumentos desde el terminal:

```
import sys
```

Dentro de este módulo, llamaremos a la propiedad **argv**, la cuál corresponde a la lista de los argumentos introducidos en el terminal.

El primer argumento (cuando se ejecuta un script) es el nombre del mismo, y se encuentra ubicado en la posición **0** de la lista de argumentos.

Todo lo que se escriba después del nombre del script, corresponderá a los arumentos siguientes.

Crearemos una variable **b**, a la cual le asignaremosel argumento en la posición `1` de la lista de argumentos, de la siguiente forma:

```
b = sys.argv[1]
```

En Python, podemos acceder a un elemento de la lita indicándole a ésta el número de la posición donde se encuentra el argumento que queremos acceder. Y estas posiciones comienzan en **0**.

Como la posición **0** era el nombre del script, accederemos a la posición siguiente, es decir **1**, para obtener el valor de esa posición.

Agreguemos `print(b)` en el script para ver ese valor.

(En el terminal)

Para ejecutarlo, se debe realizar de la siguiente forma:

```
python argumentos.py 20
```

(En el editor de texto)

Si ahora agregamos en el print `print(b + 2)` vemos que se produce el mismo error.

Por lo tanto, nuevamente debemos transformar este dato a integer con el método **int()**.

```
b = int(sys.argv[1])
```

(En el terminal)

Repetir comando anterior `python argumento.py 20` y ahora sí podremos visualizar el resultado de la suma.

Sumas y restas

Podemos operar con variables de tipo entero, tal como si estuviéramos operando con el número mismo, por ejemplo:

```
a = 5
b = 2
print(a + 2)
```

```
7
```

```
print(b - a)
```

```
-3
```

Multiplicaciones y divisiones

Para realizar operaciones de multiplicación utilizaremos `*` y para operaciones de división utilizaremos `/`

```
a = 10
b = 2
print(a * b)
```

```
20
```

```
print(a / b)
```

```
5.0
```

Sobreescribiendo variables

Podemos sobreescribir el valor de una variable realizando una nueva asignación:

```
a = 'HOLA!'
print(a)
```

```
HOLA!
```

```
a = 100
print(a)
```

```
100
```

Modificando una variable existente

También podemos modificar el valor de una variable operando sobre ella misma. Este tipo de operación es muy utilizada:

```
a = 2
a = a + 1
print(a)
```

```
3
```

Concatenando números por error

Otro detalle interesante es que `input` siempre entrega un *String*. Por lo tanto, si aplicamos operaciones de suma (+) a números ingresados por teclado, estos serán concatenados.

```
num1 = input() #4
num2 = input() #6
num1 + num2
```

```
'46'
```

Transformando los datos

Este comportamiento se puede modificar aplicando **transformaciones a los tipos de datos**.

```
num1 = int(input()) #4
num2 = int(input()) #6
num1 + num2
```

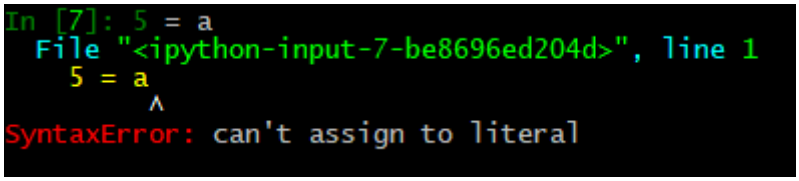
```
10
```

Errores sintácticos vs errores semánticos

Los errores podemos clasificarlos en dos tipos: sintácticos y semánticos.

- Los errores sintácticos son muy similares a los errores gramaticales, donde escribimos algo que contraviene las reglas de como debe ser escrito.

Por ejemplo, el valor de una variable se asigna de izquierda a derecha (`a = 5`), mientras que lo contrario (`5 = a`) sería un error sintáctico. En este caso, Python nos informará del error.



- Los errores semánticos en cambio, son muy distintos. Cuando se tiene un error semántico, el programa funcionará de forma normal, pero el resultado será distinto al esperado. Los errores semánticos son errores de implementación o de diseño, y, como Python no nos ha indicado el error, a menudo son más difíciles de detectar.

Floats, Booleanos y otros métodos

Otro tipo de dato es el “float”, que corresponde a los números decimales.

```
decimal = 10.5
```

En Python, se debe tener en cuenta que al aplicar una división entre 2 números enteros, el resultado será un float, aunque el resto de la división sea 0.

```
a = 10
b = 5
print(a/b) #2.0
print(int(a/b)) #2
```

El otro tipo de dato que se utiliza con frecuencia, y que aplicaremos más adelante cuando veamos control de flujo, son los booleanos. Una variable booleana solo tiene 2 valores posibles, `True` o `False` .

Se puede asignar directamente asignando `True` o `False` a una variable (primera letra con mayúscula, sin comillas), o asignando una expresión que de como resultado True o False (se verá en detalle cuando se vea control de flujo):

```
diestro = True
animal = False
```

Importancia del tipo de dato

Para terminar el capítulo, enfatizar que saber el tipo de dato con el que estamos trabajando es sumamente importante. Si no se respetan de cada tipo de dato, nos enfrentaremos a problemas para realizar lo que queremos hacer, como era el caso de hacer una suma a un valor asignado por el usuario.

Por ello, resulta útil conocer la función `type()`. Esta función nos entregará el tipo de dato con el que estamos trabajando.

```
a = "2"
type(a) # str
```

Si se necesita manejar la variable como un int, esto se soluciona utilizando la función `int()`. De la misma forma, también se puede aplicar la función `str()` para convertir una variable a tipo string o `float()` para convertir una variable a tipo float.

Resumen

En este capítulo se habló de varios elementos claves para construir un programa en Python. Estos son:

- **Variables**: Contenedores de datos.
- **Operadores**: Nos permiten sumar, restar y hasta concatenar datos.
- **Entrada y salida de datos**: Con `input` y `print`.
- **Tipos de datos**: Concepto importante. Por ahora se explicaron `integer`, `string`, `float` y `bool` y cómo manejarlos en términos generales.
- **Interpolación**: Forma de utilizar una variable dentro de un string.

Introducción a objetos

Objetivos

- Definir objeto y método.
- Conocer la documentación oficial de Python.
- Interpretar documentación oficial de Python.
- Conocer la sintaxis de métodos y argumentos.

Objetos y métodos

Hasta el momento hemos trabajado con tipos de dato **Integer** y **String**. En Python, y en los lenguajes de programación "orientados a objetos", estos tipos de dato corresponden a **Clases** .

Un *objeto* es la unidad más básica que puede tener un componente de un programa, y corresponde a una *instancia de una clase*. Por tanto, una variable de tipo *String*, será un objeto, ya que es una instancia de la clase *String*. Los objetos tienen asociadas todas las propiedades y métodos que posee la clase a la cual están instanciando.

Los *métodos* corresponden a operaciones que realizamos sobre los objetos, considerando que **tienen ciertas restricciones**. Por ejemplo, el método `+` nos permite sumar dos enteros, o concatenar dos strings, pero no nos permite sumar un string con un entero.

Ya hemos ocupado varios métodos.

- `input()`
- `format()`
- operaciones como `+`

A través de la documentación podemos conocer otros métodos asociados a un objeto.

Método nativo vs método asociado a un tipo de objeto

Python presenta una serie de operaciones que son **nativas** (o *Built-in methods*): se comportan de manera diferente dependiendo del tipo de dato que se ingresa.

Supongamos que deseamos contar cuántas letras existen en la cadena `'Lorem ipsum dolor sit amet'`. Para ello, podemos utilizar la función nativa de Python `len`.

```
cadena = 'Lorem ipsum dolor sit amet'
len(cadena)
```

```
26
```

Recordar que cada objeto creado vendrá con una serie de métodos asociados. Si deseamos ocupar alguno de estos, debemos implementar la sintáxis: `objeto.metodo()`.

Por ejemplo, si deseamos transformar todas las letras de la cadena a mayúsculas, podemos ocupar el método `upper` asociad a los objetos de tipo `string`.

```
cadena.upper()
```

```
'LOREM IPSUM DOLOR SIT AMET'
```

Parámetros y retorno

Métodos con parámetros

En algunos casos, los métodos pueden recibir **parámetros de entrada**, conocidos como **argumentos** o **parámetros**. Los parámetros pueden ser variables, expresiones, o incluso otros métodos.

Por ejemplo, el método `count` del objeto String recibe un *substring* (un conjunto de caracteres mas corto) como argumento, para poder contar cuántas veces está contenido ese substring en el String principal.

