# Nivel 2: Tomando decisiones

En este nivel se presentan los conceptos necesarios para que un programa no siempre se ejecute de la misma manera, sino que pueda tomar decisiones dependiendo de las condiciones en que se ejecute y de los datos que proporcione el usuario.

Los conceptos principales que se estudian en este nivel son los siguientes:

* Valores de verdad (booleanos)
* Instrucciones condicionales
* Diccionarios
* Paso de parámetros por valor y por referencia
* Módulos adicionales

# Un programa para leer

Como dijimos en el primer capítulo, una habilidad importante que se debe desarrollar es la capacidad para leer código escrito por alguien más. En esta sección presentamos un nuevo ejemplo para que se ejercite y para que también observe varias de las características de Python que se van a estudiar en este nivel.

Al igual que en el nivel 1, el programa no tiene comentarios para que el ejercicio de lectura sea más exigente pero también más útil. Un programa tan largo como este normalmente tendría una buena cantidad de comentarios.

## Lógica de la aplicación

El siguiente es el contenido completo del módulo logica\_juego1.

import random  
  
def mano\_nueva()->dict:  
 mano = {'carta1': carta\_nueva(), 'carta2': carta\_nueva(), 'tam': 2}  
 return mano  
  
def carta\_nueva()->dict:  
 palo = nombre\_palo(random.randint(1,4))  
 valor = random.randint(1,13)  
 nombre\_carta = "{} de {}".format(nombre\_valor(valor), palo)  
 carta = {"palo": palo, "valor": valor, "nombre" : nombre\_carta}  
 return carta  
  
def agregar\_carta(mano: dict)->dict:  
 tam\_actual = mano['tam']  
 nueva\_carta = carta\_nueva()  
 nueva\_llave = 'carta{}'.format(tam\_actual + 1)  
 mano[nueva\_llave] = nueva\_carta  
 mano['tam'] = tam\_actual + 1  
 return nueva\_carta  
  
  
def nombre\_palo(num\_palo: int)->str:  
 nombre = "Picas"  
 if num\_palo == 2:  
 nombre = "Corazones"  
 elif num\_palo == 3:  
 nombre = "Tréboles"  
 elif num\_palo == 4:  
 nombre = "Diamantes"  
 return nombre  
  
  
def nombre\_valor(valor: int)->str:  
 nombre = str(valor)  
 if valor == 1:  
 nombre = "AS"  
 elif valor == 11:  
 nombre = "J"  
 elif valor == 12:  
 nombre = "Q"  
 elif valor == 13:  
 nombre = "K"  
 return nombre  
  
  
def contar\_puntos\_mano(mano: dict)->int:  
 puntos = contar\_puntos\_carta(mano, 1)  
 puntos += contar\_puntos\_carta(mano, 2)  
 puntos += contar\_puntos\_carta(mano, 3)  
 puntos += contar\_puntos\_carta(mano, 4)  
 puntos += contar\_puntos\_carta(mano, 5)  
 return puntos  
  
  
def contar\_puntos\_carta(mano: dict, numero\_carta: int)->int:  
 puntos = 0  
 llave = "carta{}".format(numero\_carta)  
 if llave in mano:  
 carta = mano[llave]  
 valor = carta["valor"]  
 if valor == 1:  
 puntos = 11  
 elif valor > 10:  
 puntos = 10  
 else:  
 puntos = valor   
 return puntos  
  
  
def casa\_debe\_continuar(mano\_casa: dict)->bool:  
 puntos\_casa = contar\_puntos\_mano(mano\_casa)  
 return puntos\_casa < 16 or puntos\_casa > 21

**Preguntas:** A partir de su lectura del programa, intente responder las siguientes preguntas. No se preocupe si no está seguro de algo, al final del nivel todas sus dudas deberían haber quedado aclaradas.

* ¿Cuál cree que es el objetivo del programa?
* ¿Cuál es el objetivo de cada bloque?
* ¿Qué elementos de la sintaxis utilizada no conoce todavía?

## Interfaz de la aplicación

El siguiente es el contenido completo del módulo interfaz\_juego1.

import logica\_juego1 as juego  
  
def mostrar\_menu()->int:  
 print("\n¿Cómo quiere continuar?")  
 print(" 1. Quiero otra carta")  
 print(" 2. No quiero más cartas")  
 respuesta = input("Seleccione una opción: ")  
 if respuesta.isnumeric():  
 opcion = int(respuesta)  
 if opcion != 1 and opcion != 2:  
 print("Sólo podía seleccionar 1 o 2. Suponemos que no quiere más cartas")  
 opcion = 2   
 else:  
 print("Sólo podía seleccionar 1 o 2. Suponemos que no quiere más cartas")  
 opcion = 2  
 return opcion  
  
  
def mostrar\_primera\_carta(mano: dict)->None:  
 plantilla = "La mano tiene {} cartas y está mostrando un {}"  
 print(plantilla.format(mano['tam'], mano['carta1']['nombre']))  
  
  
def mostrar\_mano\_abierta(mano: dict)->None:  
 num\_cartas = mano['tam']  
 puntos\_mano = juego.contar\_puntos\_mano(mano)  
 plantilla = "La mano tiene {} cartas y vale {} puntos"  
 print(plantilla.format(num\_cartas, puntos\_mano))  
 print(" La primera carta es: ", mano['carta1']['nombre'])  
 print(" La segunda carta es: ", mano['carta2']['nombre'])  
  
 if num\_cartas >= 3:  
 print(" La tercera carta es: ", mano['carta3']['nombre'])  
  
 if num\_cartas >= 4:  
 print(" La cuarta carta es: ", mano['carta4']['nombre'])  
  
 if num\_cartas >= 5:  
 print(" La quinta carta es: ", mano['carta5']['nombre'])  
  
  
def mostrar\_juego\_actual(mano\_jugador: dict, mano\_casa: dict, abrir\_casa: bool = False):  
 print("MANO JUGADOR")  
 mostrar\_mano\_abierta(mano\_jugador)  
 print("MANO CASA")  
 if abrir\_casa:  
 mostrar\_mano\_abierta(mano\_casa)  
 else:  
 mostrar\_primera\_carta(mano\_casa)  
  
  
def turno\_jugador(mano\_jugador: dict, mano\_casa: dict)->bool:  
 continuar\_juego = True  
 opcion = mostrar\_menu()  
 if opcion == 2:  
 continuar\_juego = False  
 else:  
 nueva = juego.agregar\_carta(mano\_jugador)  
 print("Tu nueva carta es: ", nueva['nombre'])  
 mostrar\_juego\_actual(mano\_jugador, mano\_casa)  
 puntos\_mano = juego.contar\_puntos\_mano(mano\_jugador)  
 if puntos\_mano > 21:  
 print("\nTienes más de 21 puntos: ¡Perdiste!")  
 continuar\_juego = False  
 elif puntos\_mano == 21:  
 print("\nTienes exactamente 21 puntos: esperemos que la casa no tenga lo mismo.\n")  
 continuar\_juego = False  
   
 return continuar\_juego  
  
  
def turno\_casa(puntos\_jugador: int, mano\_casa: dict)->bool:  
 continuar\_juego = True  
 puntos\_casa = juego.contar\_puntos\_mano(mano\_casa)  
 if puntos\_casa > puntos\_jugador:  
 print("La casa gana: {1} de la casa vs {0} del jugador".format(puntos\_jugador, puntos\_casa))  
 continuar\_juego = False  
 else:  
 if juego.casa\_debe\_continuar(mano\_casa):  
 nueva = juego.agregar\_carta(mano\_casa)  
 print("La casa saca otra carta ... {}".format(nueva['nombre']))  
 mostrar\_mano\_abierta(mano\_casa)  
   
 puntos\_casa = juego.contar\_puntos\_mano(mano\_casa)  
 if puntos\_casa > 21:  
 print("\nLa casa tiene más de 21: ¡el jugador gana!\n")  
 continuar\_juego = False  
 elif puntos\_casa == 21:  
 print("\nLa casa tiene 21: ¡la casa gana!\n")  
 continuar\_juego = False  
   
 return continuar\_juego  
  
  
def iniciar\_aplicacion()->None:  
 mano\_jugador = juego.mano\_nueva()  
 mano\_casa = juego.mano\_nueva()  
 mostrar\_juego\_actual(mano\_jugador, mano\_casa)  
  
 continuar\_juego = True  
  
 # tercera carta del jugador  
 if continuar\_juego:  
 continuar\_juego = turno\_jugador(mano\_jugador, mano\_casa)  
  
 # cuarta carta del jugador  
 if continuar\_juego:  
 continuar\_juego = turno\_jugador(mano\_jugador, mano\_casa)  
  
 # quinta carta del jugador  
 if continuar\_juego:  
 continuar\_juego = turno\_jugador(mano\_jugador, mano\_casa)  
  
 puntos\_jugador = juego.contar\_puntos\_mano(mano\_jugador)  
 if puntos\_jugador <= 21:  
 mostrar\_juego\_actual(mano\_jugador, mano\_casa, True)  
 print("\nEs el turno de la casa de jugar\n")  
 continuar\_juego = True  
   
 # tercera carta de la casa  
 if continuar\_juego:  
 continuar\_juego = turno\_casa(puntos\_jugador, mano\_casa)  
  
 # cuarta carta de la casa  
 if continuar\_juego:  
 continuar\_juego = turno\_casa(puntos\_jugador, mano\_casa)  
   
 # quinta carta de la casa  
 if continuar\_juego:  
 continuar\_juego = turno\_casa(puntos\_jugador, mano\_casa)  
  
 puntos\_casa = juego.contar\_puntos\_mano(mano\_casa)  
 if puntos\_casa < puntos\_jugador and puntos\_jugador <= 21:  
 print("¡Ganaste!: tienes {} puntos vs {} puntos de la casa".format(puntos\_jugador, puntos\_casa))  
  
  
iniciar\_aplicacion()

**Preguntas:** A partir de su lectura del programa, intente responder las siguientes preguntas. No se preocupe si no está seguro de algo, al final del nivel todas sus dudas deberían haber quedado aclaradas.

* ¿Cuál cree que es el objetivo del programa?
* ¿Qué información tendrá que suministrar el usuario que ejecute el programa?
* ¿Cuál es el objetivo de cada bloque?
* ¿Qué es lo que primero se ejecuta?

# Lógica y valores de verdad

El objetivo de esta sección es repasar conceptos fundamentales de lógica tales como el concepto de proposición, valor de verdad, conjunción, disyunción y negación. La sección presentará los conceptos desde el punto de vista de lógica matemática antes de que sean aplicados a Python en la siguiente sección.

Hasta el momento hemos trabajado en nuestros programas usando valores de tipo numéricos y con cadenas de caracteres. A partir de ahora vamos a usar también *valores de verdad*, es decir valores que sólo pueden ser verdaderos o falsos. Estos valores son también llamados valores lógicos o Booleanos (por el matemático George Boole) y se pueden operar utilizando las reglas del *álgebra Booleana*, en lugar de las reglas del álgebra elemental que usamos para valores numéricos.

En esta sección veremos cómo representar, manipular y operar con valores de verdad, haciendo un repaso rápido de lógica matemática, incluyendo las operaciones fundamentales y las leyes y propiedades que deben tenerse en cuenta para su manipulación.

## Valores de verdad y proposiciones

Una proposición es una expresión completa que tiene un valor de verdad, es decir que es verdadera o falsa. Por ejemplo, la frase ‘Hoy está haciendo calor’ es una proposición porque expresa una idea que puede ser verdadera o falsa. Evidentemente nosotros no podemos saber si, en el momento en que el lector vea esto, estará haciendo calor o no, pero el hecho de que no lo sepamos no hace que la frase deje de ser una proposición. Por el contrario, expresiones como ‘azul’, ‘mañana’ o ‘cantando’ no son frases completas, no se puede decir que sean verdaderas o falsas y por ende no pueden ser proposiciones. En español se suelen usar también los términos *sentencia*, *afirmación* o *juicio* para referirse a una proposición.

Cuando se usan proposiciones dentro del contexto de lógica matemática, se espera que tengan un valor de verdad que, como dijimos, puede ser **verdadero** o **falso**. Dependiendo del momento en que se evalúe, una proposición podría cambiar de valor (piense en la frase ‘Hoy está haciendo calor’), pero no es posible que una proposición tenga dos valores diferentes al mismo tiempo.

Finalmente, debemos recalcar que los únicos dos valores de verdad son *verdadero* y *falso*, independientemente de cómo se representen. Por ejemplo, es usual que estos valores se representen con expresiones como *V* y *F*, *true* y *false*, *T* y *F* o incluso números como *1* y *0*. Sin embargo, en todos estos casos estamos hablando de los valores de verdad y no de expresiones de tipo numérico o de cadenas de caracteres.

## Álgebra Booleana

El álgebra Booleana es la rama del álgebra que trabaja con proposiciones y no con valores numéricos como el álgebra elemental: en lugar de tener los números **0** y **1** y las operaciones de **adición** y **multiplicación** como elementos fundamentales, el álgebra Booleana se basa en los valores **verdadero** y **falso** y las operaciones ∧ (**conjunción**), ∨ (**disyunción**) y ¬ (**negación**). Debido a limitaciones en el formato de este libro, en lugar de usar siempre los símbolos usuales para estas operaciones, usaremos también las expresiones **and**, **or** y **not** para referirnos respectivamente a la **conjunción**, la **disyunción** y la **negación**.

### Operaciones lógicas y tablas de verdad

Como dijimos, son tres las operaciones básicas del álgebra Booleana. El resto de operaciones, como la implicación o la equivalencia, son operaciones secundarias que se pueden construir a partir de la conjunción, disyunción y negación.

1. **Negación**. La negación (¬, **not**, ***no***) es la operación Booleana que toma un valor de verdad y lo convierte en el otro valor. Es decir que la negación de un valor verdadero es un valor falso, y la negación de un valor falso es un valor verdadero. La negación es una operación *unaria*, que se aplica sobre un solo operando.
2. **Conjunción**. La conjunción (∧, **and**, ***y***) es una operación Booleana binaria que tiene valor verdadero sólo cuando ambos operandos tienen valor verdadero. Esto implica que cuando un operando es verdadero y el otro es falso, o cuando ambos operandos son falsos, el resultado de una conjunción es un valor falso.
3. **Disyunción**. La disyunción (∨, **or**, ***o***) es una operación Booleana binaria que tiene valor verdadero cuando por lo menos uno de los operandos tiene valor verdadero. Esto implica que una disyunción tiene valor falso sólo cuando ambos operandos tienen valor falso.

Veamos ahora algunos ejemplos de estas operaciones aplicadas a proposiciones sencillas en Español. Para esto lo primero que vamos a definir son tres proposiciones que identificaremos con las letras *P*, *Q* y *R*.

* P: Hoy está haciendo calor
* Q: Hoy es martes
* R: Hoy es un día de fiesta

Las siguientes son 6 expresiones lógicas basadas en las 3 proposiciones y en los operadores lógicos. Para cada expresión se presenta también una “traducción” lo más directa posible a una frase en Español.

1. **Q**: Hoy es martes.
2. **¬ P**: Hoy no está haciendo calor.
3. **¬ Q**: Hoy no es martes
4. **P ∧ Q**: Hoy está haciendo calor y es martes.
5. **Q ∧ ¬ R**: Hoy es martes y no es un día de fiesta.
6. **P ∨ R**: Hoy está haciendo calor o es un día de fiesta.

A continuación, analizamos estas expresiones una por una.

