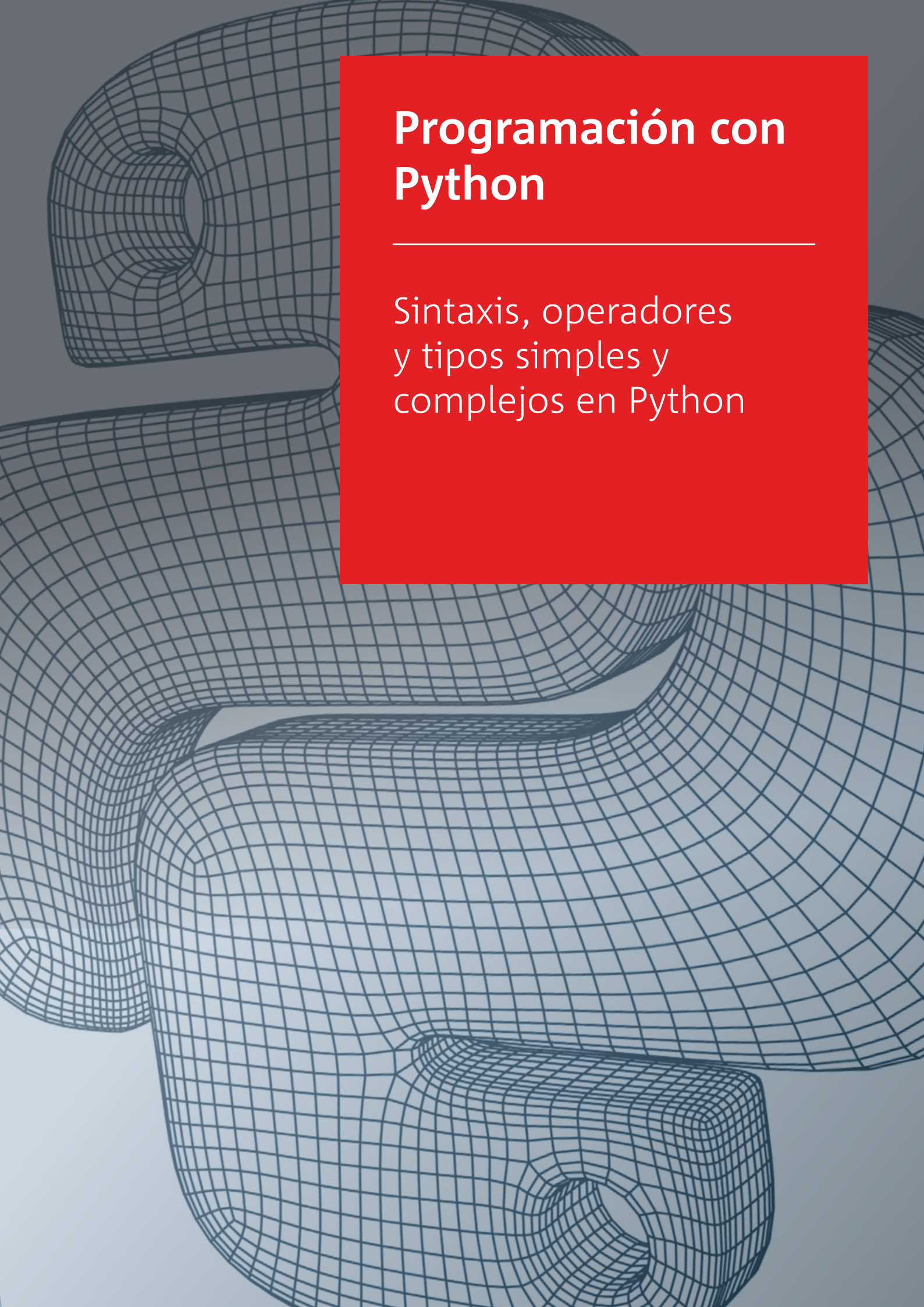

Programación con Python

Sintaxis, operadores y tipos simples y complejos en Python

INICIATIVA Y COORDINACIÓN
DEUSTO FORMACIÓN

COLABORADORES
Realización:
E-Mafe E-Learning Solutions S.L.