Imaginemos que queremos saber cuantas letras 'i' tiene la frase `Lorem ipsum dolor sit amet`. Para ello, se debe llamar al método `count` del objeto `cadena`, y especificarle como parámetro la letra `i`:

```
cadena.count('i')
```

```
2
```

El retorno de un método

Además de los **parámetros**, los métodos tienen como característica que nos pueden devolver o **retornar** un resultado. Lo que devuelve un método se conoce como el **retorno del método**.

De tal manera, podemos decir que el método `count` del objeto String:

- Recibe como **parámetro**, al menos, un substring.
- **Retorna** un entero correspondiente a la cantidad de veces que está contenido el substring en el objeto String.

```
'paralelepipedo'.count('p')
```

```
3
```

También puede ocurrir que el método utilizado no tenga retorno, y que se aplique el resultado automátocamente a la variable que se esta operando.

Por ejemplo, cuando más adelante estudiemos **listas** veremos que `append` es un método utilizado para agregar elementos a la lista.

```
lista = []
lista.append("perro")
```

En el terminal no aparece ningún retorno. Pero si solicitamos el valor de la lista `lista`

Veremos que se agregó en ella el elemento `perro`.

¿Y los paréntesis?

Cuando utilizamos un método, todo argumento **debe pasarse entre paréntesis**.

Para saber si el método que queremos utilizar recibe parámetros, y si tiene retorno y de qué tipo, debemos revisar la documentación oficial.

Documentación

Para conocer qué operaciones son válidas sobre un objeto, debemos leer su documentación oficial.

Consultar la documentación oficial es un hábito que **debemos** adoptar al momento de aprender un nuevo lenguaje de programación.

¿Cómo se lee la documentación?

Utilizaremos la documentación de Python. En esta página, podemos ver la documentación de todos los objetos incluidos en Python.

Comenzaremos revisando uno de los objetos que más se ha utilizado en la lectura: Los integers.

Leyendo la documentación

La manera más fácil de encontrar la documentación asociada a un elemento en específico es utilizando el buscador integrado a la página de documentación, el cual se encuentra en la esquina superior derecha.

Numeric Types — `int`, `float`, `complex`

There are three distinct numeric types: *integers*, *floating point numbers*, and *complex numbers*. In addition, Booleans are a subtype of integers. Integers have unlimited precision. Floating point numbers are usually implemented using `double` in C; information about the precision and internal representation of floating point numbers for the machine on which your program is running is available in `sys.float_info`. Complex numbers have a real and imaginary part, which are each a floating point number. To extract these parts from a complex number *z*, use `z.real` and `z.imag`. (The standard library includes additional numeric types, `fractions` that hold rationals, and `decimal` that hold floating-point numbers with user-definable precision.)

Numbers are created by numeric literals or as the result of built-in functions and operators. Unadorned integer literals (including hex, octal and binary numbers) yield integers. Numeric literals containing a decimal point or an exponent sign yield floating point numbers. Appending `'j'` or `'J'` to a numeric literal yields an imaginary number (a complex number with a zero real part) which you can add to an integer or float to get a complex number with real and imaginary parts.

Python fully supports mixed arithmetic: when a binary arithmetic operator has operands of different numeric types, the operand with the "narrower" type is widened to that of the other, where integer is narrower than floating point, which is narrower than complex. Comparisons between numbers of mixed type use the same rule. [2] The constructors `int()`, `float()`, and `complex()` can be used to produce numbers of a specific type.

All numeric types (except complex) support the following operations, sorted by ascending priority (all numeric operations have a higher priority than comparison operations):

Operation	Result	Notes	Full documentation
`x + y`	sum of *x* and *y*		
`x - y`	difference of *x* and *y*		
`x * y`	product of *x* and *y*		
`x / y`	quotient of *x* and *y*		
`x // y`	floored quotient of *x* and *y*	(1)	
`x % y`	remainder of `x / y`	(2)	
`-x`	*x* negated		
`+x`	*x* unchanged		
`abs(x)`	absolute value or magnitude of *x*		`abs()`
`int(x)`	*x* converted to integer	(3)(6)	`int()`
`float(x)`	*x* converted to floating point	(4)(6)	`float()`
`complex(re, im)`	a complex number with real part *re*,	(6)	`complex()`

Al consultar por `integer`, Python nos redireccionará a la sección de tipos de datos numéricos. En esta página, se encuentran los detalles de cuál es la definición de un dato de tipo numérico.

Si hacemos click en `int`, la documentación hará referencia a la clase que permite generar un número entero.

class **int**([*x*])
class **int**(*x*, *base=10*)

Return an integer object constructed from a number or string *x*, or return `0` if no arguments are given. If *x* defines `__int__()`, `int(x)` returns `x.__int__()`. If *x* defines `__trunc__()`, it returns `x.__trunc__()`. For floating point numbers, this truncates towards zero.

If *x* is not a number or if *base* is given, then *x* must be a string, `bytes`, or `bytearray` instance representing an integer literal in radix *base*. Optionally, the literal can be preceded by `+` or `-` (with no space in between) and surrounded by whitespace. A base-n literal consists of the digits 0 to n-1, with `a` to `z` (or `A` to `Z`) having values 10 to 35. The default *base* is 10. The allowed values are 0 and 2-36. Base-2, -8, and -16 literals can be optionally prefixed with `0b`/`0B`, `0o`/`0O`, or `0x`/`0X`, as with integer literals in code. Base 0 means to interpret exactly as a code literal, so that the actual base is 2, 8, 10, or 16, and so that `int('010', 0)` is not legal, while `int('010')` is, as well as `int('010', 8)`.

The integer type is described in Numeric Types — int, float, complex.

Changed in version 3.4: If *base* is not an instance of `int` and the *base* object has a `base.__index__` method, that method is called to obtain an integer for the base. Previous versions used `base.__int__` instead of `base.__index__`.

Changed in version 3.6: Grouping digits with underscores as in code literals is allowed.

Changed in version 3.7: x is now a positional-only parameter.

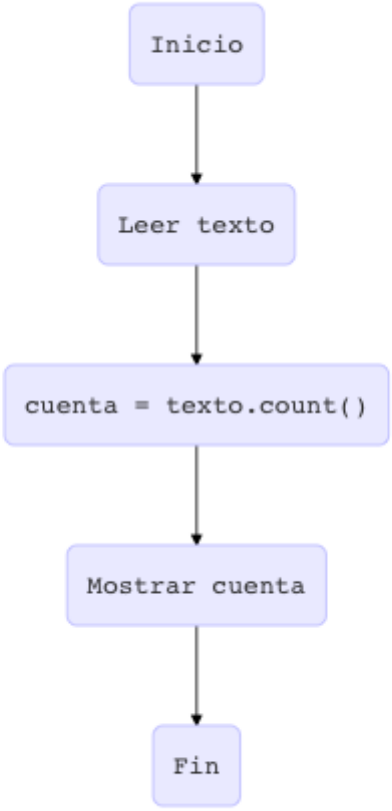
```
'paralelepipedo'.count('p')
```

```
3
```

Ejercitando lo aprendido

Lo aprendido lo podemos utilizar para crear un pequeño programa donde el usuario introduce un valor y mostramos la cantidad de letras:

Algoritmo



Código

```
content = input()
letters = len(content)
print("El contenido {} tiene {} letras".format(content, letters))
```

```
El contenido Lorem ipsum dolor sit amet tiene 26 letras
```

Resumen

En este capítulo aprendimos sobre objetos, métodos y como consultar la documentación.

Objetos

- Los tipos de datos son realmente objetos (Esto es en Python y solo aplica a algunos lenguajes).
- Los objetos tienen métodos que nos permiten realizar todo tipo de acciones.

Métodos

- Ocupar un método, llamar, o invocar un método son sinónimos.
- Solo podemos llamar métodos que ya han sido defindos (podemos definirlos nosotros o buscarlos en la documentación).
- Los métodos pueden recibir valores y esto lo hace mas flexibles.

Documentación

Al consultar en la documentación oficial sobre un método debemos fijarnos en:

- Nombre del método.
- Parámetros que recibe: opcionales y obligatorios.
- Información que devuelve.

Tipos de objeto

Objetivos

- Identificar un entero en Python.
- Identificar un flotante en Python.
- Identificar un string en Python.
- Identificar un boolean en Python.
- Diferenciar el comportamiento de método `+` en Integer vs String.
- Transformar Strings a Integers.

Recapitulación de objetos

En Python existen distintos tipos de datos. Ya sabemos que estos tipos de datos son objetos, por ende, lo correcto es llamarlos **tipos de objeto**.