1. La primera expresión será verdadera sólo cuando sea evaluada un día que sea martes; de lo contrario, será falsa.
2. La segunda expresión será verdadera sólo cuando sea evaluada un día en que no haga calor. Esto quiere decir que la expresión **¬ P** será verdadera sólo cuando la expresión **P** sea falsa. De igual forma, **¬ P** será falsa sólo cuando la expresión **P** sea verdadera.
3. La tercera expresión es similar y será verdadera sólo cuando sea evaluada un día que no sea martes.
4. La cuarta expresión usa una conjunción y será verdadera sólo cuando **P** y **Q** sean verdaderas simultáneamente. Es decir, cuando sea evaluada un día en que esté haciendo calor y también sea martes. Si cualquiera de las dos partes es falsa, o si las dos son falsas, entonces la conjunción será falsa.
5. La quinta expresión usa también una conjunción, pero en este caso será verdadera sólo cuando **Q** sea verdadera y cuando **¬ R** sea verdadero, es decir cuando **R** sea falso. En otras palabras, será verdadera sólo cuando se evalúe un martes que no sea día de fiesta. Si se evalúa cualquier otro día de la semana, o si se evalúa un martes que además sea día de fiesta, entonces la expresión será falsa.
6. La sexta expresión es la que tiene una interpretación lógica que podría estar más alejada de lo que dice la intuición. En este caso se está usando una disyunción, así que tendrá un valor verdadero cuando al menos uno de los operandos sea verdadero. Esto quiere decir que la expresión será verdadera cuando esté haciendo calor (**P**) o cuando sea un día de fiesta (**R**). Si las dos partes son verdaderas (está haciendo calor y es un día de fiesta), la expresión también será verdadera. La expresión sólo será falsa cuando las dos partes sean falsas simultáneamente, es decir cuando **P** sea falsa y **R** sea falsa, o cuando **¬ P** sea verdadera y **¬ R** también sea verdadera.

Expresiones similares a este último ejemplo a veces se interpretan equivocadamente como si hubiera un *“o exclusivo”* que sólo sería verdadero cuando una de las expresiones fuera verdadera y la otra fuera falsa. Existe un operador para expresar esta relación usualmente llamado *xor*, el cual se puede construir a partir de operaciones de conjunción, disyunción y negación.

####Ejercicios####

1. Escriba la “traducción” más clara posible a español de las siguientes expresiones y analice en qué casos sería verdadera y en qué casos sería falsa:

* **Q ∨ R**
* **¬ P ∧ ¬ R**
* **Q ∧ Q**
* **Q ∧ ¬ Q**
* **P ∧ (Q ∨ R)**
* **(P ∧ Q) ∨ R**
* **(P ∧ ¬ R) ∨ (¬ P ∧ R)**

1. Escriba las expresiones equivalentes a las siguientes proposiciones en español, usando los 3 identificadores P, Q y R.

* Hoy no es martes
* Hoy no es martes o hoy es martes
* Hoy está haciendo calor o es martes o es un día de fiesta
* Hoy ni es martes ni es un día de fiesta

1. Simplifique las siguientes expresiones (escríbalas de una forma más breve) si es posible:

* **Q ∧ Q**
* **Q ∧ ¬ Q**
* **Q ∧ Verdadero**
* **Q ∧ Falso**
* **Q ∨ Verdadero**
* **Q ∨ Falso**
* **¬ Q ∧ ¬ Q**
* **¬ Q ∨ ¬ Q**
* **¬(Q ∨ P)**
* **¬(Q ∧ P)**
* **(P ∧ Q) ∨ (P ∧ R)**

### Tablas de verdad

Para estudiar el comportamiento de una expresión es usual que se utilice algo llamado una tabla de verdad. Esta no es más que una representación sencilla que nos permite analizar todos los valores posibles de una expresión, en función de los valores de las proposiciones simples involucradas.

Consideremos la tabla más sencilla posible, en la cual sólo tenemos la proposición P [[1]](#footnote-1).

|  |  |
| --- | --- |
| P | ¬ P |
| T | F |
| F | T |

Aunque sencilla, esta tabla es interesante porque nos muestra todos los posibles valores de la expresión **P** y los valores correspondientes que tendría la expresión **¬ P**. También nos muestra que las dos expresiones no pueden ser verdaderas simultáneamente, sino que tienen valores opuestos. Si extendemos la tabla con un par de expresiones encontraremos dos principios muy importantes:

|  |  |  |  |
| --- | --- | --- | --- |
| P | ¬ P | P ∧ ¬ P | P ∨ ¬ P |
| T | F | F | T |
| F | T | F | T |

En primer lugar, vemos que la expresión **P ∧ ¬ P** siempre es falsa. Esto quiere decir que no es posible que una proposición sea simultáneamente verdadera y falsa, lo cual usualmente se conoce como el principio de no contradicción.

En segundo lugar, vemos que la expresión **P ∨ ¬ P** siempre es verdadero. Esto quiere decir que siempre bien sea una proposición o su negación tienen que ser verdaderas. Como no hay una tercera opción, eso se conoce como el principio del tercero excluido.

La siguiente tabla de verdad es un poco más compleja porque involucra dos proposiciones.

|  |  |  |  |
| --- | --- | --- | --- |
| P | Q | P ∧ Q | P ∨ Q |
| T | T | T | T |
| T | F | F | T |
| F | T | F | T |
| F | F | F | F |

Acá podemos ver, de otra manera, lo que ya sabíamos sobre la conjunción y la disyunción: una conjunción es verdadera sólo cuando los dos operandos son verdaderos y una disyunción es falsa sólo cuando los dos operandos son falsos.

### Leyes fundamentales

Así como en el álgebra elemental hay algunas leyes muy importantes y conocidas (conmutatividad, asociatividad, distribución, precedencia de operadores, etc.), en el álgebra Booleana también hay algunas leyes importantes que se deben tener en cuenta y pueden servir para replantear expresiones de formas que sean más sencillas.

1. **Conmutatividad**: Tanto la conjunción como la disyunción son conmutativas, así que **A ∨ B** es equivalente a **B ∨ A**. También es cierto que **A ∧ B** es equivalente a **B ∧ A** [[2]](#footnote-2).
2. **Asociatividad**: Tanto la conjunción como la disyunción son asociativas, así que **A ∨ (B ∨ C)** es equivalente a **(A ∨ B) ∨ C**. Además, **A ∧ (B ∧ C)** es equivalente a **(A ∧ B) ∧ C** [[3]](#footnote-3).
3. **Distribución**: en el álgebra Booleana, la conjunción distribuye sobre la disyunción y viceversa [[4]](#footnote-4). Esto quiere decir que:
   * **A ∧ (B ∨ C) ≡ (A ∧ B) ∨ (A ∧ B)**
   * **A ∨ (B ∧ C) ≡ (A ∨ B) ∧ (A ∨ B)**
4. **Identidad de la conjunción**: si se hace una conjunción con el valor verdadero, el resultado es el mismo. Es decir que **A ∧ Verdadero ≡ A** [[5]](#footnote-5).
5. **Identidad de la disyunción**: si se hace una disyunción con el valor falso, el resultado es el mismo. Es decir que **A ∨ Falso ≡ A**.
6. **Dominación de la conjunción**: si se hace una conjunción con el valor falso, el resultado es falso. Es decir que **A ∧ Falso ≡ Falso** [[6]](#footnote-6).
7. **Dominación de la disyunción**: si se hace una disyunción con el valor verdadero, el resultado es verdadero. Es decir que **A ∨ Verdadero ≡ Verdadero**.

Finalmente, proponemos la siguiente equivalencia que, aunque no tiene un nombre común, es de uso extremadamente frecuente y de mucha utilidad cuando se está empezando a programar:

* **Negación y equivalencia a falso**: es lo mismo decir que una proposición tiene un valor falso que decir que su negación tiene un valor positivo. Es decir que **A ≡ Falso** es lo mismo que decir **¬ A ≡ Verdadero**. Más aún, **B ≡ Verdadero** es lo mismo que decir **B**, así que **A ≡ Falso** se puede reescribir como **¬ A**.

### Leyes de De Morgan

Finalizamos este repaso de lógica con dos teoremas muy importantes que se deben tener muy en cuenta cuando se estén reescribiendo operaciones lógicas.

Las leyes o teoremas de De Morgan dicen que:

* **¬ (P ∧ Q) ≡ ¬ P ∨ ¬ Q**
* **¬ (P ∨ Q) ≡ ¬ P ∧ ¬ Q**

Volviendo a las proposiciones de ejemplo que usamos anteriormente, la expresión **¬ (P ∧ Q)** podría “traducirse” como “no es cierto que (hoy está haciendo calor y hoy es martes)”. Hemos agregado los paréntesis para hacer notar que las palabras “no es cierto que” hacen referencia a las dos cláusulas subordinadas.

Sabemos que la expresión completa será verdadera sólo cuando la parte que está entre paréntesis sea falsa para que la negación invierta su valor. Es decir que la expresión completa será verdadera cuando sea falso que “hoy está haciendo calor y hoy es martes”. En este caso tenemos una conjunción, así que la operación será verdadera sólo cuando ambas sean falsas. Como en este caso nos interesa que sea falsa, decimos que la operación de conjunción será falsa cuando cualquiera de las dos partes sea falsa, es decir cuando la primera parte sea falsa **o** la segunda parte sea falsa. Volviendo a la expresión completa, encontramos que la expresión será verdadera cuando la proposición **P** sea falsa o cuando la proposición **Q** sea falsa.

Esto quiere decir que la expresión original también podría haberse traducido de forma equivalente como “no es cierto que hoy esté haciendo calor o no es cierto que hoy sea martes”.

La segunda propiedad es análoga a la primera.

## Ejercicios

1. Construya una tabla de verdad para verificar cada una de las leyes de De Morgan. Para la primera, necesitará una columna para **P**, una para **Q**, una para **¬ (P ∧ Q)** y una para **¬ P ∨ ¬ Q**. Debería encontrar exactamente los mismos valores en las últimas dos columnas. Para facilitar el trabajo, también le recomendamos agregar una columna con los valores de las expresiones **¬ P**, **¬ Q**, **P ∧ Q** y **P ∨ Q**.
2. En un ejercicio anterior se le pidió simplificar unas expresiones. Vuelva a hacer el ejercicio construyendo una tabla de verdad para cada caso y aplicando las leyes que se vieron al final de esta sección.
3. Como vimos, el álgebra booleana se define en términos de dos valores y tres operaciones (conjunción, disyunción y negación). Otras operaciones lógicas pueden definirse como composiciones de las operaciones básicas. Por ejemplo, la operación **xor**, que es verdadera sólo cuando exactamente uno de los operandos es verdadero, se puede definir como: **P xor Q ≡ (P ∧ ¬ Q) ∨ (Q ∧ ¬ P)**. Defina cada una de las siguientes operaciones y para cada una construya una tabla de verdad que muestre su comportamiento y le sirva para verificar su definición.

* xor (or exclusivo): es verdadero cuando exactamente un operando es verdadero.
* == (equivalencia): es verdadero cuando los operandos tienen el mismo valor.
* != (desigualdad): es verdadero cuando los operandos tienen valores diferentes.
* nand: es falso sólo cuando los dos operandos son verdaderos.
* nor: es verdadero sólo cuando los dos operandos son falsos.

# Valores de verdad en Python

El objetivo de esta sección es mostrar como se representan y operan valores de verdad dentro de un lenguaje de programación. Los conceptos que se estudiarán en esta sección son aplicables directamente a la mayoría de lenguajes de programación.

En la sección pasada trabajamos con valores de verdad y con proposiciones. En esta sección veremos que Python tiene un tipo de datos específico para este tipo de valores (bool) y que permite también la construcción de expresiones lógicas usando conjunciones, disyunciones y negaciones. También aprenderemos a escribir expresiones lógicas usando operadores relacionales sobre valores de todos los tipos que conocemos hasta el momento (números, cadenas y valores de verdad). En la próxima sección usaremos todos estos conceptos para construir expresiones condicionales.

## El tipo bool en Python

En el nivel 1 nos encontramos ya con los tipos que Python utiliza para manejar valores numéricos (int y float) y cadenas de caracteres (str). Ahora vamos a introducir el tipo bool, que se utiliza para representar valores de verdad y puede utilizarse en variables, parámetros, llamados de funciones y otras expresiones.

En comparación con los tipos que ya conocíamos, bool es mucho más sencillo puesto que sólo hay dos literales: True, que se utiliza para expresar un valor verdadero, y False, que se usa para expresar un valor falso [^sobre]. Ponga atención al uso de minúsculas y mayúsculas, puesto que los dos literales se deben escribir exactamente así.

También note que no es lo mismo 'True' que True. El primer valor es una cadena de caracteres, mientras que el segundo es un valor de verdad:

>>> type('True')  
<class 'str'>  
>>> type(True)  
<class 'bool'>

### Conversiones

Al igual que con los otros tipos, existe la función bool que permite convertir de otros tipos a booleano. Sin embargo, no es muy recomendable utilizar esta función puesto que su uso muchas veces lleva a resultados inesperados.

Por ejemplo, las siguientes expresiones tienen valor equivalente a True:

* bool('True')
* bool('true')
* bool('cadena')
* bool(5)
* bool(-5)

También tienen valor equivalente a True las siguientes expresiones, aunque naturalmente se podría esperar lo contario:

* bool('False')
* bool('false')
* bool(' ')

Por otro lado, las siguientes expresiones tienen todas valor equivalente a False:

* bool(0)
* bool(-0)
* bool('')

Como dijimos antes, es mejor limitarse a usar los literales True y False que aprenderse todas las reglas que explican estas conversiones. Todo lo que puede hacerse usando esas conversiones puede hacerse también sin tener que recurrir a ellas [[7]](#footnote-7).

### Funciones que retornan valores de verdad

El tipo bool también puede usarse para expresar el tipo de retorno de una función. Por ejemplo, las siguientes son dos funciones que retornan un valor de tipo bool: siempre que la primera función sea invocada, el resultado será un valor verdadero, mientras que para la segunda función el valor siempre será falso.

def f(x: int)->bool:  
 print('f:', x)  
 return True  
  
def g(x: int)->bool:  
 print('g:', x)  
 return False

Estas dos funciones no parecen muy interesantes porque siempre retornan el mismo valor, pero más adelante en este capítulo las utilizaremos para una importante demostración.

## Operadores Booleanos

A continuación, estudiaremos los operadores que Python ofrece para realizar las operaciones Booleanas de conjunción, disyunción y negación.

Algo importante que se debe tener en cuenta es que, aunque las operaciones son conmutativas, el orden en que se escriban los operandos tiene un efecto sobre la forma en la que se evalúan las expresiones. Para cada operador explicaremos esto más en detalle.

### Conjunción (and)

La operación lógica de conjunción se expresa en Python usando la palabra reservada and. Cuando se tiene en un programa Python una expresión de la forma p and q se sabe entonces que tendrá un valor verdadero sólo si la expresión p y la expresión q tienen simultáneamente un valor verdadero. Además, el operador and es asociativo, así que se pueden escribir expresiones como w and x and y and z sin tener que usar paréntesis y sin riesgos de que cambie el resultado.

Ahora bien, hay una pequeña pero muy significativa diferencia entre el operador and y la operación lógica de conjunción. Para estudiarla, tomemos como ejemplo la expresión w and x and y and z. Si se tratara de una expresión lógica, la podríamos haber escrito en cualquier orden y el resultado sería siempre el mismo debido a la conmutatividad de la conjunción. Es decir que para nosotros sería exactamente lo mismo escribir z and x and w and y.

En el caso de la expresión Python entran a jugar también consideraciones prácticas. Por ejemplo, suponga que x es un valor de verdad extremadamente difícil de calcular [[8]](#footnote-8) y suponga también que sabemos que w es una expresión falsa. No tendría sentido buscar el valor de x puesto que sabríamos de antemano que la expresión completa sería falsa.