Elaboración de contenidos:
Claudio García Martorell
Licenciado IT Telecomunicaciones especialidad Telemática.
Postgrado en Sistemas de Comunicación y Certificación en Business Intelligence TargIT University.
Concejal de Innovación y Tecnología.
Ponente y docente en distintas universidades y eventos.
Josep Estarlich Pau
Técnico de Ingeniería Informática.
Director Área de Software de la empresa Dismuntel.
Participante en proyectos con Python, C#, R y PHP orientados a *Machine Learning* y a la Inteligencia Artificial.

Supervisión técnica y pedagógica:
Grufium educación y excelencia

Coordinación editorial:
Grufium educación y excelencia

Barcelona (España), 2021

Primera edición: septiembre 2021

ISBN: 978-84-1300-688-8 (Obra completa)
ISBN: 978-84-1300-691-8 (Sintaxis, operadores y tipos simples y complejos en Python)

Depósito Legal: B 11038-2021

Impreso por:
SERVINFORM
Avenida de los Premios Nobel, 37
Polígono Casablanca
28850 Torrejón de Ardoz (Madrid)

Printed in Spain
Impreso en España

Esquema de contenido

1. Sintaxis básica

1.1. Variables y constantes
1.2. Tipos
1.3. Definición, modificación y borrado

2. Operadores matemáticos y lógicos

2.1. Operaciones básicas
2.2. Casos especiales en la división
2.3. Potencias y raíces
2.4. Funciones integradas
2.5. Operadores y expresiones lógicas
2.6. Operadores de asignación

3. Tipos de datos simples o básicos

3.1. Cadenas de texto
3.2. Booleanos
3.3. Enteros y decimales
3.4. *Long* y *Float*
3.5. Números complejos (tipo *complex*)

4. Tipos de datos complejos

4.1. Rangos
4.2. Tuplas
4.3. Listas
4.4. Diccionarios

Introducción

En este módulo se van a sentar las bases para la programación en lenguaje Python. Tanto en el ámbito léxico como en el sintáctico, vamos a ver uno por uno todos los componentes de Python: variables, operadores, comparadores, tipos de datos simples, como los números enteros o las cadenas de texto, y tipos de datos complejos, como las listas, las tuplas o los diccionarios.

Conocer cada uno de estos elementos es fundamental para desarrollar cualquier tipo de *software* en Python, ya que esto nos dará un conocimiento básico para componer después estructuras más complejas y desarrollar algoritmos potentes, exprimiendo así al máximo el potencial de Python.

Además, si ya se poseen conocimientos de otros lenguajes, es aquí donde se empieza a optimizar el tiempo de aprendizaje: si bien Python tiene sus peculiaridades a la hora de tratar con los tipos de datos, guarda muchas similitudes con otros lenguajes de alto nivel (como Java o PHP, por ejemplo) y, en mayor o menor medida, simplifica la sintaxis de estos.

1. Sintaxis básica

El primer paso en el aprendizaje de las bases del lenguaje Python pasa por conocer las variables, los distintos tipos de dato que existen, las constantes y la composición de sentencias (mayormente simples, de momento), y cómo operar con todo ello.

Vamos a ver que, en este paso tan básico, se sigue la misma filosofía que en muchos otros lenguajes de alto nivel interpretados, lo que nos será de gran ayuda a la hora de no caer en errores a medida que avancemos en el módulo.

1.1. Variables y constantes

1.1.1. Variables

Las variables son objetos que poseen un valor y que se alojan en la memoria. Esto es, son pequeños "contenedores de información" que podremos usar, bien sea definiendo el valor que van a tener esas variables o modificándolos. Más adelante redefiniremos bien este concepto, pero nos centraremos de momento en las variables como "contenedores".

La naturaleza de la información que se almacena en una variable es lo que se conoce como **tipo**; por ejemplo, si guardamos una cadena de texto, nuestra variable será de tipo texto; si guardamos un número, nuestra variable será un entero.