Cada uno de estos objetos se comporta de forma distinta.

Dentro de los tipos de datos más utilizados podemos destacar:

- `Integer`: Corresponde a un número entero.
- `String`: Corresponde a un caracter o una cadena de caracteres.
- `Float`: Corresponde a un número decimal.
- `Time`: Corresponde a una fecha y hora.
- `Boolean`: Corresponde a `True` o `False`. Son el resultado de una evaluación.
- `None`: corresponde a la ausencia de un valor.

Profundizaremos más en estos tipos de objeto a medida que avancemos.

¿Cómo saber de qué tipo es un objeto?

Podemos saber el tipo de dato utilizando el método `type(x)`

Por ejemplo, si dentro de la consola de Python escribimos `type(2)`, obtendremos como resultado `int`, o si probamos con `type('hola')` obtendremos como resultado `str`.

```
suma = 5 + 2
```

```
type(suma)
```

```
int
```

```
otra_suma = 2.3 + 0.1
```

```
type(otra_suma)
```

```
float
```

¿Por qué es importante el tipo de objeto?

Es importante porque existen distintas reglas para operar entre estos distintos tipos de objeto. Estas reglas las conoceremos consultando la documentación oficial.

Por ejemplo: Al sumar dos números obtenemos el resultado de la suma, pero al sumar dos palabras obtenemos la concatenación de estas.

En algunas situaciones, cuando faltemos a estas reglas, las operaciones no serán válidas.

Concatenando strings

Observemos el siguiente ejemplo: el método `+` del objeto String recibe como parámetro otro String a concatenar.

```
"Hola " + "Mundo"
```

```
'Hola Mundo'
```

Concatenando un string con otro tipo de dato

¿Qué sucede si intentamos concatenar un Integer a un String?

```
"HOLA" + 2
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-36-8946f4d722d7> in <module>()
----> 1 "HOLA" + 2
```

```
TypeError: must be str, not int
```

Para esto debemos transformar el tipo de dato:

```
"Hola" + str(2)
```

```
Hola 2
```

¿Y si queremos sumar dos números ingresados por teclado?

```
numero_uno = input() #10
numero_dos = input() #20
print(numero_uno + numero_dos)
```

```
1020
```

Como el resultado de un input() es de tipo String, al usar el operador `+` se produce una concatenación en lugar de una suma. Para solucionar este problema y, dependiendo de nuestro objetivo, podemos aplicar transformaciones a los tipos de objeto.

Trasformando tipos de objeto

Existen distintos métodos que nos permiten transformar un objeto de un tipo a otro. Dentro de estos métodos podemos destacar:

- El método `int(x)` nos permite convertir un objeto en Integer.
- El método `str(x)` nos permite convertir un objeto en String.

```
int(2018)
```

```
2018
```

```
str(2018)
```

```
'2018'
```

¡Ahora sí podemos sumar dos números ingresados por teclado!

```
numero_uno = int(input()) #10
numero_dos = int(input()) #20
print(numero_uno + numero_dos)
```

```
30
```

Operaciones aritméticas

En este capítulo aprenderemos a hacer aplicaciones del tipo de calculadora, donde los usuarios ingresan valores y le entregamos resultados.

Objetivos

- Construir aplicaciones para calcular una función matemática.

Para lograr esto tenemos que:

- Conocer operadores aritméticos.
- Conocer la precedencia de los operadores aritméticos.
- Utilizar parentesis para priorizar operaciones.
- Operar sobre números enteros y flotantes.

Introducción a operaciones y operadores

En Python existen herramientas que nos permiten, entre otras cosas: sumar, restar, asignar valores, comparar, etc. Todo esto -y mucho más- es posible gracias a los operadores.

Operadores aritméticos

Los operadores aritméticos nos permiten realizar operaciones matemáticas sobre números.

Operador	Nombre	Ejemplo	Resultado
+	Suma	2 + 3	5
-	Resta	2 - 3	-1
*	Multiplicación	3 * 4	12
/	División	12 / 4	3
**	Potencia	2 ** 4	16

Operaciones con variables

El proceso es exactamente igual si guardamos los valores en variables:

```
a = 2
b = 3
print(a + b)
```

Creando una calculadora

Esto nos permite que el usuario ingrese los valores, transformarlos a números y luego operar.

```
a = int(input()) # 10
b = int(input()) # 22
print("a + b es: {}".format(a + b))
print("a * b es: {}".format(a * b))
```

```
10
22
a + b es: 32
a * b es: 220
```

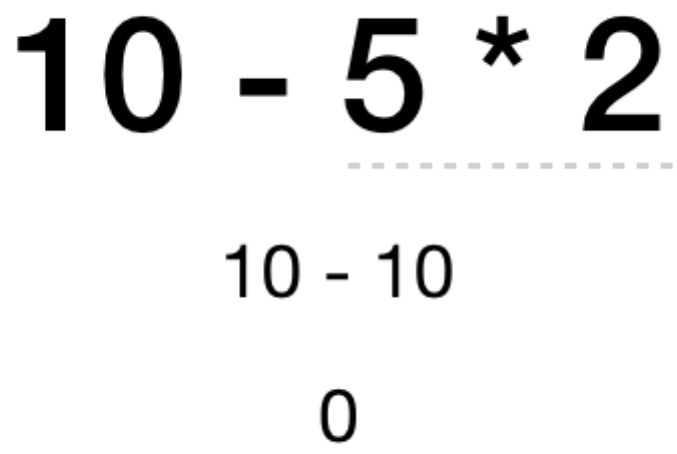
Precedencia de operadores

Un concepto muy importante que debemos conocer es el de precedencia, dicho de otro modo, **saber en qué orden se realiza un grupo de operaciones**.

Ejemplo de precedencia

Por ejemplo, en la siguiente instrucción ¿cuál es el resultado?

```
10 - 5 * 2
```




```
10 - 5 * 2 # 0
```

```
0
```

Orden de las operaciones

Veamos una tabla simplificada de precedencia. Esta tabla está ordenada de mayor a menor prioridad, esto quiere decir que la operación de exponenciación tiene mayor precedencia que la suma.

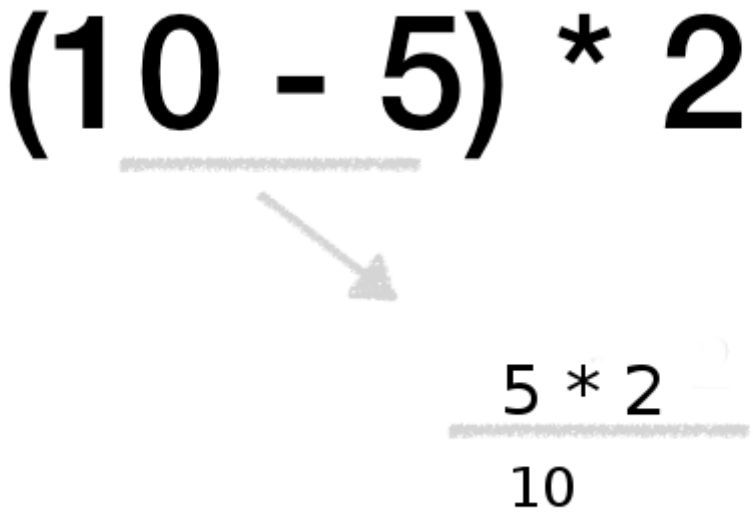
Operador	Nombre
**	Exponenciación (potencia)
* , /, %	Multiplicación, división y módulo
+ , -	Suma y resta

Cuando dos operaciones tienen el mismo nivel de prioridad entonces se resuelven de izquierda a derecha.

Operaciones y paréntesis

Al igual en matemáticas, los paréntesis cambian el orden en que preceden las operaciones dando prioridad a las operaciones que estén dentro de los paréntesis.

```
(10 - 5) * 2
```



¡Los paréntesis importan!

Operaciones con números enteros y decimales

Si dividimos números enteros nos encontraremos con una sorpresa.

```
5 / 3 # => 1
```

Esto es muy común en todos los lenguajes de programación. Para obtener la respuesta que esperamos necesitamos ocupar otro tipo de dato, el `float`.

Float

En el capítulo anterior, mencionamos que existía el tipo de dato asociado a los números **decimales** se llamaba float.

```
type(3.1) # float
```

Enteros y floats

La división entre entero y float, o float y entero, da como resultado un float.

```
5.0 / 3.0 # 1.6666666666666667
5 / 3.0 # 1.6666666666666667
```

La división entre flotas también es un float.