La diferencia entonces entre el operador and de Python y la operación de conjunción es que Python se aprovecha de las propiedades de identidad y dominancia de la conjunción ( *p∧Verdadero=p* y \*p∨Falso=Falso) para evitar calcular más términos de lo que necesario.

Esto quiere decir que cuando se tiene una operación de conjunción primero se evalúa sólo el primer término: si su valor es falso, se sabe que toda la expresión será falsa y no tiene sentido seguir avanzando. Pero si el valor del término es verdadero, entonces se puede concluir que el valor de la expresión será igual al valor de la parte restante de la expresión.

Apliquemos esto a nuestro ejemplo w and x and y and z:

* Python primero revisará el valor de w. Si es falso, Python inmediatamente dirá que la expresión completa es falsa. De lo contrario, el valor de la expresión completa será el valor de la expresión x and y and z.
* Si no hemos terminado, Python revisará el valor de x. Si es falso, Python inmediatamente dirá que la expresión completa es falsa. De lo contrario, el valor de la expresión completa será el valor de la expresión y and z.
* Si no hemos terminado, Python revisará el valor de y. Si es falso, Python inmediatamente dirá que la expresión completa es falsa. De lo contrario, el valor de la expresión completa será el valor de la expresión z.
* Si no hemos terminado, Python revisará el valor de z y como no hay más operaciones le asignará a la expresión el valor de z.

Veamos ahora esto mismo con un ejemplo un poco más elaborado en el que vamos a tener nuestras dos funciones que siempre retornan el mismo valor: la función f siempre retornará el valor True mientras que la función g siempre retornará el valor False. Además, cada función dejará una traza en la consola para que podamos ver en qué orden fueron llamadas.

def f(x: int)->bool:  
 print('f:', x)  
 return True  
  
def g(x: int)->bool:  
 print('g:', x)  
 return False  
  
print("Caso 1 - f and f and f :")  
print(f(1) and f(2) and f(3))  
  
print("Caso 2 - f and f and g :")  
print(f(1) and f(2) and g(3))  
  
print("Caso 3 - f and g and g :")  
print(f(1) and g(2) and g(3))  
  
print("Caso 4 - g and g and g :")  
print(g(1) and g(2) and g(3))

Las instrucciones que se encuentran al final del programa se encargarán de evaluar una expresión basada en invocaciones a f y g e imprimirán el resultado. Veamos ahora el resultado de ejecutar el programa:

Caso 1 - f and f and f :  
f: 1  
f: 2  
f: 3  
True  
Caso 2 - f and f and g :  
f: 1  
f: 2  
g: 3  
False  
Caso 3 - f and g and g :  
f: 1  
g: 2  
False  
Caso 4 - g and g and g :  
g: 1  
False

Lo que vemos en el caso 1 es que la función f se invoca tres veces y deja traza de las tres invocaciones (podemos ver que se llamó a la función f y el valor que se le asignó al parámetro x). También vemos que el resultado que se imprime es True y corresponde al valor de la operación de conjunción.

En el caso 2 lo que vemos es que se invocó dos veces la función f y luego se invocó la función g. El resultado de esta expresión es False.

El tercer caso es mucho más interesante: podemos ver que la función f se invocó una vez, pero la función g sólo se invocó una vez en lugar de dos veces. Esto se debe a que, una vez se encontró el valor False que retornó la función g, ya se conocía el valor de la expresión completa y no tenía sentido continuar evaluando los otros términos.

En el cuarto caso nos encontramos el mismo comportamiento: sólo se evaluó el primer término y, como tenía valor falso, hizo que toda la expresión fuera falsa sin tener que evaluar el resto de los términos.

### Disyunción (or)

La operación lógica de disyunción se expresa en Python usando la palabra reservada or. Cuando se tiene en un programa Python una expresión de la forma p or q se sabe entonces que tendrá un valor falso sólo si la expresión p y la expresión q tienen simultáneamente un valor falso. Además, el operador or es asociativo, así que se pueden escribir expresiones como w or x or y or z sin tener que usar paréntesis y sin riesgos de que cambie el resultado.

Al igual que con la operación and, Python también busca eficiencias en la evaluación de expresiones que usen en este operador. Esto se logra identificando inmediatamente que el resultado de una operación or entre el valor verdadero y cualquier valor, siempre será verdadero. Es decir que, si el primer operando de una operación que use el operador or es verdadero, entonces el valor de la operación será verdadero y no será necesario evaluar los otros términos.

Veamos esto agregándole unas instrucciones adicionales a nuestro programa anterior:

print("Caso 1 - f or f or f :")  
print(f(1) or f(2) or f(3))  
  
print("Caso 2 - f or f or g :")  
print(f(1) or f(2) or g(3))  
  
print("Caso 3 - g or f or g :")  
print(g(1) or f(2) or g(3))  
  
print("Caso 4 - g or g or g :")  
print(g(1) or g(2) or g(3))

Ahora se evaluarán expresiones basadas en disyunciones y tendremos un comportamiento muy diferente al del caso anterior:

Caso 1 - f or f or f :  
f: 1  
True  
Caso 2 - f or f or g :  
f: 1  
True  
Caso 3 - g or f or g :  
g: 1  
f: 2  
True  
Caso 4 - g or g or g :  
g: 1  
g: 2  
g: 3  
False

En el primer caso, vemos que Python inmediatamente idéntica que el primer término tiene valor verdadero. Esto implica que la expresión completa tendrá valor verdadero y la evaluación termina sin tener que evaluar los otros términos.

En el segundo caso pasa algo exactamente igual: a pesar de que los otros términos tienen valores diferentes, es suficiente con evaluar el primer término para saber cuál será el valor de toda la expresión.

En el tercer caso lo que vemos es que se evalúa el primer término que, como sabemos por la definición de la función, es falso. Python tiene entonces que recurrir al segundo término, que en este caso es verdadero. Esto hace que la expresión completa sea verdadera y no sea necesario seguir con el resto de la evaluación.

Sólo en el cuarto caso se evalúan los tres términos: el primero es falso, obligando a la evaluación del segundo, que también es falso. Finalmente se evalúa el tercer término y, como también es falso se concluye que la expresión completa era falsa.

### Negación (not)

El tercer operador lógico en Python es la negación, la cual se expresa con la palabra reservada not. Como se trata de un operador unario se debe anteponer al operando. Es decir que se usa igual que como se usa el signo - para convertir a un número en su negativo (por ejemplo, 5, -5 y -(-5))).

El operador not no tiene ninguna diferencia con la operación lógica de negación y tiene una precedencia que es mayor a la de la conjunción y a la de la disyunción. Es decir que not a and b siempre será equivalente a (not a) and b y que a and not b siempre será equivalente a a and (not b).

Veamos esto en un ejemplo:

print("Caso 1:", not f(1) and f(2))  
print(not f(1) and f(2))  
print("Caso 2:", not g(1) and f(2))  
print(not g(1) and f(2))

El resultado de la ejecución de este programa se muestra a continuación:

Caso 1: not f(1) and f(2)  
f: 1  
False  
Caso 2: not g(1) and f(2)  
g: 1  
f: 2  
True

En el primer caso vemos que se evaluó la función f y se obtuvo necesariamente el valor True. A continuación, se aplicó el operador de negación y se obtuvo el valor False. En ese punto Python pudo descubrir que la expresión completa iba a tener valor False y la evaluación terminó con ese resultado.

En el segundo caso vemos que, por el contrario, el primer término de la conjunción tiene valor True porque primero se evaluó g y luego se aplicó la negación. Después de esto se hizo la evaluación del segundo término y se llegó al valor final: True.

Note que en ambos casos la evaluación de la operación de negación se hizo **antes** de la evaluación de la operación de conjunción.

[^ sobre]: En realidad, en Python se pueden usar otros literales para expresar valores verdaderos y falsos, pero en general es mucho más claro cuando se usan los literales True y False. Por ejemplo, las siguientes dos expresiones tienen valores que podríamos llamar *desconcertantes*: 1. not -66 and 'a' 2. not -66 and 'a'. La primera expresión tiene valor False, mientras que la segunda tiene valor 'a'. Aunque esos resultados no son un error y están perfectamente justificados en la especificación de Python, debería ser mucho más fácil deducir el valor de una expresión sin tener que conocer detalles de implementación relativamente oscuros.

#### Ejercicios

1. ¿Cuál es el valor de las siguientes expresiones Python basadas en conjunciones? ¿Cuáles de las reglas y propiedades que se estudiaron en la sección anterior se aplican en cada caso?

* False and p
* True and p
* p and True
* p and False
* p and q and r

1. ¿Cuál es el valor de las siguientes expresiones Python basadas en disyunciones? ¿Cuáles de las reglas y propiedades que se estudiaron en la sección anterior se aplican en cada caso?

* False or p
* True or p
* p or True
* p or False
* p or q or r

1. ¿Cuál es el valor de las siguientes expresiones Python basadas en negaciones?

* not True
* not not True
* not not not False

1. ¿Cuáles de las reglas y propiedades que se estudiaron en la sección anterior se podrían aplicar en cada una de las siguientes expresiones?

* not p and not q
* (p and q) or (q and r)
* p and (q or r)
* p or (q and r)

## Operadores relacionales

Hasta este momento todas las operaciones que hemos visto cuyo resultado es un valor de verdad han utilizado otros valores de verdad como operandos. Los operadores relacionales que veremos a continuación nos permitirán hacer comparaciones entre diferentes valores con el fin de obtener un valor de verdad.

La siguiente tabla resume los operadores disponibles en Python para hacer comparaciones entre valores. Estos operadores pueden aplicarse a cualquier tipo de valor, aunque no necesariamente el resultado tendrá un sentido muy evidente. Por ejemplo, la expresión True < False es falsa, pero no es fácil imaginarse una razón para comparar dos valores booleanos usando ese operador.

|  |  |  |  |
| --- | --- | --- | --- |
| Operador | Significado | Ejemplo | Resultado |
| < | Es menor que … | 4 < 7 | True |
| <= | Es menor o igual que … | “Ab” <= “ab” | True |
| > | Es mayor que … | 4.5 > 7.1 | False |
| >= | Es mayor o igual que … | ‘1A’ >= ‘A1’ | False |
| == | Es igual a … | “abc” == “ab” + “c” | True |
| != | Es diferente de … | 4 != 7 | True |
| is | Dos *objetos* son el mismo | 4 is 7 | False |
| is not | Dos *objetos* no son el mismo | 4 is not 7 | True |

### Operadores de orden

Los primeros cuatro operadores tienen un comportamiento intuitivo cuando se aplican sobre valores numéricos (int y float). Cuando se aplican sobre cadenas de caracteres, la comparación se hace de forma lexicográfica (como se organizarían las palabras en un diccionario). De esta manera, la cadena "hola" debería ser *menor* que la cadena mundo. Sin embargo, en este sistema las mayúsculas siempre son *menores* que las minúsculas, así que la expresión 'Z' < 'a' siempre será verdadera. Por su parte, los números son *menores* que las mayúsculas, y algunos símbolos son menores que los números [[9]](#footnote-9). Si se quiere hacer una comparación más sencilla se pueden convertir las cadenas a letras minúsculas o mayúsculas usando uno de los mecanismos que veremos en una de las siguientes secciones.

Si se intentan usar estos 4 operadores entre valores de tipos diferentes se presentará un error similar al siguiente:

TypeError: '<' not supported between instances of 'int' and 'str'

Las siguientes funciones nos servirán para ilustrar algunas de las ideas que acabamos de presentar.

def es\_positivo(x: int)->bool:  
 positivo = x > 0  
 return positivo  
  
def es\_negativo(x: int)->bool:  
 negativo = x > 0  
 return negativo  
   
def es\_cero(x: int)->bool:  
 positivo = es\_positivo(x)  
 negativo = es\_negativo(x)  
 return not positivo and not negativo  
   
def ordenadas(antes: str, despues: str)->bool:  
 """   
 Revisa si dos cadenas en un diccionario están ordenadas lexicográficamente  
 Parámetros:  
 antes (str): una cadena que está antes que la otra en un diccionario  
 despues (str): una cadena que está después que la otra en un diccionario  
 Retorno:  
 (bool): Indica si las cadenas estaban ordenadas.  
 El resultado será verdadero si la cadena 'antes' tiene que ir   
 antes que la cadena 'despues' en orden lexicográfico.  
 El resultado será falso de lo contrario.   
 """  
 estan\_ordenadas = antes < despues  
 return estan\_ordenadas

La primera función revisa si el valor que se pasa en el parámetro x es mayor que 0. En caso de que esto sea cierto, en la variable positivo debería quedar el valor True y este valor sería retornado por la función. En caso contrario (es decir, si x <= 0), la función retorna False.

La segunda función hace algo análogo, pero para ver si el valor que se pasa como parámetro es negativo.

La tercera función combina las dos funciones anteriores para ver si el valor x es cero: si el número *no* es positivo y *no* es negativo, entonces tiene que ser el valor 0.

La cuarta función utiliza las comparaciones lexicográficas para ver si dos palabras extraídas de un diccionario estaban correctamente ordenadas. En la variable estan\_ordenadas queda el resultado de comparar las palabras lexicográficamente, y este valor posteriormente se retorna.

### Operadores de igualdad

Los siguientes operadores (== y !=) sirven para establecer si dos valores son iguales o no.

**Atención**: no confunda el operador == con la instrucción de asignación =. En el primer caso el resultado será un valor de verdad mientras que en el segundo caso se modificará el valor de una variable.

En el ejemplo de la tabla se puede apreciar un caso muy interesante: a la derecha del operador == tenemos la cadena "abc", mientras que a la derecha tenemos la expresión "ab" + "c". Por lo que aprendimos en el capítulo anterior, sabemos que el valor de la parte de la derecha será la concatenación de las dos cadenas, así que será abc. Como las dos cadenas son iguales (tienen los mismos caracteres, en el mismo orden), el valor de la expresión será verdadero.

Por otro lado, el operador != sirve para comparar dos valores y saber si son diferentes. Esto es equivalente a utilizar el operador de igualdad y luego negar el resultado.

A continuación, presentamos la definición de 3 funciones que nos servirán para ilustrar el uso de los operadores que acabamos de introducir.

def es\_par(x: int)->bool:  
 # En la siguiente línea sobran los paréntesis,   
 # pero si incluyeron para hacer más claro el código  
 residuo\_cero = (x % 2 == 0)  
 return residuo\_cero  
  
def no\_es\_7\_v1(x: int)->bool:  
 # En la siguiente línea sobran los paréntesis,   
 # pero si incluyeron para hacer más claro el código  
 x\_es\_7 = (x == 7)   
 return not x\_es\_7  
   
def no\_es\_7\_v2(x: int)->bool:  
 return x != 7

La primera función verifica si un número es par. Para eso, calcula el residuo del número módulo 2 y lo compara con 0: si son iguales, significa que el número era par y la función retorna el valor True; si son diferentes, la función retorna el valor False.

La siguiente función es muy parecida, aunque en este caso lo que se retorna es la negación de la variable x\_es\_7. En este caso queríamos mostrar que es posible negar un valor justo antes de retornarlo.

La tercera función hace lo mismo que la segunda (dice si un número NO es 7), pero lo hace en la misma línea en la cual hace el retorno. Si analizamos con detenimiento esta función, veremos que lo primero que se hace es comparar el valor del parámetro x con el número 7 para ver si son diferentes (verdadero si son diferentes, falso si son iguales). El resultado de esta comparación es inmediatamente retornado y la función termina su ejecución.