Para asignar un valor a una variable, se utiliza el símbolo de la igualdad ("="). A diferencia de la concepción matemática de este símbolo, que representaría una igualdad, en programación hay que entenderlo como un operador de asignación. Al utilizarlo, le estamos indicando al programa que calcule lo que hay a la derecha del símbolo igual ("=") y lo asigne a la variable que está a la izquierda.

En esos lenguajes, no estaría permitido escribir

```
>>> 2 = a
```

...porque "2" no es un nombre de variable válido (Figura 1.1).

Certificación

Es importante interiorizar el concepto de variables y constantes, pues en numerosas preguntas de la certificación se hace referencia a estas.

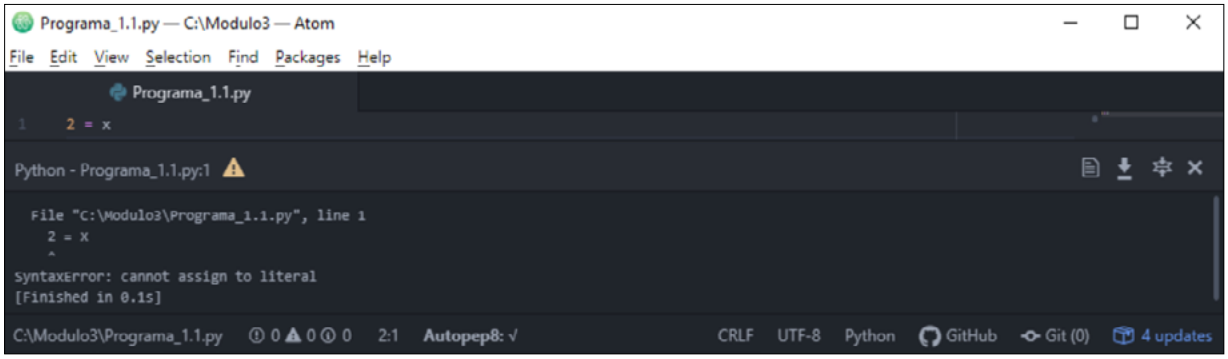

Figura 1.1
El valor se ha de situar a la derecha del igual.

Tampoco estaría permitido escribir

```
>>> x + 3 = 5
```

...porque en el lado izquierdo no pueden aparecer operadores (Figura 1.2).

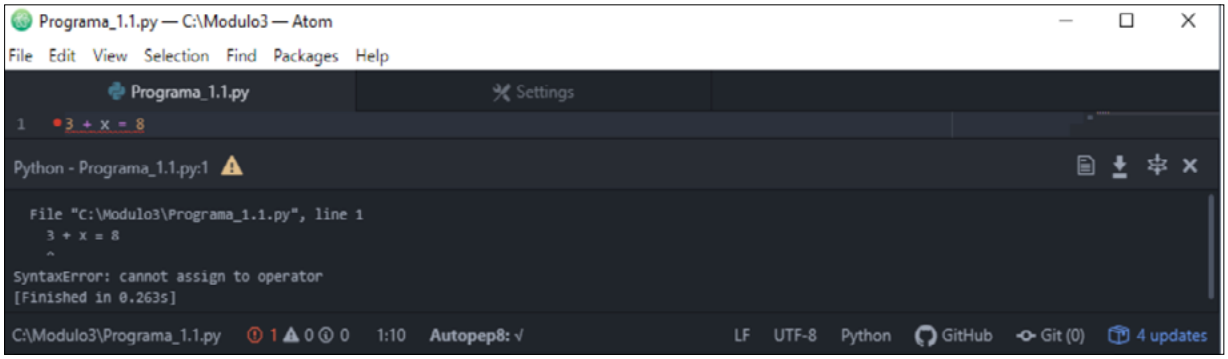

Figura 1.2
Los operadores no pueden estar a la izquierda del igual.

En este último caso, nos da un error por asignar una variable con un nombre de solo caracteres numéricos. La aplicación nos marca un error en la sintaxis previa a la ejecución del código. Una vez ejecutado, nos señala dónde se encuentra dicho error y nos explica que no se puede asignar de este modo.