Los floats son muy importantes dentro de la programación, y tienen propiedades curiosas. Una de las más importantes es que solo son una representación aproximada de los números.

```
(10 / 3.0)
 3.3333333333333335
```

También podemos transformar a float ocupando el método `float()`, esto será especialmente útil cuando estemos trabajando con variables que contengan enteros.

```
a = int(input()) # 1
b = int(input()) # 2
print(a / float(b)) # 0.5
```

O, pudimos haber transformado en un inicio la variable a float.

Fahrenheit

Fahrenheit es una escala de temperatura. Podemos transformar una temperatura de la escala Fahrenheit a Celsius, ocupando la siguiente ecuación.

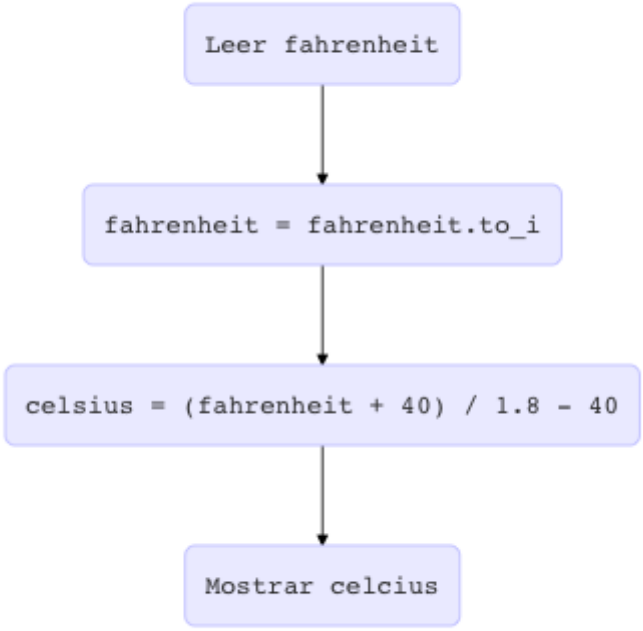
$$((f + 40) \div 1.8) - 40 = c$$

Antes de desarrollar el algoritmo revisar los siguientes puntos:

- ¿Cuántos valores tiene que ingresar el usuario, 1 ó 2?
- ¿Existe algún punto donde tengamos que tener cuidado con la precedencia de operadores?

Luego dibuja el diagrama de flujo y finalmente escribe el código.

Algoritmo




```
farenheit = int(input())
celsius = (farenheit + 40) / 1.8 - 40
print("la temperatura es de {} celsius".format(celsius))
```

Resumen

- Precedencia: El orden en que deben ser resuelta las operaciones.
- División entre enteros: El resultado es flotante `6 / 4 #=> 1.5`
- División entre flotante y entero: El resultado es flotante `6 / 4.0 # => 1.5`
- Paréntesis(): Al igual que en matemáticas nos ayudan a darle prioridad a una operación.

Manejo básico de flujo

Objetivos

En este capítulo aprenderemos a resolver algoritmos donde hayan una o dos opciones a resolver.

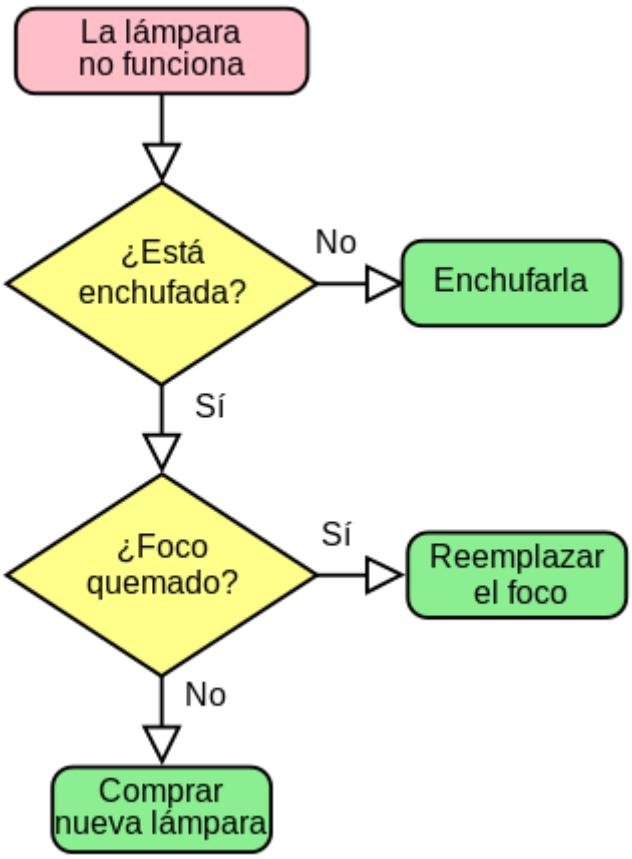
- Crear algoritmos que tomen decisiones en base al valor de una variable.

Para esto tenemos que:

- Conocer operadores de comparación.
- Diferenciar asignación de comparación.

Manejo de flujo

Al inicio de la unidad hablamos mostramos el siguiente diagrama de flujo.



El rombo corresponde a una **evaluación condicional**. En este punto se toma una decisión y el programa sigue en función de esa decisión. En este capítulo aprenderemos a escribir programas que implementen estas decisiones.

La instrucción IF

Python -y muchos otros lenguajes de programación- tiene instrucciones para implementar condiciones. Una de la más utilizada es la instrucción `if`. La sintaxis es la siguiente:

```
if condition:
    # código que se ejecutará SÓLO si se cumple la condición
```

Lo anterior se lee como: "**Si** se cumple la condición, **entonces** ejecuta el código". Dentro de la sintaxis, se debe escribir la condición a evaluar a continuación de la palabra "if", y al final de ésta agregar dos puntos `:`. El código que se ejecute si se cumple la condición, se debe escribir en una nueva línea después de `:`, identado por **4 espacios**.

Ejemplo de if

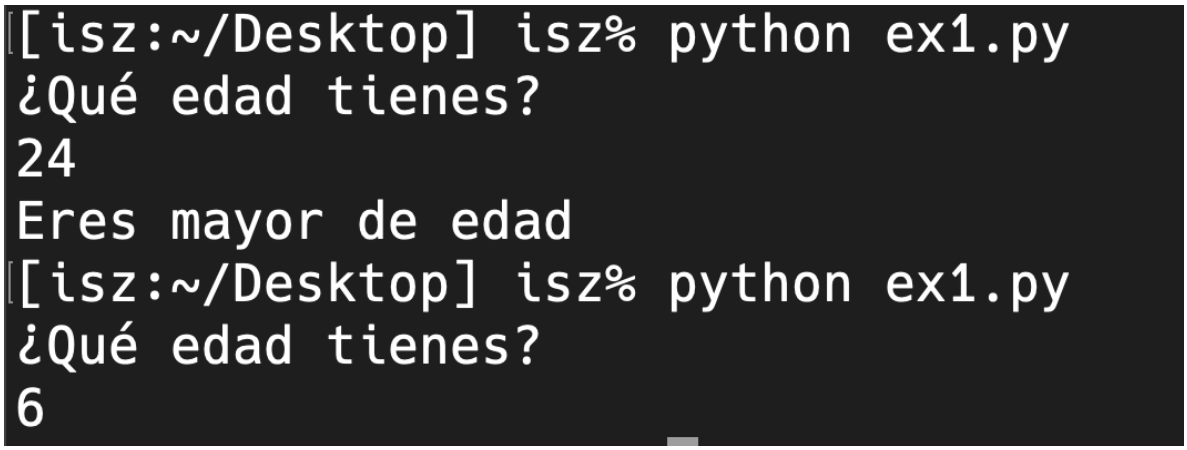
Analicemos el siguiente ejemplo: En muchos países de Latinoamérica, la mayoría de edad se alcanza a la edad de 18 años. Crearemos un programa que pregunte la edad al usuario. Si la edad es mayor o igual a 18 entonces, le diremos que es mayor de edad.

```
edad = int(input("¿Qué edad tienes?"))

if edad >= 18:
    print("Eres mayor de edad")
```

```
¿Qué edad tienes?33
Eres mayor de edad
```

Probando el programa

Si ejecutamos el programa e introducimos un valor mayor o igual a 18, veremos el mensaje "Eres mayor de edad". En caso contrario no veremos ningún mensaje.



En este ejercicio, se realizó una comparación utilizando el operador `>=`, pero existen varios operadores que permiten comparar. Los estudiaremos a continuación.