**Atención**: Tenga mucho cuidado cuando haga comparaciones con números decimales. Mientras que las comparaciones de enteros usualmente no tienen problemas, las comparaciones entre decimales pueden tener inconvenientes por culpa de errores en la precisión de los cálculos en Python [[10]](#footnote-10). Por ejemplo, el valor de evaluar la expresión 0.2 == 1.2 - 1.0 es, sorprendentemente, False. Para entender por qué, se puede evaluar la parte derecha en el intérprete de Python, obteniendo el valor 0.19999999999999996 que, aunque es muy cercano a 0.2, no es idéntico.

Para evitar este problema, la recomendación general es que, cuando se vayan a comparar valores numéricos con decimales se defina una precisión y se revisen los intervalos, como en el siguiente ejemplo:

epsilon = 0.005  
iguales = (0.2 > (1.2 - 1.0) - epsilon) and (0.2 < (1.2 - 1.0) + epsilon)

En este caso, en lugar de comparar 0.2 con el resultado de la resta 1.2 - 1.0 estamos mirando que esté dentro del intervalo que va desde 1.2 - 1.0 - epsilon hasta 1.2 - 1.0 + epsilon. El valor de epsilon se puede definir tan grande o tan pequeño como sea conveniente para el problema que se esté resolviendo.

### Operadores de identidad

Finalmente, los operadores is y is not sirven para revisar que dos valores no sólo sean iguales, sino que también sean el mismo. Como en este libro no vamos a ocuparnos mucho de objetos, no ahondaremos en la distinción entre == e is.

Más adelante explicaremos en qué casos tiene sentido utilizar el operador is not en el contexto de la expresión is not None.

## Ejercicios [[11]](#footnote-11)

1. Escriba una función que dada la edad de una persona indique si puede manejar (tiene que tener al menos 16 años)
2. Escriba una función que dada la altura en metros y el peso en kilogramos de un adulto diga si está dentro de los rangos típicamente considerados saludables. Para esto debe usar el Índice de Masa Corporal (BMI), que se calcula como *peso/altura2*. Un adulto se considera que tiene sobrepeso cuando su BMI es mayor o igual a 25. Un adulto se considera que está bajo de peso cuando su BMI es menor a 18.5.
3. Escriba una función que dados dos números diga si el primero es divisible por el segundo.
4. Queremos saber si una persona tiene el dinero suficiente para pagar la cuenta en un restaurante dados los siguientes parámetros: la cantidad de dinero que tiene la persona, el valor de la cuenta, si la persona va a dejar propina o no. La propina corresponde al 10% del valor de la cuenta.

## Más allá de Python

Al igual que Python, muchos lenguajes utilizan tipos especiales para representar valores de verdad, pero incluyen también convenciones tales como usar 0 para falso y cualquier otro número para verdadero. Por ejemplo, en PHP existen los literales ‘true’ y ‘false’ (escritos sin importar el uso de mayúsculas o minúsculas), pero también tienen un valor falso los números 0, 0.0, -0 y -0.0, las cadenas vacías, la cadena “0”, un arreglo sin elementos, y el tipo NULL. Algo similar sucede en C y C++. Por el contrario, en Java los únicos literales para valores booleanos son ‘true’ y ‘false’. Como se dijo en la nota sobre el caso de Python, en general es mucho mejor que el código sea claro y se usen los valores booleanos de forma explícita en lugar de depender de conversiones.

La explicación que se presentó sobre el orden en el que se evalúan los términos de una conjunción o una disyunción en general aplica para todos los lenguajes de programación. C, C++, Java, PHP y JavaScript, entre muchos otros, implementan ideas similares.

La distinción entre *son iguales* y *son el mismo* es sutil y se presta para confusiones en diferentes lenguajes. Por ejemplo, en JavaScript existen los operadores == y === para diferenciar entre los todos tipos de comparación, mientras que en Java el operador == representa uno o el otro dependiendo de qué se esté comparando. Más aún, en Java los objetos de tipo String se manejan de forma diferente dependiendo de en qué momento se les asigne su valor, haciendo que a veces dos cadenas iguales sean también la misma y a veces no lo sean. Entender con absoluta claridad este punto para el lenguaje de programación que esté usando le evitará muchísimos dolores de cabeza.

# Instrucciones condicionales

El objetivo de esta sección es introducir las instrucciones condicionales, mostrando por qué son necesarias, qué significan y cómo pueden utilizarse en Python.

Entre el nivel 1 y lo que llevamos hasta este momento del nivel 2, hemos estudiado tipos de datos (int, str, float, bool), operadores (aritméticos, sobre cadenas, lógicos), expresiones y variables, y un conjunto reducido de instrucciones que nos permiten:

* Asignarle un valor a una variable
* Definir una nueva función
* Invocar una función con unos argumentos
* Retornar un valor al terminar una función
* Importar un módulo existente

Aunque no parece mucho, este reducido conjunto de elementos son la base para construir cualquier tipo de programas. En esta sección vamos a introducir un nuevo tipo de instrucción, los condicionales, con el cual incrementaremos enormemente el potencial de problemas que podemos solucionar y de programas que podemos construir.

## Motivación

La gran limitación que tienen los programas que podemos construir hasta el momento es que siempre van a ejecutar las mismas instrucciones. Esto implica que no pueden tomar ninguna decisión y que incluso funcionalidades pequeñas son imposibles de implementar.

Tomemos como ejemplo la funcionalidad de verificar si la contraseña introducida por un usuario es correcta o no: si es correcta, quisiéramos informarle al usuario que sí es correcta; si no es correcta, quisiéramos decirle que no es correcta y que debe volver a intentarlo. En la sección anterior aprendimos a comparar cadenas de caracteres, así que ya podríamos saber si la contraseña es correcta o no:

contrasena\_correcta = contrasena\_almacenada == contrasena\_ingresada

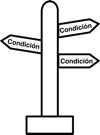
Lo que no podemos hacer, sin usar instrucciones condicionales, es *calcular* el mensaje correcto para mostrarle al usuario con base en el valor de contrasena\_correcta.

Consideremos ahora otro ejemplo en el que queremos calcular el valor de una factura:

valor\_producto = 14000  
impuestos = valor\_producto \* 0.19  
valor\_factura = valor\_producto + impuestos

¿Qué pasa si no todos los productos tienen que pagar impuestos? ¿Qué pasa si el porcentaje de impuestos es diferente dependiendo del valor del producto (ej. 19% para productos de más de 10000 y 10% para productos de menos)? Para lograr representar estas situaciones también necesitamos usar instrucciones condicionales.

Cuando una persona recorre un camino, es usual que llegue a un punto en el que tiene que detenerse, analizar algo y tomar una decisión sobre qué camino seguir. Eso es precisamente lo que nos van a permitir modelar las instrucciones condicionales: marcar un punto en el programa en el que se tiene que evaluar una condición y continuar por un camino dependiendo del resultado de la evaluación.



## Condicionales en Python: IF-ELSE

Una instrucción condicional es una instrucción cuya ejecución depende del valor que resulte de evaluar una expresión booleana. Esta evaluación se hace durante la ejecución de un programa, así que cuando estamos programando nosotros no podemos saber a ciencia cierta qué es lo que va a ejecutarse. Además, ejecuciones sucesivas del mismo programa no necesariamente van a comportarse de la misma manera.

En Python, las instrucciones condicionales se usan utilizando la siguiente estructura:

if expresion\_booleana:  
 cuerpo\_if  
else:  
 cuerpo\_else

* **expresion\_booleana**: es una expresión que tiene que tener un valor de verdad (una operación relacional, una variable booleana, un llamado a una función booleana, etc.). Note los dos puntos (**:**) al final de la expresión.
* **cuerpo\_if**: son las instrucciones que se van a ejecutar si la expresión booleana es verdadera en el momento de la evaluación. El cuerpo del IF debe ir indentado como el cuerpo de una función.
* **cuerpo\_else**: son las instrucciones que se van a ejecutar si la expresión booleana es falsa en el momento de la evaluación. El cuerpo del ELSE también debe ir indentado.

Veamos ahora cómo podemos usar esta estructura usando el ejemplo de la verificación de contraseña:

contrasena\_almacenada = "secreto"  
contrasena\_ingresada = input("Ingrese su contraseña:")  
  
if contrasena\_almacenada == contrasena\_ingresada:  
 mensaje = "Su contraseña es correcta"  
else:  
 mensaje = "La contraseña es incorrecta"  
  
print(mensaje)

En primer lugar, tenemos que notar que este programa tiene sólo cuatro instrucciones: 1. una asignación, 2. un llamado a una función (input), 3. una instrucción condicional y 4. una invocación a una función (print). Como veremos, la instrucción condicional tiene varios elementos por dentro, pero deberíamos considerarla una sola instrucción.

En la primera instrucción se hace una asignación y se almacena en una variable la cadena 'secreto' para que ese sea la contraseña contra la que se va a hacer la verificación.

En la segunda instrucción se hace una invocación a la función input y se espera a que el usuario ingrese un valor. Ese valor se almacena en la variable contrasena\_ingresada pero nosotros no lo conocemos, así que no podemos saber si es el valor correcto o no.

La siguiente instrucción en el programa es el if con todos sus elementos. En primer lugar, Python verifica el valor de la expresión booleana para ver cuál es la siguiente instrucción que tendrá que ejecutar. En este caso la expresión es contrasena\_almacenda == contrasena\_ingresada y tiene un valor que nosotros no podemos conocer porque depende de lo que escriba el usuario cada vez que se ejecute el programa. Lo que sí sabemos es que si el valor es verdadero se ejecutará el *cuerpo del if* y que si es falso se ejecutará el *cuerpo del else*.

El *cuerpo del if* en este caso sólo tiene una instrucción que hace una asignación a la variable mensaje con un valor que indique una autenticación exitosa. El *cuerpo del else* también tiene sólo una asignación, pero esta vez deja un mensaje fallido. Estos dos bloques de código (el cuerpo del if y el cuerpo del else) son muy sencillos en este caso, pero podrían ser tan complicados como se quisiera: dentro de ellos puede haber casi cualquier cosa que pueda estar dentro de un programa, incluyendo asignaciones, llamados de funciones y otros condicionales [[12]](#footnote-12).

Un aspecto muy importante para tener en cuenta acá es la **indentación**: así como cuando definimos una función tenemos que indentar el cuerpo de la función, cuando usamos instrucciones condicionales el cuerpo del if y el cuerpo del else tienen que estar consistentemente indentados.

Cuando se termine de ejecutar bien sea el cuerpo del if o el cuerpo del else, se considerará que la instrucción condicional terminó de ejecutarse y Python podrá pasar a la siguiente instrucción, que en este imprime el mensaje para el usuario.

### Alternativas

Veamos ahora otras dos maneras en las que habríamos podido escribir el mismo programa.

En la primera alternativa vamos a extraer la instrucción condicional y convertirla en una función que calcule el mensaje.

def revisar(contrasena\_almacenada: str, contrasena\_ingresada: str)->str:  
 contrasena\_correcta = contrasena\_almacenada == contrasena\_ingresada  
   
 if contrasena\_correcta:  
 mensaje = "Su contraseña es correcta"  
 else:  
 mensaje = "La contraseña es incorrecta"  
 return mensaje  
  
contrasena\_almacenada = "secreto"  
contrasena\_ingresada = input("Ingrese su contraseña:")  
mensaje = revisar(contrasena\_almacenda, contrasena\_ingresada)  
print(mensaje)

Este pequeño cambio hace mucho más evidente lo que dijimos antes: la instrucción condicional, aunque tuviera varios elementos por dentro, era fundamentalmente sólo una instrucción que calculaba el valor de la variable mensaje.

Esta nueva versión también tiene un cambio con respecto a la expresión condicional: la comparación no se hace ahora dentro del if sino que se hace antes y su resultado se almacena en una variable. Aunque funcionalmente es equivalente, es posible que esta versión sea un poco más legible y haga que la instrucción condicional sea más fácil de entender. En ejercicios más avanzados, en los que las condiciones son mucho más largas y complejas, es recomendable aplicar esta técnica para simplificar el código y disminuir la posibilidad de introducir errores.

La siguiente es la segunda alternativa al código original: en este caso no hay una función adicional pero tampoco hay un else.

contrasena\_almacenada = "secreto"  
contrasena\_ingresada = input("Ingrese su contraseña:")  
  
mensaje = "La contraseña es incorrecta"  
if contrasena\_almacenada == contrasena\_ingresada:  
 mensaje = "Su contraseña es correcta"  
  
print(mensaje)

Como se ve en este programa, no es obligatorio que cada instrucción condicional tenga un bloque else: sólo el bloque if tiene que existir siempre en una instrucción condicional. En estos casos, la condición sirve para saber si un bloque de instrucciones tiene que ejecutarse o no.

Volviendo a nuestro ejemplo, el bloque condicional revisa que la contraseña sea correcta y, en caso afirmativo, le asigna a la variable mensaje la cadena que informa del éxito en la autenticación. Como en este caso la variable mensaje siempre tiene que tener un valor, bien sea exitoso o no, tuvimos que asignarle un valor inicial antes de llegar a la instrucción condicional. Si no lo hubiéramos hecho, la variable mensaje no existiría cuando el usuario pusiera la contraseña equivocada y nuestro programa habría fallado al momento de intentar imprimir el mensaje:

NameError: name 'mensaje' is not defined

### Ejercicios

1. Queremos escribir una función que nos diga quién ganará en el lanzamiento de una moneda. Escriba una función que reciba el nombre de la persona que pidió ‘cara’, el nombre de la persona que pidió ‘sello’ y el resultado del lanzamiento (‘cara’ o ‘sello’) y responda con el nombre de la persona que haya acertado.

## Condicionales en Python: IF-ELIF-ELSE

Vimos ya que una instrucción condicional puede tener una pareja de bloques if y else, o puede tener sólo un bloque if. Ahora agregaremos la posibilidad de tener más de una condición, para lo cual tenemos que introducir los bloques que en Python se llaman elif y que en muchos otros lenguajes se conocen como **else if**.

Veamos un ejemplo de una instrucción que incluya varias condiciones. Suponga que las funciones longitud, tiene\_numeros, tiene\_minusculas, tiene\_mayusculas y cambiar\_contrasena ya están definidas en alguna otra parte del programa. El bloque condicional empieza con la palabra reservada if y una condición; de ahí en adelante, cada condición se acompaña de la palabra reservada elif. Al final encontramos un bloque ELSE que, como todos los bloques ELSE que hemos estudiado, incluye la palabra else pero no tiene una condición asociada.

nueva = input("Introduzca su nueva contraseña:")  
  
if longitud(nueva) <= 8:  
 mensaje = "La nueva contraseña debe tener al menos 8 caracteres"  
   
elif nueva == contrasena\_anterior:  
 mensaje = "La nueva contraseña no puede ser igual a la anterior"  
  
elif nueva.isalnum():  
 mensaje = "La nueva contraseña debe tener signos de puntuación"  
   
elif not tiene\_numeros(nueva):  
 mensaje = "La nueva contraseña debe tener al menos un número"  
  
elif not tiene\_mayusculas(nueva):  
 mensaje = "La nueva contraseña debe tener al menos una letra mayúscula"  
  
elif not tiene\_minúsculas(nueva):  
 mensaje = "La nueva contraseña debe tener al menos una letra minúscula"  
  
else:  
 cambiar\_contrasena(nueva)  
 mensaje = "La contraseña se cambió exitosamente"  
  
print(mensaje)

En este programa lo que tenemos es una sola instrucción condicional que incluye la evaluación de varias condiciones. La primera condición que se evalúa es la que tiene que ver con la longitud de la cadena (longitud(nueva) <= 8) para ver si se ejecuta el primer cuerpo o no. Si la condición es falsa, se pasa a revisar la segunda condición. Lo interesante es si la condición es verdadera, porque entonces se ejecuta el cuerpo del IF y se pasa a la siguiente instrucción del programa. Es decir que se pasa a imprimir el mensaje.