Cada variable va a tener un nombre único; no podemos definir dos variables con el mismo nombre. Por el contrario, sí que podemos asignar el mismo valor y tipo a dos variables de diferente nombre, ya que se consideran dos variables diferentes, o, dicho de otra manera, como dos "contenedores de información" diferentes.

El alcance de las variables en Python es, por defecto, local. Esto significa que las variables son válidas o se pueden usar dentro del mismo bloque de código en el que se definen, por lo que no interfieren con variables del mismo nombre fuera de ese bloque de código. Es decir, si definimos una

variable dentro de una función, esa variable se podrá usar dentro de esa función, pero no fuera. Si creamos fuera de esa función una variable con el mismo nombre, aunque dentro de la función le hubiésemos asignado un valor, fuera de esta lo perderá porque, en esencia, son variables distintas. De la misma forma, una variable definida fuera de una función tiene validez fuera de esta, pero no dentro.

Por otro lado, una variable puede pasar del ámbito local (por defecto) al ámbito global (como variable global) definiéndola como global; es decir, se declararía como global en el código y su ámbito sería todo el código y no solo el bloque de este donde está definida.

- **Ejemplo de asignación de valor a variable:**

```
import sys
import io

sys.stdout = io.TextIOWrapper(sys.stdout.detach(), encoding='utf-8')
sys.stderr = io.TextIOWrapper(sys.stderr.detach(), encoding='utf-8')

x = "Aprendemos en el módulo "
n = 3
print(x,3)
print("La variable 'x' es de tipo ", type(x), "y la variable n es de tipo ", type(n))
```

Salida:

```
Aprendemos en el módulo  3
La variable 'x' es de tipo  <class 'str'> y la variable n es de tipo  <class 'int'>
```

De la línea 1 a la 5 incluimos lo necesario para que la ejecución nos permita mostrar los caracteres UTF-8 como á, ó...; solo se debe utilizar si ejecutamos con ATOM, si lo ejecutamos con **google colab**, debemos eliminar estas líneas.

En la línea 7 creamos una variable de tipo *string* con el texto "Aprendemos en el módulo".

En la siguiente línea declaramos una variable de tipo int con el valor "3".

En la siguiente línea concatenamos las dos variables en la misma salida por consola mediante la función "print()".

En la línea 10 mostramos los tipos que automáticamente ha asignado Python a nuestras asignaciones.

Podemos visualizar en la consola la salida de las dos funciones “print()” ejecutadas en nuestro código.

Por norma general, las variables en Python no se definen; se les asigna un valor, pero sin declarar su tipo. En esto difiere de Java, por ejemplo, en el que, para inicializar una variable a un valor, hay que definir de qué tipo va a ser la variable.

```
Stringc=”Hola Mundo”;
inte=23;
```

- **Ejemplo de cambio de valor a variable:**

```
x = “Aprendemos en el módulo “
n = 3
x = x + str(n)
print(x)
```

Cuaderno Colab

Practica **este ejemplo** en Colab con los conocimientos adquiridos hasta esta unidad.

Salida:

```
Aprendemos en el módulo 3
```

Utilizando el ejemplo anterior, agregamos la línea número 9, en la cual inicialmente vamos a convertir la variable *n* con la función “str()” en tipo *string* y así podemos concatenar mediante el operador “+” la variable *x* y la variable *n*. Esta concatenación queremos mantenerla en la variable x, que va a ser sobrescrita con el resultado de la operación que está a la derecha del signo “=”.

Como podemos visualizar en la consola, la función “print()” que hemos incluido en nuestro **código nos muestra la concatenación deseada**.

- **Ejemplo de asignación de múltiples valores a múltiples variables**

A continuación, se crearán múltiples variables (entero, coma flotante, cadenas de caracteres) a las que asignaremos múltiples valores:

```
x, y, z = 7, 12.25, “Prueba”
print(x)
print(y)
print(z)
```

Salida:

```
7
12.25
Prueba
```

Si se quiere asignar el mismo valor a múltiples variables al mismo tiempo, podemos definirlo de la siguiente manera:

```
a = b = c = 25
print(a)
print(b)
print(c)
```

Salida:

```
25
25
25
```

El segundo programa asigna el mismo valor booleano a las tres variables *x*, *y*, *z*.