Operadores de comparación

Los operadores de comparación son aquellos que comparan dos valores y obtienen como resultado `True` (verdadero) o `False` (falso). A este tipo de objeto, resultado de una comparación, se le conoce como **booleano**.

Operadores de comparación en integers

Operador	Nombre	Ejemplo	Resultado
==	Igual a	2 == 2	true
!=	Distinto a	2 != 2	false
>	Mayor a	3 > 4	false
>=	Mayor o igual a	3 >= 3	true
<	Menor a	4 < 3	false
<=	Menor o igual a	3 <= 4	true

Realicemos una prueba en Python donde el usuario ingrese los valores y veamos si el primero es mayor que el segundo.

```
a = int(input()) # 3
b = int(input()) # 5
print(a > b)
```

```
False
```

Operadores de comparación en strings

Aunque en la tabla solo hayamos mostrado números, podemos comparar dos objetos de tipo string utilizando un operador de comparación:

```
'texto1' == 'texto2' # False
```

Asignación vs Comparación

No debemos confundir asignación del valor de una variable con el de comparación.

```
# Asignación
nombre = 'Carlos Santana'
print(nombre) # 'Carlos Santana'

# Comparación
nombre == 'Carlos Santana' # True
```

¿Puede un texto ser mayor que otro?

Otros operadores también pueden ser aplicados a los strings, por ejemplo el `>`

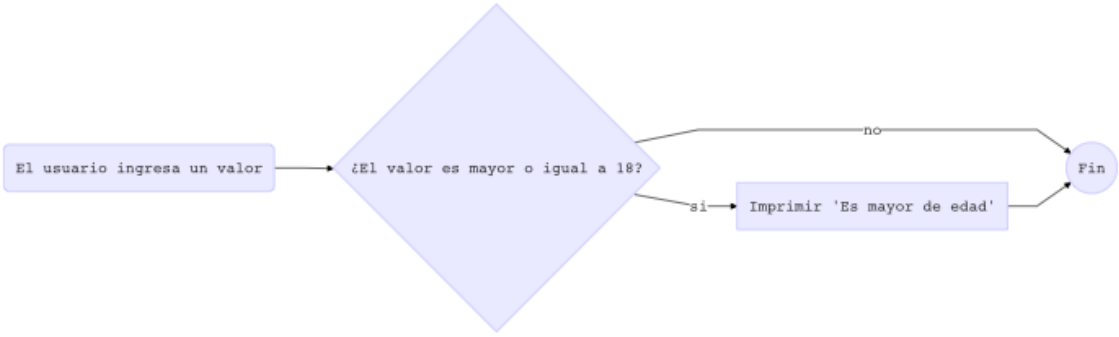
```
'a' > 'b' # False
```

En este caso, la comparación es por orden alfabético. Las letras que aparecen primero en el alfabeto son menores que las que aparecen después. Para entenderlo de forma sencilla, cuando comparamos dos palabras, es menor aquella que aparecería antes en un diccionario.

En el ejemplo vemos que la palabra `recycle` es menor que la plabra `red` porque letra **c** es menor que **d**.

Ahora que ya conocemos los operadores de comparación volvamos a nuestro código anterior y analicemos el diagrama de flujo.

```
edad = int(input("¿Qué edad tienes?"))

if edad >= 18:
    print("Eres mayor de edad")
```



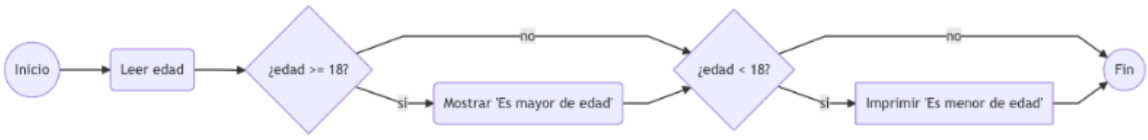
Importante

- Todo lo que está dentro del `if`, sucederá solo si se cumple la condición.
- El código dentro del `if` **debe** estar indentado por 4 espacios. De esta forma, se logra reconocer visualmente de forma sencilla dónde empieza y termina el código que se ejecuta dentro de la condición. Además del reconocimiento visual, la identación es la forma en que Python detecta el código a ejecutar.

¿Qué hacer si la condición no se cumple?

¿Cómo podemos modificar nuestro programa para que muestre un mensaje cuando el usuario sea menor edad y otro mensaje cuanto el usuario sea mayor de edad?

Una muy **buena práctica** es la de realizar un diagrama de flujo antes de comenzar a programar. Esto ayuda a abstraerse del código y pensar en los pasos críticos.

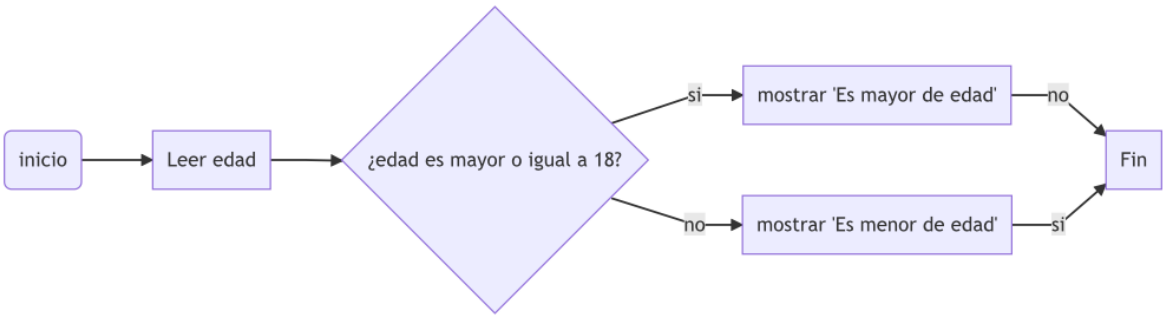


Transcribamos nuestro diagrama de flujo a código Python:

```
edad = int(input("¿Qué edad tienes?"))

if edad >= 18:
    print("Eres mayor de edad")

if edad < 18:
    print("Eres menor de edad")
```

Una pregunta interesante que nuestro diagrama refleja de manera implícita es: ¿Puede el usuario ser mayor y menor de edad a la vez?

La respuesta es evidentemente **no**. Este tipo de situaciones se puede modelar mejor de la siguiente forma.



Para implementarlo tenemos que introducir una nueva instrucción: `else`.

La instrucción ELSE

La instrucción `else` se utiliza junto con `if` para seguir el flujo de código **en cualquier caso donde no se cumpla la condición**.

```
if condition:
    # código que se ejecutará SÓLO SI se cumple la condición
else:
    # código que se ejecutará si NO se cumple la condición
```

Lo anterior se lee como: "**Si** se cumple la condición, **entonces** ejecuta el bloque de código, **sino** ejecuta el otro bloque de código". En cuanto a la sintaxis, la palabra `else` asociada al `if` debe estar a su mismo nivel de identación, y se debe agregar `:` al final. El código a ejecutar se escribe en la línea siguiente, identado por 4 espacios.

```
edad = int(input("¿Qué edad tienes?"))

if edad >= 18:
    print("Eres mayor de edad")
else:
    print("Eres menor de edad")
```

Ejercicio práctico

Crear un programa donde el usuario deba ingresar dos números. El programa debe imprimir el mayor de ambos números.

```
valor1 = int(input("Ingrese valor 1"))
valor2 = int(input("Ingrese valor 2"))

if valor1 >= valor2:
    print("valor1 {} es mayor".format(valor1))
else:
    print("valor2 {} es mayor".format(valor2))
```

Ejercicios propuestos

- El usuario ingresa debe ingresar un password en la plataforma. Si el password tiene menos de 6 letras, se debe mostrar un aviso.
- El usuario debe ingresar un password. Si el password es 12345, se debe informar que el password es incorrecto.

Resumen

- En este capítulo aprendimos a utilizar las instrucciones `if` y `else`, las cuales nos permiten manejar el flujo de un programa.
- Existen operadores de comparación que permiten comparar si una expresión es verdaera o falsa.
- Los operadores de comparación se pueden utilizar dentro de la sintaxis de la instrucción `if`.
- Se debe tener cuidado de no confundir la asignación con la comparación. Es decir, `=` es distinto que `==`.
- Todo lo que está dentro del if, sucederá solo si se cumple la condición.
- El código dentro del `if` debe estar indentado por 4 espacios para poder reconocer de forma sencilla y visualmente dentro del código donde empieza, termina, y que código se ejecuta dentro de la condición.

Profundizando en flujo

Objetivos

En este capítulo aprenderemos a resolver problemas donde las posibles alternativas a una decisión sean mas de dos.