Esto es muy importante y debe quedar muy claro: cuando se encuentra una condición verdadera, se ejecuta el cuerpo asociado a la condición y se termina la ejecución del condicional sin importar si más adelante había otras condiciones que también fueran verdaderas.

La implicación de esto para nuestro programa es que el mensaje que le vamos a mostrar al usuario es el mensaje que corresponda al primer error que hayamos encontrado. Por ejemplo, si la nueva contraseña no tenía signos de puntuación y no tenía números, el usuario sólo se enterará del primer problema.

Finalmente, analicemos las estructuras de los siguientes condicionales. En el primer condicional, usaremos bloques ELIF.

if cond1:  
 bloque1()  
elif cond2:  
 bloque2()  
elif cond3:  
 bloque3()  
else:  
 bloque\_else()

Ahora veamos una estructura equivalente pero que no utiliza bloques ELIF.

if cond1:  
 bloque1()  
if not cond1 and cond2:  
 bloque2()  
if not cond1 and not cond2 and cond3:  
 bloque3()  
if not cond1 and not cond2 and not cond3:  
 bloque\_else()

En este segundo caso se remplazaron los bloques ELIF por bloques IF y se volvieron mucho más complejas las condiciones. Por ejemplo, en la primera estructura el llamado bloque2() sólo se podía ejecutar si la condición cond1 no había sido verdadera y si la condición cond2 sí lo había sido. En la segunda estructura encontramos exactamente las mismas condiciones, pero de una manera mucho más explícita que se va a complicando progresivamente.

Esto nos demuestra la ventaja que nos trae utilizar bloques ELIF para ayudar a simplificar y hacer más comprensible el código.

### Condicionales consecutivos

Suponga ahora que queremos cambiar nuestro programa para que al usuario no se le informe sólo el primer problema que se encuentre con su nueva contraseña, sino que se le digan de una vez todos los problemas que tenga. En este caso la estructura basada en IF-ELIF-ELSE ya no nos sirve porque necesitamos que *siempre* se verifiquen *todas* las condiciones.

Para lograr que esto pase, lo único que podemos hacer es separar las condiciones en instrucciones diferentes, asegurando así que todas sean verificadas. Es decir que, en lugar de tener un IF seguido de varios ELIF, en los cuales sólo 1 condición puede ser cierta a la vez, vamos a tener varios IF totalmente desconectados. Observe esto en la siguiente versión de nuestro programa:

nueva = input("Introduzca su nueva contraseña:")  
contrasena\_correcta = True  
mensaje = ""  
  
if longitud(nueva) <= 8:  
 contrasena\_correcta = False  
 mensaje += "La nueva contraseña debe tener al menos 8 caracteres" + "\n"  
   
if nueva == contrasena\_anterior:  
 contrasena\_correcta = False  
 mensaje += "La nueva contraseña no puede ser igual a la anterior" + "\n"  
  
if nueva.isalnum():  
 contrasena\_correcta = False  
 mensaje += "La nueva contraseña debe tener signos de puntuación" + "\n"  
   
if not tiene\_numeros(nueva):  
 contrasena\_correcta = False  
 mensaje += "La nueva contraseña debe tener al menos un número" + "\n"  
  
if not tiene\_mayusculas(nueva):  
 contrasena\_correcta = False  
 mensaje += "La nueva contraseña debe tener al menos una letra mayúscula" + "\n"  
  
if not tiene\_minúsculas(nueva):  
 contrasena\_correcta = False  
 mensaje += "La nueva contraseña debe tener al menos una letra minúscula" + "\n"  
  
if contrasena\_correcta:  
 cambiar\_contrasena(nueva)  
 mensaje = "La contraseña se cambió exitosamente"  
  
print(mensaje)

Si seguimos la traza de ejecución de este programa veremos que, después de que se ejecuten las primeras tres asignaciones, Python revisará la condición del primer IF: si es verdadera se ejecutará el cuerpo y se pasará a la siguiente instrucción; si es falsa, se pasará a la siguiente instrucción sin ejecutar nada mas. En este caso, la siguiente instrucción será el segundo IF que se evaluará sin tener en cuenta el valor de la condición del primer IF. Es decir que estamos hablando de instrucciones condicionales completamente independientes la una de la otra. Este mismo proceso se repetirá para cada una de las instrucciones siguientes.

Ahora bien, sabemos que la función cambiar\_contrasena sólo debe ejecutarse si la nueva contraseña cumple con todos los requisitos de forma que se revisaron en las condiciones anteriores. Si le asignáramos a esas condiciones los nombres P, Q, R, S, T y U, entonces el último bloque IF tendría que escribirse de la siguiente manera:

if not(P or Q or R or S or T or U):  
 cambiar\_contrasena(nueva)  
 mensaje = "La contraseña se cambió exitosamente"

Esta condición no es muy fácil de leer porque los nombres de variables no son explícitos, pero si pusiéramos nombres más claros terminaríamos con una expresión muy larga y también difícil de leer. También podríamos haber escrito las expresiones originales, pero eso significaría que cada una la evaluaríamos dos veces: una vez en el IF que la verifique y otra vez en el IF para cambiar la contraseña.

Por estas razones, en nuestro programa decidimos incluir una nueva variable llamada contrasena\_correcta: cuando inicia el programa suponemos que la nueva contraseña que ingresó el usuario cumple con todas las restricciones, así que inicializamos la variable en el valor True. Luego, en cada una de las instrucciones condicionales cambiamos el valor de la variable a falso si descubrimos que la nueva contraseña no cumple alguna de las restricciones.

Como sabemos que si la nueva contraseña no cumple con *al menos una* de las reglas entonces no debemos hacer el cambio, en el último IF basta con revisar el valor de la variable contrasena\_correcta: si tiene valor verdadero significa que no se encontró ningún problema con la nueva contraseña; si tiene valor falso significa que se encontró *al menos* un problema la nueva contraseña.

El último aspecto para revisar de este programa es el que tiene que ver con el mensaje. Cuando se inicia la ejecución tenemos un mensaje vacío. Luego, cada vez que se encuentra que no se cumple con una restricción se *concatena* un nuevo mensaje al mensaje que se tenía. Esto quiere decir, por ejemplo, que si no se cumple con ninguna de las restricciones entonces la variable mensaje tendrá la información de todas las restricciones que no se cumplieron.

Note que en el último IF, el que cambia la contraseña si no hubo ningún problema, no se concatena nada al mensaje, sino que se reemplaza por un mensaje de éxito.

**Recuerde**: un elif depende del if anterior mientras que un if es independiente de lo que haya pasado antes.

### Ejercicios

1. Modifique el programa que cambia la contraseña para que en caso de que no se pueda cambiar la contraseña se le informe al usuario no sólo las restricciones que no se cumplieron sino también las condiciones que sí se cumplieron.
2. El valor de un peaje depende del tipo de vehículo (categoría) que pase y de la cantidad de ejes que tenga: automóviles, camionetas y camperos (AUTO) deben pagar 10000; buses y busetas (BUS), 14300; camiones grandes de 2 ejes (CAMION), 16000; camiones de 3 y 4 ejes (CAMION), 28100; camiones de 5 ejes (CAMION), 37700; y camiones de 6 o más ejes (CAMION) deben pagar 41400. Adicionalmente, un vehículo podría tener un descuento especial del 15% por paso frecuente. Escriba una función que reciba el tipo de un vehículo (‘AUTO’, ‘BUS’ o ‘CAMION’) y si tiene descuento especial y calcule el valor que debe pagar en el peaje.

## Aspectos metodológicos

Como hemos visto con algunos de los ejemplos anteriores, usualmente el mismo problema puede resolverse correctamente usando estructuras diferentes. A continuación exploramos cuatro formas de resolver el mismo problema: queremos saber si un número es positivo y además es menor de 10.

### Condicionales anidados

El siguiente bloque muestra la definición de una función que resuelve el problema utilizando condicionales anidados:

def es\_positivo\_de\_un\_solo\_digito\_v1(x: int)->bool:  
 if x > 0:  
 if x < 10:  
 return True  
 else:  
 return False  
 else:  
 return False

Llamamos condicional anidado a la situación en la cual hay una instrucción condicional dentro del cuerpo de otra instrucción condicional.

En el caso de nuestro ejemplo, lo que se hizo en este ejemplo fue analizar por partes el problema. En primer lugar se decidió que, si el número es positivo, entonces se debe revisar la segunda condición. Esta revisión quedó entonces anidada en un condicional dentro del cuerpo del primer if. Por otra parte, si el número es negativo no hay necesidad de revisar la segunda condición y se puede retornar False inmediatamente.

Sin embargo, este ejemplo podría simplificarse un poco si se analizan las condiciones que llevan al único caso en que se retorna True en esta función: para que se pueda llegar a este punto es necesario que las dos condiciones sean verdaderas simultáneamente, es decir que x > 0 *y* x < 10. En todos los otros casos, la función debería retornar False.

Si llevamos eso en al código llegamos a la segunda versión de la función:

def es\_positivo\_de\_un\_solo\_digito\_v2(x: int)->bool:  
 if x > 0 and x < 10:  
 return True  
 else:  
 return False

No se puede decir ni que el uso de condicionales anidados sea una mala idea ni que armar expresiones condicionales más completas sea una buena idea. De hecho, siguiendo esta estrategia es muy fácil terminar con estructuras muy difíciles de entender y, por ende, de mantener, extender y corregir cuando tengan problemas.

Como siempre, mantener la claridad del código debería ser una prioridad para tomar decisiones con respecto a su estructura.

### Returns

A pesar de sus diferencias, los dos ejemplos anteriores coinciden en un aspecto: tan pronto encuentran la respuesta, retornan verdadero o falso y terminan la ejecución de la función. Aunque esto no está necesariamente mal, una práctica que suele hacer el código más fácil de entender y por ende mucho más mantenible, es limitar la cantidad de puntos en los cuales se utilice la instrucción return. Por ejemplo, la última versión de la función puede remplazarse por la siguiente versión remplazando los return por asignaciones a una variable y dejando sólo un return al final de la función.

def es\_positivo\_de\_un\_solo\_digito\_v3(x: int)->bool:  
 resultado = False  
 if x > 0 and x < 10:  
 resultado = True  
   
 return resultado

En esta función tan pequeña no es fácil apreciar que esta nueva versión es mucho más fácil de mantener que la versión anterior. Sin embargo, nuestra recomendación es que minimice la cantidad de puntos en los cuales usted retorne un valor desde una función.

### Expresiones vs. condicionales

En varios casos, incluyendo el que hemos estado trabajando, existen formas de remplazar las instrucciones condicionales por expresiones booleanas. Eso es lo que hemos hecho en la cuarta y última versión de la función: en lugar de calcular un resultado y luego retornarlo, acá el valor mismo de la expresión es el retorno de la función.

def es\_positivo\_de\_un\_solo\_digito\_v4(x: int)->bool:  
 return x > 0 and x < 10

Esta estrategia funciona muy bien cuando las condiciones son fáciles de leer y entender. Cuando las condiciones son mucho más complicadas probablemente sea mejor tener la estructura de las instrucciones condicionales, incluso si son anidadas, para ayudar a quien lea el código a entender su propósito.

## Caso de estudio: el mayor de 4 números

En esta sección vamos a estudiar un problema recurrente y muy importante que debe resolverse utilizando instrucciones condicionales, pero puede resolverse de maneras diferentes. Empezaremos planteándole el problema para que usted lo resuelva y luego pasaremos a discutir nuestras alternativas de solución. Esperamos que usted vaya comparando nuestras soluciones con la suya.

### Enunciado del problema

Escriba una función que reciba por parámetro cuatro números enteros y devuelva el mayor de estos. Si hay dos o más iguales y mayores, retorna cualquiera de estos. La signatura de la función debe ser:

def mayor(a: int, b: int, c:int, d:int)->int:

Escriba su solución y compárela con las que nosotros proponemos a continuación.

### Solución 1: múltiples returns

La primera solución es la solución que es más natural para este problema: si nos damos cuenta que el valor a es mayor al valor b y es mayor al valor c y es mayor al valor d, entonces tiene que se que a es el mayor valor. Algo similar se hace con los otros parámetros.

def mayor(a: int, b: int, c:int, d:int)->int:  
 if (a >= b) and (a >= c) and (a >= d):  
 return a  
 elif (b >= a) and (b >= c) and (b >= d):  
 return b  
 elif (c >= 1) and (c >= b) and (c >= d):  
 return c  
 else:  
 return d

El problema con esta solución es que requiere mucho cuidado en su elaboración: es fácil tanto intercambiar dos variables como olvidar hacer una de las comparaciones. Esta solución además tiene como limitante que agregar un nuevo elemento para calificar requiere modificaciones sobre prácticamente toda la función.

### Solución 2: un solo return

La segunda solución no dista mucho de la primera: el único cambio que se hizo fue eliminar las instrucciones return que estaban dentro de cada IF.

def mayor(a: int, b: int, c:int, d:int)->int:  
 if (a >= b) and (a >= c) and (a >= d):  
 mayor = a  
 elif (b >= a) and (b >= c) and (b >= d):  
 mayor = b  
 elif (c >= 1) and (c >= b) and (c >= d):  
 mayor = c  
 else:  
 mayor = d  
  
 return mayor

### Solución 3: aproximación algorítmica

La tercera solución es muy diferente de las anteriores y tiene una aproximación mucho más algorítmica, en la que los cuatro valores van analizándose uno por uno. Primero observemos con cuidado la nueva función hasta estar convencidos de que funciona.

def mayor(a: int, b: int, c:int, d:int)->int:  
 mayor = a  
 if (b > mayor):  
 mayor = b  
 if (c > mayor):  
 mayor = c  
 if (d > mayor):  
 mayor = d  
   
 return mayor

Al igual que en la segunda solución, tenemos una variable llamada mayor donde debería quedar la respuesta porque es la variable que vamos a retornar al final.

La primera instrucción de la función le asigna el valor de a a la variable mayor, implicando que a es el mayor valor. ¿Es esto correcto? No podemos saber si será correcto al final, pero lo que sí podemos decir con seguridad es que, si sólo hubiera un valor, entonces a sería el mayor. Esto es tan trivial que a veces resulta difícil al leerlo: el mayor en un conjunto que tiene sólo un elemento es ese único elemento.

¿Qué hace la primera instrucción condicional? En este caso hemos ampliado nuestro conjunto y ahora tenemos también a b, así que comparamos a b con a para ver cuál es el mayor. Si resulta que b era mayor, entonces dejamos en la variable mayor a b. Si no era mayor, entonces dejamos el valor que ya teníamos.

¿Qué hace la segunda instrucción condicional? Acá es donde todo se pone interesante porque agregamos a c a nuestro conjunto de valores y lo comparamos con el *mayor*. Esto quiere decir que lo comparamos con a o lo comparamos con b, pero no lo comparamos con los dos porque sería inútil: lo único que nos interesa saber es si c es mayor que el valor mayor que se había encontrado hasta el momento. Note que tampoco podemos saber si el mayor era a o b. Sólo sabemos que el valor mayor está dentro de la variable mayor. Al finalizar este condicional dentro de la variable también podría estar el valor de c, pero sólo si este fuera mayor que el mayor valor entre a y b.