1.1.2. Constantes

Una constante en programación tiene el mismo significado que, por ejemplo, en el ámbito matemático: un tipo de variable cuyo valor no puede cambiarse, es siempre el mismo.

Las constantes en Python se declaran en archivos independientes del archivo principal, de modo que sirven de “contenedor de constantes” que pueden ser importadas por varios ficheros si fuese necesario. Esto suele ocurrir cuando el valor de las constantes es común para varias tareas y no vale la pena declararlas en cada programa por razones de rendimiento o economía de lenguaje. Ejemplos claros suelen ser los datos de conexión a una base de datos o los parámetros clave del servidor.

Las constantes se suelen declarar en mayúsculas, y si su nombre está compuesto de varias palabras, se utiliza el guion bajo para separarlas.

Hay una serie de constantes integradas dentro del propio espacio de nombres de Python y, si bien no se pueden alterar, sí se pueden usar. Son las llamadas ***built-in constants***:

None
NotImplemented
Ellipsis
False
True
__debug__

Estas son las constantes integradas por defecto en Python 3, pero si por ejemplo nos fijamos en el módulo matemático de Python (módulo Math), podemos encontrar las constantes PI o E.

A continuación, se presentan algunos ejemplos del uso de *constantes*:

- **Ejemplo de constantes desde un módulo externo**

Fichero variables.py

```
Web = “http://modulo.python.es”
Usuario = “Claudio”
Contraseña = “123456#”
Edad = 39
```

Fichero Programa.py

```
import variables as var

print(“Quiero acceder a la web ({0}) con usuario {1} y contraseña {2}”.format(
    var.Web, var.Usuario, var.Contraseña
))
print(“Mi edad es de {0} años”.format(
    var.Edad
))
```

Salida:

```
Quiero acceder a la web (http://modulo.python.es) con usuario Claudio y contrase-
ña 123456#
Mi edad es de 39 años
```

1.2. Tipos

El tipo de variable que podemos encontrar en Python dependerá, como ya se ha dicho antes, de la naturaleza de su contenido. Atendiendo a esta, podremos tener números enteros, cadenas de texto, variables booleanas o listas.

Categoría de tipo	Nombre	Descripción
Números inmutables	int	Número entero
	long	Número entero de tipo *long*
	float	Número en coma flotante
	complex	Número complejo
	bool	Valor booleano
Secuencias inmutables	str	Cadena de caracteres
	unicode	Cadena de caracteres Unicode
	tuple	Tupla
	xrange	Rango inmutable
Secuencias mutables	list	Lista
	range	Rango mutable
Mapeos	dict	Diccionario
Conjuntos mutables	set	Conjunto mutable
Conjuntos inmutables	frozenset	Conjunto inmutable

Más adelante veremos una diferenciación de tipos atendiendo a un criterio de complejidad, ya que separaremos las variables según sean tipos de datos simples o complejos. Ahora mismo, podemos hacer una separación atendiendo a la flexibilidad del valor de cada variable para cambiar en tiempo de ejecución, de manera que encontramos tipos de datos mutables e inmutables.

- **Mutable:** su contenido puede cambiarse en tiempo de ejecución.

- **Inmutable:** su contenido no puede cambiarse en tiempo de ejecución.

1.3. Definición, modificación y borrado

Por todo lo visto hasta ahora, podemos asegurar que Python es un lenguaje de programación eminentemente orientado a objetos y que todo

su modelo de datos está basado en ellos. Esto significa que, para cada dato que aparece en un programa, Python crea un objeto que lo contiene. Este objeto vendrá definido por:

- Un ID unívoco (una especie de matrícula o serializador que identifica de manera única a cada objeto).
- El tipo de dato de objeto (entero, coma flotante, booleano, decimal, cadena de caracteres, etc.). Esto dará la pista de las operaciones que pueden realizarse con este objeto.
- El valor (la información que contiene el objeto).