- Analizar situaciones más allá de la opción y su caso contrario.
- Utilizar la instrucción 'elif' en Python

¿Qué hacer si se quiere analizar más de 2 casos?

Abordemos el problema desde un ejemplo

En el capítulo anterior creamos un algoritmo donde de dos números debíamos determinar cuál es el mayor. Utilizaremos el mismo ejemplo, agregando qué se debe mostrar cuando el caso sea que ambos números son iguales.

Antes:

- caso1: A >= B => Decimos que A es mayor
- caso2: A < B => Decimos que A es menor

Ahora:

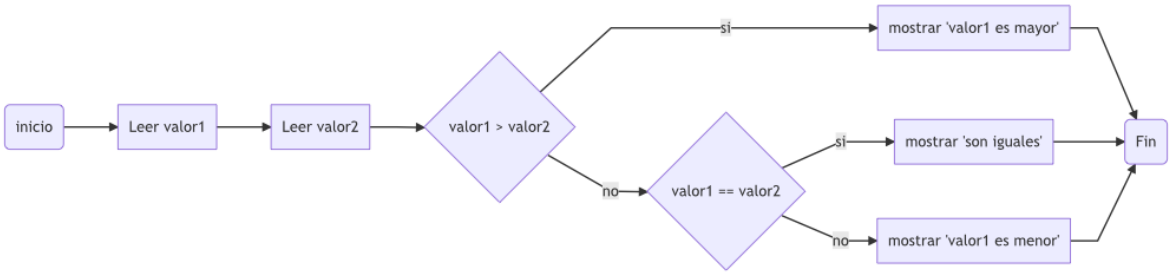
- caso1: A > B => Decimos que A es mayor
- caso2: A == B => Decimos que A y B son iguales
- caso3: A < B => Decimos que A es menor que B

El mayor de dos números: Escenario donde se ingresan dos números iguales

¿Cómo hacemos para mostrar que ambos números son iguales? Ante cualquier problema de este tipo, lo mejor es ir paso a paso y analizar el problema.

- Si el primero es mayor -> Mostramos mensaje.
- Si NO es menor -> Tenemos dos opciones:
    - El primer valor es menor
    - Ambos son iguales

Con esta descomposición del problema tenemos toda la información necesaria para construir nuestro diagrama:



Con el diagrama de flujo realizado, la transcripción a código Python se vuelve sencilla:

```
valor1 = int(input("Ingrese valor 1"))
valor2 = int(input("Ingrese valor 2"))

if valor1 > valor2:
    print("valor1 {} es mayor".format(valor1))
else:
    if valor1 == valor2:
        print("Ambos valores son iguales")
    else:
        print("valor2 {} es mayor".format(valor2))
```

Un código siempre debe ser probado y debemos probar todos los casos, probemos con los valores:

- 2, 1
- 2, 2
- 2, 3

Existe otra solución sin tener que agregar un if dentro del otro.

La instrucción ELIF

La instrucción `elif` nos permite capturar el flujo y volver a realizar una evaluación condicional cuando no se cumplió una evaluación previa. Con esta instrucción podemos reescribir el algoritmo del ejemplo anterior:

```
valor1 = int(input("Ingrese valor 1"))
valor2 = int(input("Ingrese valor 2"))

if valor1 > valor2:
    print("valor1 {} es mayor".format(valor1))
elif valor1 == valor2:
    print("Ambos valores son iguales")
else:
    print("valor2 {} es mayor".format(valor2))
```

La instrucción `elif` nos permite continuar realizando tantas evaluaciones condicionales como necesitemos. Finalmente podemos utilizar la instrucción `else` para cuando no es cumpla ninguna condición anterior.

Cabe destacar que dos instrucciones `if` **no es lo mismo** que utilizar `elif`. Para su sintaxis, se debe respetar que debe ir **obligatoriamente** asociado a un `if`, que se encuentre previamente declarado, e identado a su mismo nivel. De existir una instrucción `else`, debe declararse antes de ésta.

Clasificación según rangos

Es normal estar en situaciones donde nos pidan clasificar algún elemento según un rango.

Por ejemplo, supongamos que tenemos una palabra y queremos clasificarla en corta, mediana o larga:

- 4 letras o menos será corta.
- 5 a 10 letras será mediana.
- Más de 10 letras será larga.

```
palabra = input("Ingrese una palabra")
largo = len(palabra)

if largo <= 4:
    print("Pequeña")
elif largo < 10:
    print("Mediana")
else:
    print("Larga")
```

Ejercicio: Modifica el código anterior para poder distinguir palabras muy largas, cuando tengan 15 o más caracteres.

```
# solución
palabra = input("Ingrese una palabra")
largo = len(palabra)

if largo <= 4:
    print("Pequeña")
elif largo < 10:
    print("Mediana")
elif largo < 14:
    print("Larga")
else:
    print("Muy larga")
```

```
Larga
```

Diferencia entre “if” al mismo nivel y “elif” asociados a un “if”

¿Cuál es la diferencia entre este código primer y segundo código?

1)

```
palabra = input("Ingrese una palabra")
largo = len(palabra)

if largo <= 4:
    print("Pequeña")
elif largo < 10:
    print("Mediana")
else:
    print("Larga")
```

2)

```
palabra = input("Ingrese una palabra")
largo = len(palabra)

if largo <= 4:
    print("Pequeña")

if largo < 10:
    print("Mediana")
else:
    print("Larga")
```

Probemos con la palabra 'hola'. En el primer código obtendremos 'Pequeña', pero en el segundo 'Pequeña' y 'Mediana'.

Una vez que una condición `if` o `elif` se evalúa como verdadera, el flujo entra en esa dirección, y no evalúa el resto de las condiciones. Con dos if se evalúa el primero y luego el segundo.

Condiciones de borde

Objetivos

- Conocer la importancia de condiciones de borde.
- Analizar condiciones de borde.
- Evaluar comportamiento de operadores >, >=, < y <= en Python.

Motivación

En este capítulo aprenderemos a analizar código y buscar errores semánticos, o sea, cuando el código está bien escrito pero no hace exactamente lo que queremos que haga. Este tipo de errores es muy típico y es el que más tiempo consume.

La falta de un símbolo, un `;` a algo similar, son errores sintácticos. Estos errores son bastante fáciles de detectar, ya que el al ejecutar el código se informará de error. En cambio, cuando un programa se ejecuta sin errores pero no funciona como debería, puede llegar a ser bastante mas complejo, porque no hay ningún mensaje de error asociado a una línea de código en específico que falle.

En este capítulo aprenderemos un tipo de análisis que nos ayuda a descubrir este tipo de problemas. A esto se le conoce como **análisis de condiciones de borde**.

Cuando evaluamos las casos específicos alrededor de una condición estamos hablando de **condiciones de borde**. Estos son, precisamente, los puntos donde se debe tener más cuidado.

Analizando una condición de borde

Revisemos algunos problemas que hemos visto hasta ahora y analicemos sus bordes.

En el siguiente ejemplo donde se pregunta si una persona es mayor de edad:

```
edad = int(input("¿Qué edad tienes?"))

if edad >= 18:
    print("Eres mayor de edad")
else:
    print("Eres menor de edad")
```

En este ejemplo, el punto que define una rama de la otra es el 18. A este punto se le suele llamar **punto crítico**, mientras que los bordes son los números que "bordean" el 18. Por tanto, **analizar los bordes** sería probar con los valores **17, 18 y 19**.

En el siguiente caso intenta analizar los bordes:

```
palabra = input("Ingrese una palabra")
largo = len(palabra)

if largo <= 4:
    print("Pequeña")
elif largo < 10:
    print("Mediana")
else:
    print("Larga")
```

En el ejemplo anterior los puntos críticos son 4 y 10. Por tanto, nuestros puntos de análisis deberían ser 3,4,5 y 9,10,11

Veamos un caso un poco distinto, cuando el punto crítico no es un número exacto.

```
valor1 = int(input("Ingrese valor 1"))
valor2 = int(input("Ingrese valor 2"))

if valor1 > valor2:
    print("valor1 {} es mayor".format(valor1))
elif valor1 == valor2:
    print("Ambos valores son iguales")
else:
    print("valor2 {} es mayor".format(valor2))
```

En este ejemplo, el caso crítico es cuando ambos valores son iguales, y los bordes son cuando el primer número es mayor que el segundo y cuando el primer número es menor que el segundo.

En este caso, podemos escoger un par de números para hacer la prueba, digamos 2 y 2, y luego los bordes serían 3 y 2, y 1 y 2.