¿Qué hace la tercera instrucción condicional? Lo mismo que la instrucción anterior: compara a d con el mayor valor que se hubiera encontrado entre a, b y c y, si resulta que d es mayor, entonces remplaza el valor de mayor con el valor de d.

Al llegar a la última instrucción de la función podemos estar seguros que dentro de la variable mayor se encuentra el mayor valor entre a, b, c y d, a pesar de que nunca hayamos comparado los cuatro valores entre ellos.

Esta estrategia es mucho más conveniente que la de las otras versiones de la función por varias razones. Por un lado, el código es mucho más sencillo y tiene menos comparaciones: es más fácil de leer y tiene menos posibilidades de tener errores. Por otro lado, es clara cuál era la intención del programador y, si ahora hubiera un quinto valor para comparar, sería clarísimo cómo debería modificarse la función para soportar este nuevo valor.

## Ejercicios

1. Modifique el ejercicio que retorna el número mayor para que retorne el número del parámetro en el que se encuentra el número mayor. Si hay dos o más iguales y mayores, debe retornar el número de parámetro menor (ej. si el primer y el tercer parámetro eran los mayores, debe retornar el número 1).
2. La calificación final de un estudiante en un curso depende de las calificaciones que obtenga en 3 exámenes, pero con unas reglas especiales. Si el estudiante sacó más de 4.0 sobre 5.00 en el tercer examen (el examen final), la nota en el curso será la nota del examen final. Si el estudiante menos de 2.0 en el examen final, ese examen valdrá el 50% de la nota y los otros dos exámenes valdrán el 25% cada uno. En cualquier otro caso, los exámenes pesarán lo mismo para calcular la nota final. Escriba una función que dadas las notas de los tres exámenes calcule la nota del estudiante en el curso.
3. En muchos torneos de fútbol es usual que dos equipos jueguen dos partidos para definir cuál es el mejor de ellos: uno en el estadio del primer equipo y otro en el estadio del segundo equipo. También es usual que, en caso de empate en la cantidad de partidos ganados, los goles de los equipos visitantes cuenten por dos al momento de calcular la diferencia de goles. Escriba una función para calcular el ganador entre dos equipos. La función debe recibir el nombre del primer equipo (A), el nombre del segundo equipo (B), los goles que hizo A de local, los goles que hizo B de visitante, los goles que hizo A de visitante y los goles que hizo B de local. La función debe retornar el nombre del ganador de la serie o la cadena “EMPATE” si hubo un empate entre los equipos.
4. Las denominaciones de las monedas actualmente disponibles en un país son: 20, 50, 100, 200, 500 y 1000. Escriba una función que reciba la cantidad de monedas de cada denominación que tiene una persona y el valor de un producto y le diga si es posible pagar el producto con el dinero en efectivo que tiene. Ayuda: tiene que revisar si tiene suficiente dinero y, si tiene más de lo necesario, si es posible que le den el cambio con las denominaciones de monedas que se encuentran en circulación.
5. *Picas y Fijas* es un juego en el que dos personas intentan adivinar un número de 4 dígitos que mantiene en secreto el otro jugador. En cada turno, un jugador propone un número de 4 dígitos y el otro jugador debe informar la cantidad de *picas* y la cantidad de *fijas* de ese número. Cada *pica* significa que en el número propuesto hay un dígito que también está en el número secreto, pero en una posición diferente. Cada *fija* significa que en el número propuesto hay un número que también está en el número secreto y que está en la misma posición. Por ejemplo, si el número secreto es 5678 y el número que el otro jugador propone es 6579, la respuesta sería \*2 picas y 1 fija" porque 5 y 6 están en las posiciones equivocadas y porque el 7 está en la posición correcta. El ganador del juego es el jugador que encuentre el número del otro en la menor cantidad de intentos. En este ejercicio usted debe escribir dos funciones: la primera calculará la cantidad de picas dados un número secreto entre 1000 y 9999, y un número propuesto también entre 1000 y 9999; la segunda función calculará la cantidad de fijas y recibirá también un número secreto y un número propuesto.
6. Un número primo es un número que es divisible sólo por 1 y por sí mismo. Los primeros 10 números primos son 2, 3, 5, 7, 11, 13, 17, 19, 23 y 29. Escriba una función que dado un número diga cuál es el menor de los primeros diez números primos por el que es divisible o retorne -1 si no es divisible por ninguno de esos números.
7. En una competencia sólo pueden participar estudiantes universitarios que sean menores de 23 años o que cumplan 23 años durante el año en curso. Además, pueden participar todos los estudiantes universitarios que hayan cursado menos de 2 años de estudios en la Universidad. Escriba una función que reciba el año de nacimiento de una persona y el año de entrada a la universidad y retorne un valor de verdad indicando si la persona puede participar o no.
8. En una ciudad existe una restricción de circulación para los vehículos que depende del número de la placa, del tipo de vehículo, del día de la semana y de la hora del día. Los vehículos particulares sólo tienen restricción de lunes a viernes, dependiendo del último dígito de su placa: los que terminen en un número par no podrán circular entre 6:00 y 8:30 y 15:00 y 19:30 en los días del mes que sean pares; los que terminen en un número impar no podrán circular en los mismos horarios, pero de los días del mes que sean impares. La restricción para los taxis va desde las 5:30 hasta las 21:30, de lunes a sábado: los taxis cuya placa termine en el mismo dígito en que termine el día del mes no podrán circular ese día. Escriba una función que diga si un vehículo puede circular o no dados: el tipo de vehículo (str, TAXI o PARTICULAR), la placa (str, por ejemplo DMZ042), el día del mes (int, entre 1 y 31), el día de la semana (str - Lunes, Martes, Miércoles, Jueves, Viernes, Sábado o Domingo), la hora (int, entre 0 y 23) y el minuto (int, entre 0 y 59) [^invento].

## Más allá de Python

Además de las estructuras basadas en IF-ELSE IF-ELSE, muchos lenguajes ofrecen instrucciones condicionales basados en *switch* y en *operadores ternarios*. Estos últimos son de mucha utilidad porque permiten seleccionar valores basados en condiciones, pero sin tener que utilizar demasiado espacio. Por ejemplo, este código se podría usar en Java para seleccionar una cadena de caracteres basado en el valor de la variable booleana exito:

respuesta = exito ? "sí":"no";

Lo que haría la máquina virtual de Java (JVM) al encontrar este código sería revisar si la variable exito es verdadera y, en caso afirmativo, asignarle a respuesta el valor “sí”; en caso negativo, asignarle el valor “no”.

En Python existe una estructura similar llamada *expresión condicional*. La traducción del ejemplo anterior a Python sería:

respuesta = "sí" if exito else "no"

La interpretación sería exactamente igual que en el caso de Java: se le asignará “sí” a la variable respuesta sólo si exito es verdadero y de lo contrario se le asignará “no”. Las expresiones condicionales pueden ser de mucha utilidad cuando las condiciones son tan fáciles como en el ejemplo, pero se pueden volver difíciles de leer cuando las condiciones son más elaboradas. Sin embargo, todo lo que se haría con una expresión condicional se puede hacer con un IF.

# Más sobre cadenas de caracteres

El objetivo de esta sección es explorar un poco más el tipo str y descubrir otras funcionalidades que ofrece Python para manipular cadenas de caracteres. Sin embargo, esta no pretende ser una guía exhaustiva sino un abrebocas para que el lector estudie la documentación Python y descubra muchas otras funcionalidades que están disponibles.

Hace años, cuando se empezaron a construir los primeros computadores programables, el objetivo principal que tenían era permitir hacer cálculos numéricos (trayectorias de misiles, órbitas de naves espaciales, análisis de estructuras, cálculos financieros). Por este motivo, cuando se diseñaron los primeros lenguajes de programación, su diseño estuvo fuertemente influenciado por la necesidad de soportar con facilidad la expresión de cálculos numéricos cada vez más complicados. En esos lenguajes era prácticamente imposible representar una cadena de caracteres.

Hoy en día la situación ha cambiado y la representación y manipulación de información textual es igual o más importante que los cálculos numéricos. Esto ha llevado a que cualquier lenguaje de programación moderno ofrezca sólidos tipos de datos para representar cadenas de caracteres, los cuales vienen acompañados de mecanismos que permiten manipular con facilidad estas cadenas de acuerdo con los requerimientos de cada aplicación. Por ejemplo, hoy en día es muy común que las aplicaciones tengan que incluir mecanismos de internacionalización (para traducir los mensajes a otros idiomas), varios alfabetos diferentes (cirílico, griego, kanjis japoneses y hasta *emojis*), sistemas de cifrado (para garantizar confidencialidad), y sistemas de búsqueda que encuentren términos rápidamente dentro de textos cada vez más grandes.

En esta sección vamos a continuar con el estudio del tipo str de Python y vamos a ver algunos de los mecanismos que ofrece para manipular cadenas de caracteres de forma rápida y sencilla. Sin embargo, en esta sección no termina toda la presentación de str: en el siguiente capítulo presentaremos los mecanismos adicionales disponibles que requieren del estudio previo de secuencias e instrucciones repetitivas.

## Aspectos de codificación (encoding)

Si usted está leyendo este libro es muy posible que su lengua materna sea el español, que esté acostumbrado a escribir usando el alfabeto latino y que el teclado de su computador sea un teclado ‘QWERTY’[[13]](#footnote-13). Por esto, además de necesitar los 127 caracteres básicos de ASCII, usted también necesita algunos caracteres adicionales, como 'ñ', 'Ñ' y todas las vocales con tilde en mayúsculas y minúsculas.

Si todo esto es cierto, muy posiblemente usted ha estado utilizando un conjunto de caracteres (encoding) que extiende al conjunto ASCII y que se conoce con el nombre ISO-8859-1: además de tener los 127 caracteres de ASCII, este conjunto incluye caracteres de uso frecuente en lenguajes originados principalmente en Europa occidental. Por ejemplo: á, Á, ñ, Ñ, ç, ü, ò [[14]](#footnote-14).

Ahora bien, si usted intercambia información con personas que viven en países donde ISO-8859-1 no es de utilidad (por ejemplo, Rusia, Japón, o todo el Sudeste Asiático) tendrá que utilizar un conjunto de caracteres mucho más grande, como el ofrecido por el estándar UNICODE que describe muchos más caracteres incluyendo los cada vez más populares *emojis* - 😀. Sin embargo, si usted quiere usar caracteres definidos en UNICODE, los archivos o mensajes que usted escriba tendrán que utilizar bien sea el encoding **UTF-8** o el encoding **UTF-16** [[15]](#footnote-15).

Si usted puede ver los siguientes tres caracteres (un kanji, un emoji y un caracter griego) significa que su navegador o el programa que está utilizando para leer este texto es compatible con UNICODE y con UTF-8:

* Kanji: 字
* Emoji: 😜
* Caracter griego: Δ.

Con esta brevísima explicación no esperamos que usted sea ya un experto en sistemas de codificación de caracteres, pero sí que se dé cuenta que es un asunto más complicado de lo que parece a simple vista. Mientras que sus programas se limiten a usar caracteres que son normales en su idioma, probablemente no tendrá ningún problema. Pero si quiere que sus programas sean capaces de manejar caracteres de otros alfabetos, tendrá que ponerles atención a los asuntos de codificación si no quiere perder información o ver caracteres remplazados por marcas desconocidas.

La buena noticia es que, por defecto, Python soporta cadenas UNICODE. Así que, si está teniendo problemas con caracteres extraños dentro de sus cadenas, probablemente el problema no sea de Python sino de algo más de lo que usted está haciendo.

Otra cosa con la que usted tiene que tener cuidado es la codificación que utilice para los archivos .py en los que escriba sus programas. La recomendación que le hacemos es que se asegure de que los archivos se guarden con la codificación UTF-8. En Spyder esto se logra poniendo el siguiente comentario al principio de cada archivo:

# -\*- coding: utf-8 -\*-

Si lo hizo correctamente, en la esquina inferior derecha del ambiente debería ver una etiqueta que diga *Encoding: UTF-8*. Si en lugar de esto dice *Encoding: ASCII*, entonces cualquier caracter que usted escriba en su programa y que no sea parte de ASCII (por ejemplo ‘á’) se convertirá en basura cuando usted cierre el archivo y lo vuelva a abrir.

Finalmente le hacemos una recomendación con respecto al uso de caracteres *no ASCII* dentro de sus programas: está bien usar estos caracteres dentro de la documentación y comentarios, pero evite usarlos en los identificadores de variables, parámetros y nombres de funciones. Por ejemplo, en lugar de llamar a una función calcular\_año use calcular\_anho, y en lugar de usar la variable máximo use la variable maximo. Esta no es una regla escrita en piedra que usted tenga que seguir. Es sólo un consejo para evitar que se tope con problemas sencillos, pero a veces difíciles de diagnosticar.

## Caracteres de control

Además de los caracteres como los que se describieron en la sección anterior, una cadena de caracteres (str) también puede tener caracteres que son llamados caracteres de control. Estos caracteres se pueden usar como cualquier otro dentro de una cadena.

A continuación, describimos los más importantes y que usted podría necesitar.

|  |  |  |  |
| --- | --- | --- | --- |
| Caracter | Significado | Ejemplo de uso | Resultado |
|  | Cambio de línea | “Hola!” | Hola Mundo! |
| Tabulación | “Hola” | Hola        Mundo |  |
| \ | Barra invertida | “C:\usuarios\invitado” | C: |
| ' | Comilla sencilla (2) | “It's me!” | It’s me! |
| " | Comilla doble | “A so called "expert"” | A so called “expert” |

Como es posible que con un ejemplo tan breve no se aprecie la importancia de la tabulación, a continuación presentamos un ejemplo un poco más complejo. El uso del caracter de tabulación indica que se quiere dejar espacio hasta la siguiente tabulación, con el objetivo de armar columnas más o menos definidas. Una tabulación típicamente equivale al espacio de 8 caracteres.

print("a\tb\tc" + "\n" + "100\t200\t300")

El resultado de ejecutar el programa anterior es el siguiente:

a b c  
100 200 300

## Funciones y operadores sobre cadenas de caracteres

En la sección anterior estudiamos los operadores relacionales y vimos que pueden aplicarse a diferentes tipos de datos. En particular, vimos que los operadores <, >, <=, >= hacen comparaciones lexicográficas. De esta manera sabemos que las siguientes expresiones son todas verdaderas:

* "AB" < "BC
* "AB" < "ab"
* "234" < "345"
* "!450" < "345"

Los otros operadores relacionales que podemos aplicar a las cadenas de caracteres son “==” y “!=”, los cuales comparan las cadenas caracter por caracter. De esta manera, las siguientes expresiones son todas falsas:

* "ABC" == "abc"
* "ABC" == "ABC!"
* "3" == "1+2"
* "Hola, Mundo!" != "Hola, Mundo!"

Dos nuevos operadores que también se pueden aplicar sobre cadenas de caracteres son in y not in. El primero permite revisar si una cadena está incluida en otra, mientras que el segundo hace exactamente lo contrario. Veamos un ejemplo de expresiones **verdaderas** basadas en estos operadores:

* "Mundo" in "Hola, Mundo!"
* "mundo" not in "Hola, Mundo!"
* "!" in "Hola, Mundo!"
* '?' not in "Hola, Mundo!"
* "Hola, Mundo!" in "Hola, Mundo!"

Como vemos, estos operadores hacen comparaciones que distinguen entre mayúsculas y minúsculas. Esto debe tenerse en cuenta para evitar tener falsos negativos y falsos positivos.

Finalmente, Python nos ofrece la función len que podemos utilizar para conocer el tamaño de una cadena.