Operadores lógicos

Objetivos

- Hacer uso de los operadores lógicos para evaluar y simplificar expresiones.
- Invertir una condición.

Motivación

Los operadores lógicos nos ayudan a simplificar los flujos y a evaluar condiciones más complejas. En este capítulo se enseña a utilizarlos.

¿Cómo evaluarías la siguiente expresión?__

```
a = 24
a > 20 y a < 30
```

Para probarla, aprenderemos que la expresión **"y"** se puede escribir como `and`

```
a = 24
a > 20 and a < 30 # True
```

¿Cómo evaluarías la siguiente expresión?

```
a = 32
a > 20 and a < 30
```

```
a = 32
a > 20 and a < 30 # False
```

Es falso, porque solo cumple uno de los criterios, no ambos.

Operadores lógicos

Operator	Nombre	Ejemplo	Resultado
&	y (and)	False & True	Devuelve true **si ambos operandos son true.** En este ejemplo se devuelve false.
|	o (or)	False | True	Devuelve true **si al menos una de los operando es true**.En este ejemplo devuelve true.
!=	distinto de (not)	True != False	Devuelve lo opuesto al resultado de la evaluación. En este ejemplo devuelve true.

Observemos los siguiente ejemplos:

```
nombre = 'Carlos'
apellido = 'Santana'

nombre == 'Carlos' and apellido == 'Santana'
# True

nombre == 'Carlos' and apellido == 'Vives'
# False

nombre == 'Carlos' or apellido == 'Vives'
# True
```

Identidades

Hay varias formas de expresar una afirmación en español, de la misma forma sucede en la lógica y en la programación. Por lo mismo hablamos de identidades. A continuación se muestran ejemplos de esto:

"Igual" es lo mismo que "no distinto"

Negar algo dos veces, es afirmarlo (esto no siempre es así en español, pero en programación siempre es así). Por tanto, estas dos afirmaciones son equivalentes:

```
a = 18
print(a == 18)# True
print((a != 18) != True) # True
```

Son identidades, porque para cualquier valor de a ambas expresiones siempre serán iguales. Prueba cambiando el valor asignado:

```
a = 17
print(a == 18) # False
print((a != 18) != True) # False
```

"Mayor" y "no menor igual"

Un caso similar es la comparación `a > 18`. Decir que *"a no es mayor a 18"*, es equivalente a decir que *"a no es menor o igual a 18"* (debe incluir "18" al negar).

```
a = 18
print(a > 18) # False
print((a <= 18) != True) # False
```

```
a = 19
print(a > 18) # True
print((a <= 18) != True) # True
```

Resumen

- Los *operadores lógicos* son importante porque nos ayudan a determinar si una expresión es cierta o falsa (la base de la programación).
- Los *operadores lógicos* nos pueden ayudar a simplificar expresiones.
- Se debe intentar escribir las condiciones siempre en positivo.
- (a > b) no es lo mismo que (a <= b).
- (a == b) no es lo mismo que (a != b).

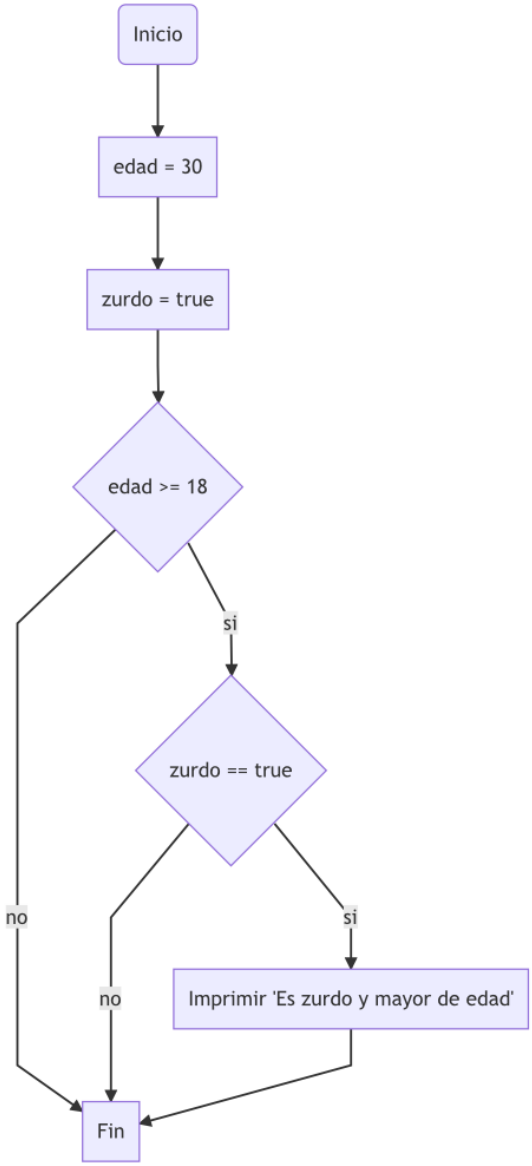
Simplificando IFs anidados

Objetivos

- Simplificar problemas con condiciones anidadas en Python.
- Entender complicaciones en `if` anidados en Python.

Identificando ifs anidados

Analicemos el siguiente ejemplo:



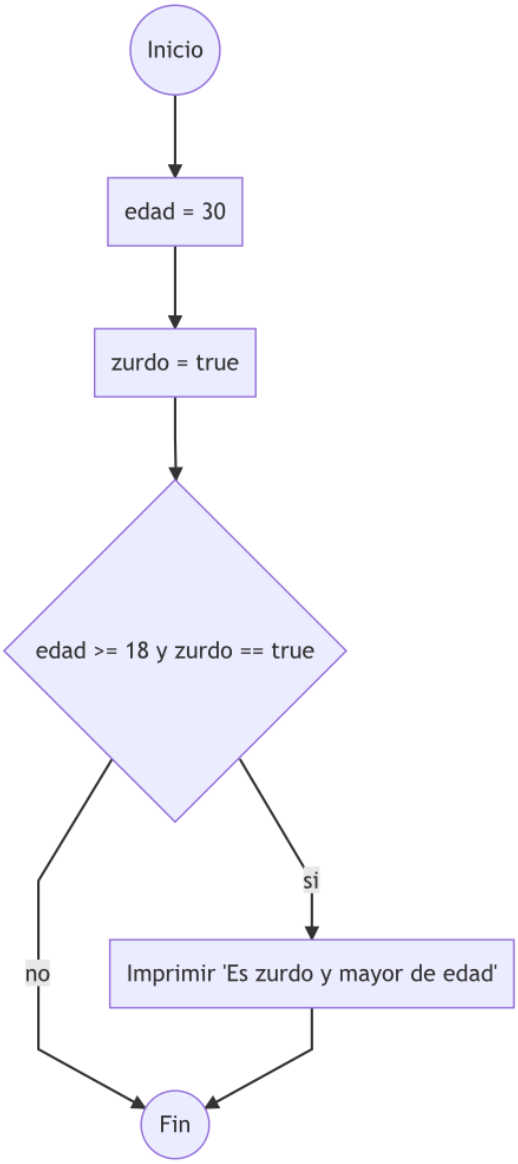
```
edad = 30
zurdo = True

if edad >= 18:
    if zurdo is True:
        print("Es zurdo y mayor de edad")
```

```
Es zurdo y mayor de edad
```

En el ejemplo anterior vemos un `if` *dentro* de otro `if`. A esto se le llama "tener `if` anidados". El código muestra el texto solo si se cumplen las dos condiciones.

Podemos simplificar el código para evaluar ambas condiciones en una misma instrucción `if`. Para una correcta utilización de instrucciones condicionales múltiples, debemos recordar la tabla de operadores lógicos:

¿Qué operador podemos utilizar si necesitamos que **ambas condiciones sean verdaderas**?




```
edad = 30
zurdo = True

if edad >= 18 and zurdo is True:
    print("Es mayor de edad y zurdo")
```

```
Es mayor de edad y zurdo
```

Ejercicio de integración

Se busca crear un programa que solicite al usuario ingresar tres números. El programa debe determinar el mayor de ellos. Se asume que los números ingresados serán distintos.

¿Cómo lo podemos resolver?

Si bien es posible utilizar `if` anidados, vamos a utilizar lo aprendido para resolverlo de forma inteligente.

- Opción 1: El **primer número** es mayor o igual que el **segundo número** y mayor o igual que el **tercer número**.
- Opción 2: El **segundo número** es mayor o igual que el **tercer número**.
- Opción 3: Cuando no se cumple ninguno de los casos anteriores.

```
primer_numero = int(input('Ingresa primer número:\n'))
segundo_numero = int(input('Ingresa segundo número:\n'))
tercer_numero = int(input('Ingresa tercer número:\n'))

if primer_numero > segundo_numero and primer_numero > tercer_numero:
    print("El primer número es el mayor")
elif segundo_numero > tercer_numero:
    print("El segundo número es el mayor")
else:
    print("El tercer número es el mayor")
```

Simplificando el flujo

Objetivos

- Simplificar el código de un script en Python utilizando buenas prácticas y la regla de condición en positivo.
- Refactorizar flujo condicional en Python.

Refactorización

Parte de la *refactorización*, es simplificar el código para hacerlo más entendible, y, en muchos casos, más óptimo en tiempo de ejecución.

Al comenzar a operar con condicionales, es común que se incurra en redundancias innecesarias. Veremos un ejemplo para ir *refactorizando* paso a paso:

```
mayor_de_edad = True
zurdo = False

if mayor_de_edad is True:
    if zurdo is True:
        print("Mayor de edad y zurdo!")
    else:
        print("Mayor de edad pero no zurdo!")
else:
    if zurdo is True:
        print("Menor de edad y zurdo")
    else:
        print("Menor de edad pero no zurdo!")
```

Primero reemplacemos los `if` anidados por condiciones múltiples:

```
if mayor_de_edad is True and zurdo is True:
    print("Mayor de edad y zurdo")
elif mayor_de_edad is True and zurdo is False:
    print("Mayor de edad pero no zurdo")
elif mayor_de_edad is False and zurdo is True:
    print("Menor de edad y zurdo")
else:
    print("Menor de edad y no zurdo")
```

Ahora podemos refactorizar las comparaciones en los condicionales.

Se mencionó con anterioridad que podemos tratar una variable como si fuera el objeto mismo. Cuando evaluamos variables booleanas en un condicional no es necesaria la comparación. Por tanto, como nuestras variables `mayor_de_edad` y `zurdo` son booleanas, podemos aplicar esto para seguir refactorizando.

¿Cómo?

Recordemos que la instrucción `if` espera que el resultado se evalúe como `true` o `false`. Por lo tanto:

```
mayor_de_edad = True

if mayor_de_edad is True:
    print("Mayor de edad")
```

```
Mayor de edad
```

Es lo mismo que:

```
mayor_de_edad = True

if mayor_de_edad:
    print("Mayor de edad")
```

```
Mayor de edad
```

La comparación `mayor_de_edad == True` es redundante, ya que la variable por sí sola se puede evaluar como `True` o `False`. Apliquemos esto en el ejemplo:

```
mayor_de_edad = True
zurdo = False

if mayor_de_edad and zurdo:
    print("Mayor de edad y zurdo")
elif mayor_de_edad and zurdo is False:
    print("Mayor de edad pero no zurdo")
elif mayor_de_edad is False and zurdo:
    print("Menor de edad y zurdo")
else:
    print("Menor de edad y no zurdo")
```

```
Mayor de edad pero no zurdo
```

Análisis Léxico

Objetivos

- Entender cómo Python interpreta el código.
- Entender importancia de escribir código lo más simple posible.
- Conocer las categorías léxicas de tokens: identificadores, operadores, delimitadores, literales, y comentarios.

Interpretación de código en Python

La forma en que un computador lee nuestro código es similar a la manera en que aprendimos a analizar la estrcutura de una oración. Python genera una interpretación mediante un programa encargado de traducir cada una de nuestras instrucciones a un lenguaje que nuestro computador entiende.

Entender cómo Python interpreta nuestras expresiones al leer un programa nos facilita entender el por qué es necesario respetar estas normas sintácticas.

¿Qué sucede cuando escribimos `python programa.py`?

Al ejecutar esta instrucción, Python lee y procesa cada una de las líneas de código, permitiendo el cálculo de operaciones y subsecuentes resultados. Resulta que suceden muchos procesos para llegar a éste resultado.

No se necesita saber estos procesos de memoria para programar en Python, pero comprender su flujo ayudará a entender de una mejor manera cómo funciona Python. De esta forma, se logra sanitizar el código al momento de escribirlo, y así anticiparse a posibles bugs.

El *proceso* de Python para interpretar el código puede resumirse en tres grandes etapas: *Tokenización, Lexing y Parsing*

Etapa 1: Tokenización

Al leer código, Python busca separar todas las palabras que componen las instrucciones en unidades llamadas **tokens**. Si tenemos la instrucción `x = 10`, Python separará la expresieon en tres partes:

- `x`
- `=`
- `10`

Etapa 2: Lexing

Python tiene una serie de definiciones y reglas para cada uno de los tokens. Mediante el proceso de **lexing**, Python imputa significado a cada componente para que posteriormente sepan cómo interactuar.

En este caso, la asignación `x = 10` funciona de la siguiente manera:

- Como `x` no está entre comillas y precede al operador de asignación `=`, se considera como una variable.
- Sabiendo que `x` se comporta como una variable, el operador de asignación busca en el lado derecho de la expresión qué imputar.
- En el lado derecho de la expresión se encuentra `10` que se asigna a la variable.

Etapa 3: Parsing

La última etapa de la expresión `x = 10` consiste en generar un árbol de *sintáxis abstracta*, que busca genera una representación del código y ordenar la forma de resolucieon de las expresiones.

Reglas Básicas

En Python existe una serie de reglas básicas que se establecen a partir de los tokens. Éstos pueden tener algunas de las siguiente categorías léxicas:

- Identificadores.
- Operadores.
- Delimitadores.
- Literales.
- Comentarios.

Identificadores

Cuando hablamos de identificadores, estamos hablando de **nombres de variables y métodos**. Para que un identificar sea válido, debe cumplir con ciertas características:

- Debe empezar con algún carácter alfabético (de la **a** a la **z**) o un guíon bajo `_`. Puede incluír números siempre y cuando no sean el primer carácter.
- Los identificadores son sensibles a la mayúsculas.
- No se pueden empezar con números.

Si intentamos crear la variable `1dia`, Python avisará que la sintáxis no es válida

```
1dia = '24 horas'
```

```
  File "<ipython-input-14-4e40c8852bb3>", line 1
    1dia = '24 horas'
       ^
SyntaxError: invalid syntax
```

Si se define una variable con la primera letra en mayúscula, y posteriormente se invoca con minúscula, Python avisará que el nombre no se ha definido, porque no se encuentra en memoria.

```
Animal = "gato"
animal
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-15-ae420d7ae5e6> in <module>()
      1 Animal = "gato"
----> 2 animal
```

```
NameError: name 'animal' is not defined
```

Operadores

Los operadores calculan un resultado en base a los valores de sus operandos. La mayoría de éstos componen de uno o dos caracteres ordinarios, pero en Python también existen algunos operandos lógicos que se escriben como palabras. Los operadores disponibles son: `+`, `-`, `*`, `/`, `//`, `%`, `-`, `**`, `=`, `!=`, `<`, `>`, `<=`, `>=`, `&`, `|`, `~`, `^`, `<<`, `>`, `and`, `in`, `is`, `not`, `or`.

Literales

Los literales son los los valores que se asignan a las variables para operar. Para almacenar y realizar manipulaciones con la información, se necesita definir el tipo de información que se almacenará y será posteriormente interpretada por Python. Ejemplos de literales incluyen a los tipos de datos `int`, `float`, `str`, `bool`.

Delimitadores

Los delimitadores son símbolos que desempeñan tres funciones en Python: agrupar, separar objetos con puntuaciones, realizar asignación o concatenación de información a objetos. Ejemplos de delimitadores de agrupación y separación son `[]`, `{}`, `()`, `:`. Ejemplos de delimitadores concatenadores incluyen `=`, `+=`, `-=`.

Comentarios

Los comentarios son tokes que serán ignorados, nos ayudarán a compentar el código y son de vital importancia para implementar documentación sobre el código. Para escribir un comentario, se debe anteponer el símbolo `#`.

```
# Esta línea es un comentario
# Los comentarios son ignorados

print(2 + 2) # También pueden existir contiguos a una línea de cógido válida

# print 2 + 3 (Esta línea también es ignorada)
```

```
4
```

Palabras reservadas

Corresponden a ciertas palabras que poseen un significado predefinido en el lenguaje y por tanto **no se pueden utilizar para nombrar variables**. Podemos solicitar la lista de palabras reservadas en Python con:

```
import keyword
keyword.kwlist
```

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