* len("Hola, Mundo!")
* len("Hola,\nMundo!")
* len("")
* len(" ")
* len("\t")

El resultado de las primeras dos invocaciones es el mismo (12). Esto porque el caracter de control \n es un solo caracter aunque se represente con dos.

El resultado de la tercera invocación es 0, porque es una cadena vacía (no tiene ningún caracter).

El resultado de las últimas invocaciones es 1 porque se trata de cadenas con sólo un caracter: la cuarta tiene sólo un espacio mientras que la quinta sólo tiene una tabulación.

## Métodos sobre cadenas de caracteres

### Métodos versus funciones

Además de todo lo que ya hemos estudiado, el tipo str también ofrece una serie de funciones que pueden utilizarse para transformar cadenas de caracteres. Debido a la forma particular en la que estas funciones están declaradas dentro del módulo str, estas funciones se conocen usualmente como métodos y se pueden invocar de una forma diferente a como lo haríamos con funciones como las que ya conocíamos [[16]](#footnote-16).

Para ilustrar este punto, estudiemos la función lower que está definida dentro del contexto del módulo str. En este módulo es donde se define el tipo que hemos estado usando para las cadenas de caracteres. Usando la función help podemos consultar la documentación de esta función:

>>> help (str.lower)  
Help on method\_descriptor:  
  
lower(self, /)  
 Return a copy of the string converted to lowercase.

Lo que esto nos dice es que si invocamos esta función usando como parámetro una cadena de caracteres vamos a recibir como respuesta *una copia* de la cadena en la cual las letras se habrán pasado todas a minúsculas. Veamos una prueba que nos ayuda a comprobar que esta interpretación es correcta:

>>> str.lower('ABcdEg')  
'abcdeg'

Sin embargo, la documentación de la función también nos dice que esta función es un método, así que podemos invocarla de una forma un poco diferente que en la mayoría de los casos resulta más conveniente:

>>> 'ABcdEg'.lower()  
'abcdeg'

Observe como el parámetro de la función pasó a ocupar el lugar frente al caracter ., dejando a la función sin parámetro.

La pregunta que cabe hacer en este punto es ¿cómo sabe Python que la función está definida dentro del módulo str? La respuesta está en que Python primero va a verificar el tipo del valor 'ABcdEg' y como el tipo es str significa que las funciones que se llamen sobre este valor tienen que estar implementadas en el módulo str.

Hagamos ahora el mismo ejercicio con la función llamada str.zfill:

>>> help(str.zfill)  
Help on method\_descriptor:  
  
zfill(self, width, /)  
 Pad a numeric string with zeros on the left, to fill a field of the given width.  
  
 The string is never truncated.

La documentación nos dice que esta función le va a agregar ceros a la izquierda a una cadena, hasta lograr una cadena de la longitud indicada. La documentación también nos dice que esta función requiere dos parámetros (*self* y *width*). Como es un método, esta función podría llamarse sobre una cadena de la siguiente manera:

>>> "abc".zfill(10)  
'0000000abc'

Si quisiéramos invocarla de la manera *normal*, tendríamos que convertir la cadena en el primer parámetro e indicar en qué módulo está definida la función. El resultado sería el siguiente:

>>> str.zfill("abc", 10)  
'0000000abc'

En resumen: el módulo str ofrece unas funciones llamadas métodos que implementan una gran cantidad de funcionalidades interesantes para trabajar con cadenas de caracteres. Estas funciones se pueden invocar de dos formas: **como funciones**, usando un llamado de la forma str.funcion(cadena, ...otros parámetros ...), o **como métodos**, usando un llamado de la forma cadena.metodo(... otros parámetros ...). De acá en adelante nosotros vamos a usar estas funciones de la segunda forma (como métodos).

### Inmutabilidad

Antes de presentar los métodos más utilizados de str es muy importante hablar de la *inmutabilidad* de las cadenas de caracteres en Python. Esto significa que cuando Python construye una cadena no puede modificarla para cambiar su contenido. Si queremos hacerle algún cambio a la cadena, lo único que Python puede hacer es construir una nueva cadena con los cambios requeridos.

Observemos un ejemplo utilizando el método lower que ya estudiamos y los operadores is (comparación de identidad) y == (comparación de igualdad).

original = "texto original"  
minusculas = original.lower()  
print("Son iguales:", original == minusculas)  
print("Son el mismo:", original is minusculas)

El resultado de ejecutar este pequeño programa es el siguiente:

Son iguales: True  
Son el mismo: False

Lo que esto nos muestra es que, como dice la documentación del método, lower() retorna una *copia* de la cadena en la cual se hayan hecho las modificaciones para que todas las letras sean minúsculas. En el caso de nuestro ejemplo no hubo ninguna modificación (la comparación de igualdad es exitosa) y de todas maneras el método creó una copia de la cadena original.

**Recuerde:** Las cadenas de caracteres en Python son *inmutables*. Todas las operaciones que se realicen sobre una cadena siempre van a producir una **cadena nueva**.

### Métodos de str

Los siguientes son algunos de los métodos de str que podría necesitar con más frecuencia cuando trabaje con cadenas de caracteres.

#### Mayúsculas y minúsculas

|  |  |  |  |
| --- | --- | --- | --- |
| Nombre | Descripción | Ejemplo | Resultado |
| lower | Remplaza todas las mayúsculas por minúsculas | “Hola, Mundo!”.lower() | ‘hola, mundo!’ |
| upper | Remplaza todas las minúsculas por mayúsculas | “Hola, Mundo!”. upper() | ‘HOLA, MUNDO!’ |
| title | Le pone mayúscula inicial a todas las palabras de la cadena | “Hola, Mundo!”. title() | ‘Hola, Mundo!’ |
| swapcase | Intercambia mayúsculas por minúsculas y viceversa | “Hola, Mundo!”. swapcase() | ‘hOLA, mUNDO!’ |

#### Análisis de cadenas

|  |  |  |  |
| --- | --- | --- | --- |
| Nombre | Descripción | Ejemplo | Resultado |
| find | Busca la primera posición en la que aparezca la cadena buscada. Si no la encuentra retorna -1. | “Hola, Mundo!”.find(‘o’) | 1 |
| rfind | Busca la primera posición en la que aparezca la cadena buscada, empezando por la derecha. Si no la encuentra retorna -1. | “Hola, Mundo!”.find(‘o’) | 10 |
| count | Cuenta cuántas veces está la cadena indicada en otra cadena. Si no la encuentra se produce un error. | “Hola, Mundo!”.find(‘o’) | 2 |
| isnumeric | Revisa si todos los caracteres de una cadena son números | “Hola, Mundo!”.isnumeric() | False |

#### Manipulación de cadenas

|  |  |  |  |
| --- | --- | --- | --- |
| Nombre | Descripción | Ejemplo | Resultado |
| replace | Remplaza algunos elementos de una cadena con otros elementos que llegan como parámetro. El reemplazo puede ser la cadena vacía para eliminar los elementos buscados. | “Hola, Mundo!”.replace(‘o’, ‘67’) | ‘H67la, Mund67!’ |
| strip | Elimina espacios y cambios de línea al final de una cadena de caracteres. No elimina ningún elemento que no se encuentre al final de la cadena. | “Hola, Mundo! ”.strip() | ‘Hola, Mundo!’ |
| ljust | Amplía la cadena hasta el ancho indicado, alinea el contenido a la izquierda y llena el espacio vacío con espacios. | “Hola, Mundo!”.ljust(15) | ‘Hola, Mundo!    ’ |
| rjust | Amplía la cadena hasta el ancho indicado, alinea el contenido a la derecha y llena el espacio vacío con espacios | “Hola, Mundo!”.rjust(15) | ‘   Hola, Mundo!’ |
| center | Amplía la cadena hasta el ancho indicado y centra el contenido. | “Hola, Mundo!”.center(15) | ‘  Hola, Mundo! ’ |
| zfill | Amplía la cadena hasta el ancho indicado, alinea el contenido a la derecha y llena el espacio vacío con caracteres 0. | “Hola, Mundo!”.zfill(15) | ‘000Hola, Mundo!’ |

#### El resto de métodos

El módulo str ofrece una gran cantidad de métodos adicionales que no vamos a cubrir acá pero que sí están bien descritos en la documentación de Python: https://docs.python.org/3.6/library/stdtypes.html#string-methods

Le recomendamos estudiar esa documentación y practicar con todos estos métodos para aprender a utilizarlos y agregarlos a su caja de herramientas de programación.

## Format

Las últimas funcionalidades de las cadenas de caracteres que vamos a estudiar tienen que ver con el método format. Este método es muy poderoso y permite utilizar repetidamente una cadena como plantilla para luego remplazar algunos fragmentos por valor particulares.

Supongamos que tenemos que hacer un programa que sea capaz de generar las siguientes cadenas:

Un Lamborghini Aventador del 2016 con motor de 6.5 litros es capaz de pasar de 0 a 100 kph en 2.80 segundos  
Un Ferrari Enzo del 2002 con motor de 6.0 litros es capaz de pasar de 0 a 100 kph en 3.60 segundos  
Un Bugatti Veyron 16.4 del 2010 con motor de 8.0 litros es capaz de pasar de 0 a 100 kph en 2.50 segundos  
Un Porsche 911 GT3 del 2011 con motor de 4.0 litros es capaz de pasar de 0 a 100 kph en 3.80 segundos  
Un McLaren P1 del 2013 con motor de 3.8 litros es capaz de pasar de 0 a 100 kph en 2.80 segundos  
Un Pagani Huayra BC del 2017 con motor de 6.0 litros es capaz de pasar de 0 a 100 kph en 3.30 segundos

Todas estas cadenas tienen la misma estructura que incluye la marca del carro, el nombre del modelo, el año de lanzamiento, el tamaño del motor en litros usando una cifra decimal y el tiempo que necesita para pasar de 0 a 100 kilómetros por hora expresado en segundos con dos cifras decimales.

Estas cadenas se podrían haber armado usando concatenaciones. Sin embargo, el método format ofrece características para estructurar las cadenas que son muy difíciles de replicar usando sólo concatenaciones.

Tomemos como ejemplo la plantilla utilizada para generar los mensajes anteriores:

plantilla = "Un {0} {1} del {2:d} con motor de {3:.1f} litros es capaz de pasar de 0 a 100 kph en {4:.2f} segundos"

En primer lugar, la plantilla es una cadena de caracteres que tiene marcas rodeadas por los caracteres { y } en los campos donde se tendrán que insertar valores. Aunque no es obligatorio, es recomendable numerar esos campos empezando desde el 0 y llegando, en este caso, hasta el 4.

Además del número, cada campo también puede tener una especificación de su formato, la cual viene a continuación del caracter :. En nuestro caso, los campos 0 y 1 no tienen ningún formato, así que lo que metamos en esas posiciones se remplazará sin ningún cambio. Para el campo 2 sólo usamos el caracter d para indicar que se trataba de un número entero. Para los campos 3 y 4 especificamos que se trataban de números decimales y además dijimos que nos interesaba tener 1 dígito decimal en el primer caso y 2 dígitos decimales en el segundo. Esto quedó especificado usando los formatos .1f y .2f respectivamente.

Para usar una plantilla se tiene que invocar el método format utilizando como parámetros los valores que queremos insertar en los campos. En nuestro caso, invocamos la función format con 5 parámetros:

mensaje = plantilla.format(marca, modelo, anho, cc, sec)

Acá puede verse la importancia de numerar los campos: el valor marca quedará en el campo marcado con el 0, el valor modelo en el campo marcado con el 1 y así sucesivamente. Esto significa que el orden en el que aparecen los campos en la plantilla no necesariamente es el mismo orden en que se deben pasar los argumentos. También significa que, si hubiéramos marcado dos campos con el mismo número, el valor correspondiente habría aparecido dos veces en el mensaje.

Estudiemos ahora otros mensajes que tienen un poco más de complicaciones que no son fáciles de solucionar sólo usando concatenación de cadenas:

100 metros 5.3 s @ 126 kph  
500 metros 13.6 s @ 211 kph  
1000 metros 22.0 s @ 249 kph

Estos nuevos mensajes tienen una característica que no tenían los mensajes anteriores: los campos no están completamente llenos y están alineados a la derecha (columnas 2 y 3) o a la izquierda (columna 1). Analicemos entonces la plantilla que se utilizó en este caso

plantilla2 = "{0:<14}{1:>4.1f} s @ {2:>3d} kph"

El formato para el primer campo dice <14, lo cual se interpreta como que se debe alinear el contenido a la izquierda y que debe tener 14 caracteres de ancho en total. Si el contenido del campo es más corto que esto, se agregarán tantos espacios como haga falta *a la derecha del contenido*.

El formato para el segundo campo dice >4.1f. Como ya sabemos, 1.f significa que se tratará de un número con exactamente un decimal. La otra parte, >4 indica que el número debe alinearse a la derecha y que debe ocupar, en total, el espacio de 4 caracteres. En este caso como el número está alineado a la derecha los caracteres faltantes se agregan a la *izquierda del contenido*.

El formato para el tercer campo es muy similar: se espera un número entero que terminará alineado a la derecha en una columna de 3 caracteres de ancho.

Finalmente, veamos cómo se generaron las 3 cadenas invocando el método format sobre la plantilla:

plantilla2.format('100 metros', 5.3, 126)  
plantilla2.format('500 metros', 13.6, 211)  
plantilla2.format('1000 metros', 22, 249)

En resumen, cuando se construye una plantilla para usar con el método format se debe:

1. Marcar los campos que se van a remplazar, numerándolos de cero en adelante.
2. Para cada campo que se quiera alinear, indicar si la alineación debe ser a la izquierda (<), derecha (>) o el centro (^) y el ancho que debe tener el campo. También se puede especificar el caracter que se debe utilizar para los espacios vacíos (ver documentación).
3. Especificar el formato, especialmente si se trata de números decimales.

Se puede conseguir mucha más información sobre la especificación de plantillas usando el comando help('FORMATTING'). La documentación completa también está disponible en <https://docs.python.org/3.7/library/string.html#format-string-syntax>

## Ejercicios

1. Escriba una función que reciba una cadena de caracteres y una letra. Su función debe retornar la misma cadena que recibió, pero cambiando todas las vocales por la letra que también llegó por parámetro. Por ejemplo, si la cadena original era “Hola, Mundo!” y la letra entregada fue ‘I’, el resultado debería ser “HIlI, MIndI!”.
2. Escriba una función que reciba dos cadenas de caracteres. La función debe retornar 1 si las cadenas son idénticas, 2 si las cadenas sólo se diferencian por las mayúsculas y minúsculas, o 0 de lo contrario.
3. Escriba una función que reciba dos cadenas de caracteres que sólo van a contener letras mayúsculas y minúsculas. La función debe retornar -1 si en un diccionario la primera cadena debería ir antes que la segunda, debe retornar 1 si la segunda cadena debe ir antes que la primera, o 0 si las dos cadenas son la misma (ignorando mayúsculas y minúsculas).
4. Escriba una función que reciba una cadena de caracteres y cuente las palabras que aparecen en la cadena. Usted puede suponer que la cadena tendrá letras (mayúsculas y minúsculas) y espacios, pero no tendrá ningún signo de puntuación ni espacios seguidos.
5. Tres equipos de futbol participaron en un pequeño torneo en que jugaron entre ellos 3 partidos. Escriba una función que reciba el nombre de los tres equipos y los marcadores de los tres partidos, y que retorne una tabla con las posiciones de los equipos al finalizar el torneo. La función recibirá entonces 9 parámetros: primero los nombres de los tres equipos y luego 6 enteros con los marcadores de los 3 partidos. Cada partido ganado entregaba 3 puntos y cada partido empatado entregaba 1 punto. La tabla con el resultado del torneo tiene que ser una cadena de caracteres con la información organizada en columnas bien alineadas. Las columnas deben estar organizadas de la siguiente forma:

* posición,
* nombre del equipo,
* puntos obtenidos,
* partidos jugados,
* partidos ganados,
* partidos empatados,
* partidos perdidos,
* goles a favor,
* goles en contra,
* diferencia de goles

1. Escriba una función que, dada la altura de un edificio, retorne una cadena como en el siguiente ejemplo: *“Un objeto que cae de un edificio de 30 metros tarda 2.47 segundos en llegar al piso y alcanza una velocidad de 24.25 metros por segundo.”* Su programa debe usar las funciones de formato de cadenas.

**Ayuda:** El tiempo que tarda la caída es igual a la raíz cuadrada de dos veces la altura sobre la aceleración de la gravedad (9.8 m/s2). La velocidad que alcanza el objeto es igual al tiempo de la caída multiplicado por la aceleración de la gravedad.

1. Escriba una función que dados el nombre de un país, la cantidad de habitantes en millones y el Producto Interno Bruto en millones de USD, retorne una cadena como en el siguiente ejemplo: “Colombia……………..|\*\*\*\*\*\*\*45 millones\*\*\*\*\*\*\*|    336599 USD Million”. La primera columna debe estar alineada a la izquierda, debe tener 25 caracteres y ocupar los espacios vacíos con .; la segunda columna debe estar centrada, tener 25 caracteres y ocupar los espacios vacíos con \*; la tercera columna debe estar alineada a la derecha, tener 10 caracteres más el espacio ocupado por USD Million y debe ocupar los espacios vacíos con espacios.

## Más allá de Python

Las cadenas de caracteres son de muchísima importancia en prácticamente todos los lenguajes de programación, pero hay mucha variedad en los mecanismos para representarlos y manipularlos. Por ejemplo, en lenguajes como C y C++ el uso de cadenas de caracteres no es tan sencillo como en lenguajes más modernos puesto que se construyen sobre tipos más sencillos y requieren de particular cuidado para evitar problemas en la manipulación de la memoria. En lenguajes como Java y Python, estas limitaciones ya no existen y las cadenas se pueden usar con mucha más facilidad, aunque sigue siendo necesario poner mucha atención a ciertos detalles. Por ejemplo, en Java las cadenas de caracteres son objetos que aparentemente son como cualquier objeto otro, pero reciben un tratamiento *preferencial*: hay expresiones en el lenguaje específicas para trabajar con cadenas y el compilador procesa de forma diferente los objetos de tipo String que se construyan explícitamente dentro de un programa.

Otro aspecto común en varios lenguajes es el que tiene que ver con la inmutabilidad de las cadenas de caracteres. Así como en Python, en Java y en otros lenguajes las cadenas de caracteres son inmutables: una vez se construyen no pueden cambiar su contenido. Esto se hace por motivos de eficiencia tanto en los cálculos como en el uso de memoria, pero podría llevar a errores si no se tiene en cuenta esta condición.

En Python las cadenas de caracteres tienen otra propiedad muy interesante y es que son *secuencias*. Es decir que pueden manipularse igual que como se manipularían listas, diccionarios y otras estructuras de datos básicas del lenguaje. Este aspecto de las cadenas de caracteres se estudiará en el siguiente capítulo.

# Módulos de la librería estándar

El objetivo de esta sección es presentar un par de módulos para ilustrar el poder de las funcionalidades que ya están disponibles en la librería estándar de Python.

Afortunadamente vivimos en una época en la cual no tenemos que preocuparnos por programar absolutamente todo cada vez que queremos construir un nuevo programa. Por ejemplo, cuando en 1992 la compañía idSoftware empezó a trabajar en Wolfenstein 3D, tuvieron que construir desde cero todas las funcionalidades para representar espacios tridimensionales en la pantalla. Hoy en día, no sólo hay infinidad de librerías que hacen las mismas funcionalidades, sino que algunas de esas funcionalidades están implementadas dentro de las tarjetas de video para que su uso sea aún más fácil y su ejecución sea más rápida.

Por otra parte, esta realidad hace que sea cada vez más importante tener la curiosidad de buscar librerías que puedan ayudarnos con nuestro trabajo y la habilidad para aprender a usarlas rápidamente.

En esta sección enfrentamos esta realidad introduciendo dos módulos que hacen parte de la librería estándar de Python. Es decir, estos dos módulos hacen parte de la librería que debería acompañar a cualquier distribución de Python y que siempre deberíamos tener disponible. Esta librería incluye más de un centenar de módulos que cubren aspectos como procesamiento de texto, manipulación de fechas y calendarios, compresión de archivos, criptografía, comunicación por Internet e interacción con el sistema operativo.

## El módulo math

El primer módulo que vamos a introducir es el módulo math, cuya documentación actual se puede encontrar en <https://docs.python.org/3.7/library/math.html>.

El módulo math define funciones que permiten hacer con facilidad importantes operaciones matemáticas y definen también unas constantes de uso frecuente.

La manera de importar todo lo que ofrece este módulo es a través de la instrucción import math.

### Funciones

Las siguientes son algunas de las funciones que define el módulo y cuyo conocimiento podría ser de mucha utilidad.

* gcd(x, y): función para calcular el máximo común divisor de dos números (Gratest Common Denominator).
* log(x, base): función para calcular el logaritmo de un número con respecto a una base.
* log2(x): función para calcular el logaritmo de un número en base 2.
* sqrt: función para calcular la raíz cuadrada de un número.
* sin, cos, tan: funciones para calcular el seno, coseno y tangente de un ángulo medido en radianes.
* degrees: función para convertir un ángulo en radianes a un ángulo medido en grados.
* radians: función para convertir un ángulo medido en grados a un ángulo medido en radianes.

### Constantes

Este módulo también define unas constantes que son de utilidad tanto para hacer otros cálculos como para detectar problemas con cálculos previos. Estos valores son:

* PI
* E (número de Euler)
* INF, el valor que utiliza Python para representar el infinito.
* NaN, el valor que utiliza Python para representar un número indefinido (NaN significa Not A Number).

>>> print("El valor de pi:", math.pi)  
El valor de pi: 3.141592653589793  
>>> print("El valor de e:", math.e)  
El valor de e: 2.718281828459045  
>>> print("El valor de infinito:", math.inf)  
El valor de infinito: inf  
>>> print("El valor de un número indefinido:", math.nan)  
El valor de un número indefinido: nan

## El módulo random

El segundo módulo que vamos a introducir es el módulo random, cuya documentación actual se puede encontrar en <https://docs.python.org/3.7/library/random.html>.

El módulo random define funciones que generan números aleatorios de acuerdo con diferentes reglas. Por ejemplo, este módulo ofrece mecanismos para generar valores continuos, valores discretos y también valores que se ajusten a las principales distribuciones aleatorias.

La manera de importar todo lo que ofrece este módulo es a través de la instrucción import random.

### Valores continuos

Una variable aleatoria continua es una variable que puede asumir cualquiera de los valores dentro de un rango determinado, con una probabilidad que depende de la distribución asociada a la variable.

Dentro del módulo random, la función también llamada random es tal vez la más utilizada porque permite generar valores uniformemente distribuidos entre 0 y 1. Es decir, cada vez que se invoque la función random.random() se obtendrá un número entre 0 y 1, escogido de forma completamente aleatoria. Note que 0 es un valor posible, pero 1 está por fuera del intervalo considerado.

La gran ventaja que tiene esta distribución es que multiplicar el resultado de la función por un valor ‘x’ hace que se encuentren valores uniformemente distribuidos entre 0 y ‘x’.

Por ejemplo, si quisiéramos generar un valor aleatorio entre 0 y 7, podríamos ejecutar el siguiente código:

import random  
  
valor = random.random() \* 7

Un efecto similar se puede lograr usando la función uniform que recibe dos parámetros ‘a’ y ‘b’ y genera un número aleatorio en el intervalo [a, b). Note que llamar random.uniform(a, b) es equivalente a invocar random.random()\*(b-a) + a.

### Valores discretos

Una variable aleatoria discreta toma sólo valores discretos dentro de un rango determinado, con una probabilidad que depende de la distribución asociada a la variable. A diferencia de las variables continuas, cuando las variables son discretas los posibles valores que pueden asumir son enumerables.

La principal función para generar variables aleatorias discretas se llama randint y sirve para generar valores enteros entre dos números ‘a’ y ‘b’. En el siguiente programa se usa esta función para simular el lanzamiento de un dado:

import random  
  
lanzamiento = random.randint(1, 6)

Una función relacionada es randrange, que genera valores enteros desde un número inicial (start), hasta un número final (stop), con un cierto intervalo (step). Por ejemplo, si queremos un número múltiplo de 3 entre 6 y 30 podemos usar la siguiente invocación:

numero = random.randrange(6, 30, 3)

Note que esta función puede generar el número ‘start’ pero nunca generará el valor ‘stop’.

### Variables aleatorias

Finalmente, el módulo random incluye funciones para generar valores siguiendo la distribución triangular, Beta, exponencial, Gamma, Normal, y Pareto, entre otras.

Como esta no pretende ser una revisión exhaustiva sólo revisaremos la función random.normalvariate, que genera números distribuidos de acuerdo a una distribución normal.

>>> help(random.normalvariate)  
Help on method normalvariate in module random:  
  
normalvariate(mu, sigma) method of random.Random instance  
 Normal distribution.  
  
 mu is the mean, and sigma is the standard deviation.

Esta función requiere de un parámetro mu (el valor promedio de los valores en la distribución) y de un parámetro sigma (la desviación estándar de los valores) para generar valores que se distribuyan de forma normal de acuerdo con los parámetros. Los siguientes son 10 valores generados con esta función usando una media de 10 y una desviación estándar de 1.5:

# random.normalvariate(10,1.5)  
0 - 9.74230603318132  
1 - 9.765339949262536  
2 - 10.309760658154236  
3 - 10.00652736167399  
4 - 8.828709896119436  
5 - 9.105408757081975  
6 - 8.28137647679426  
7 - 9.898607684096598  
8 - 7.545894557163404  
9 - 10.83177690308728

## Ejercicios

1. Usando la función random.normalvariate genere 15 números aleatorios con media 3.8 y desviación estándar de 1. Calcule ahora usted la media de los números generados y la desviación estándar. ¿Qué tan lejos están de la media y la desviación planeada? Ejecute su programa y observe cómo cambian los resultados con cada ejecución.

## Más allá de Python

Así como Python define una librería estándar con módulos para las tareas más comunes (¡y muchas tareas no tan comunes!), la mayoría de lenguajes de programación ofrecen su propia librería estándar que debería estar disponible para todos los que usen el lenguaje. A juzgar por las discusiones y procesos legales de los últimos años sobre la propiedad y la disponibilidad de las librerías estándar de Java, se podría creer que las librerías son incluso más importantes que el lenguaje mismo.

Muchas veces, aprender a utilizar efectivamente las funcionalidades disponibles en las librerías estándar es más difícil y toma más tiempo que aprender a usar la sintaxis misma de un lenguaje. Por ejemplo, un programador experimentado debería ser capaz de dominar en un día la sintaxis del lenguaje de programación SmallTalk[[17]](#footnote-17) , pero seguramente le tomaría mucho más tiempo dominar las librerías estándar que son absolutamente imprescindibles para construir incluso programas sencillos con el lenguaje. Algo similar le pasa a los programadores que pasan de Java a C# y viceversa: hay muchísimos conceptos comunes pero la principal dificultad en el proceso de aprendizaje es aprender a utilizar las librerías principales.

1. Para mantener las tablas relativamente pequeñas, usaremos T para valores verdaderos y F para valores falsos. [↑](#footnote-ref-1)
2. En el álgebra elemental tanto la adición como la multiplicación son conmutativas. [↑](#footnote-ref-2)
3. En el álgebra elemental tanto la adición como la multiplicación son asociativas. Es decir que **a + (b + c)** es igual a **(a + b) + c**. [↑](#footnote-ref-3)
4. El álgebra elemental, la multiplicación distribuye sobre la adición, pero no al contario. Es decir que **a×(b+c)=a×b + a×c**, pero la ecuación no se mantiene si se intercambian las operaciones. [↑](#footnote-ref-4)
5. Esto es equivalente a multiplicar por 1 o a sumar 0. [↑](#footnote-ref-5)
6. Esto es equivalente a multiplicar por 0. No existe algo similar para la adición. [↑](#footnote-ref-6)
7. Acá hemos incluido sólo algunos ejemplos de las conversiones que se pueden hacer. Hizo falta incluir ejemplos basados en números complejos, valores nulos (None), listas, diccionarios, conjuntos y en general todos los tipos de secuencias y colecciones. [↑](#footnote-ref-7)
8. Más adelante en este libro veremos muchos ejemplos de funciones y valores que toman tiempos largos para ser calculados. [↑](#footnote-ref-8)
9. Para entender bien el orden lexicográfico se debe revisar la tabla ASCII que se discutió en el nivel 1 y entender que lo que el resultado de la comparación es el resultado de comparar los números de cada caracter en la tabla ASCII. De esta forma, el caracter ‘k’ (#107) va después del ‘R’ (#82), que va después del ‘;’ (#59), que va después del ‘4’ (#52), que va después del ‘&’ (#38). [↑](#footnote-ref-9)
10. Este problema no es exclusivo de Python. Casi todos los lenguajes sacrifican precisión en los cálculos para ganar un poco de eficiencia y mejorar el uso de la memoria. [↑](#footnote-ref-10)
11. Si usted ha programado antes o si ya leyó las siguientes secciones podría pensar que es necesario usar condicionales para resolverlos. La realidad es que todos estos ejercicios pueden resolverse usando sólo lo que se estudió en este capítulo. [↑](#footnote-ref-11)
12. En realidad, también puede haber cosas que es poco usual encontrar dentro de un condicional, como declaración de funciones y llamados para importar módulos. No es muy recomendable usar estas capacidades a menos que haya motivos muy bien justificados para hacerlo. [↑](#footnote-ref-12)
13. QUERTY hace referencia a las primeras 5 letras de la fila de letras superior del teclado. [↑](#footnote-ref-13)
14. Si usted no puede ver alguno de estos caracteres, significa que el encoding ISO-8859-1 no está soportado o no está activo en su navegador, en el visor que esté utilizando para leer el texto, o en su computador. [↑](#footnote-ref-14)
15. Esta última parte puede ser un poco complicada de entender, pero no es crítico que entienda todos los detalles: UNICODE define cuáles son los caracteres, cómo deberían verse y qué número le corresponde a cada uno. UTF-8 define cómo se deben representar los caracteres UNICODE usando números binarios de 8, 16, 24 o 32 bits. UTF-16 define cómo representar esos caracteres usando 16 o 32 bits. [↑](#footnote-ref-15)
16. Siendo más precisos, str es en Python una *clase* y las clases definen un espacio de nombres dentro de los que están definidas un conjunto de funciones que son llamados métodos. Como se anunció en la introducción, en este libro no se tratará el tema de la programación orientada a objetos, así que no ahondaremos en el tema de la definición de clases y métodos. Sin embargo, para construir efectivamente programas Python, es necesario saber que existen librerías básicas basadas en clases, como el caso de str, y es necesario ser capaz de utilizar los métodos implementados en esas librerías. [↑](#footnote-ref-16)
17. SmallTalk es un lenguaje de programación muy poderoso, pero con una sintaxis extremadamente pequeña. Por ejemplo, en el lenguaje mismo no existen instrucciones condicionales, sino que estas están implementadas en las librerías estándar. [↑](#footnote-ref-17)