De esta manera, y siendo puristas, las variables en Python no son los "contenedores de información", sino que simplemente son identificadores, nombres de objetos creados para el almacenaje de valores, que sí serían los auténticos "contenedores de datos".

Así, pues, si escribiésemos algo como:

```
variable1 = 42
```

La secuencia que se dispararía de manera interna sería:

- Se crea el objeto "42".
- A este objeto se le asigna un ID numérico para que Python pueda identificarlo a lo largo de todo el ciclo de ejecución del programa.
- El objeto creado guardará el valor "42".
- El objeto creado será del tipo *número* (entero).
- El objeto tendrá el nombre "variable1".

En Python podemos definir en una sola línea diversas variables, bien sea con el mismo o con distinto valor.

En el primer caso, tanto a la variable a como la variable *b* les será asignado el valor 99, mientras que, en el segundo caso, a la primera variable se le asignará el primer valor después del operador de igual ("="), a la segunda variable se le asignará el segundo valor después del igual, etc.

Si el número de variables a la izquierda no coincidiese con el número de valores a la derecha, Python generaría un mensaje de error.

Correcto:

```
x, y = 1, 2
```

Incorrecto:

```
x, y = 1, 2, 5
```

Salida

```
ValueError
Traceback (most recent call last)
<ipython-input-4-0abf21e58065> in <module>()
----> 1 x, y = 1, 2, 5
      2

ValueError: too many values to unpack (expected 2)
```

De la misma forma, se pueden modificar distintas variables en una sola línea, lo que supone un ahorro de código, ya que la modificación o el intercambio de datos se produce en un solo paso.

```
x, y = 1, 2
print(x, y)
x, y = y, x
print(x,y)
```

Salida

```
1 2
2 1
```

Como curiosidad, si este mismo proceso lo hubiésemos querido replicar sentencia a sentencia, no hubiésemos podido, ya que el resultado sería completamente diferente.

```
x, y = 1, 2
print(x, y)
x, y = y, x
x=y #si hacemos esta asignación perdemos el anterior valor de x
y=x
print(x,y)
```

Cuaderno Colab

Practica **este ejemplo** en Colab con los conocimientos adquiridos hasta esta unidad.

Salida

```
1 2
1 1
```

En cuanto al borrado o la destrucción de variables en Python, volvemos sobre la redefinición de variables que hemos hecho al principio de este punto: hay que entender las variables como objetos, y un objeto existe mientras esté referenciado. Cuando desparece la última referencia al objeto, este desaparece mediante una llamada a su destructor (método "*__del__*") y desaparecerá entonces de la memoria.

No nos preocupamos de esta tarea de manera activa ya que, cuando finaliza la ejecución de una función o método, todas las referencias que se habían creado en él desaparecen (ya se hizo referencia con anterioridad al ámbito local de las variables en Python por defecto), ya que todos los objetos creados dejan de referenciarse y, por tanto, desaparecen, exceptuando aquellos que se devuelvan como resultado de una función o método.

Sin embargo, existen algunas ocasiones en las que se guardan las referencias a objetos que ya no nos van a hacer falta de ese punto del programa en adelante; al no dejar de referenciarlos, no se destruyen y ocupan memoria, por lo que, dependiendo del algoritmo o su recursividad, pueden llegar a representar un problema de rendimiento. Es por ello por lo que podemos hacer uso de la instrucción de borrado en Python.

La instrucción "del" borra completamente una variable.

```
x, y = 1, 2
print(x, y)
del x
print(x)
```

Salida

```
1 2
NameError                                 Traceback (most recent
call last)
<ipython-input-7-5fa0aed3642c> in <module>()
      2 print(x, y)
      3 del x
----> 4 print(x)
      5

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

Ejercicios resueltos

Practica el **ejercicio 1** con los conocimientos adquiridos hasta esta unidad.

Resumen

- Las variables son objetos que poseen un valor y se alojan en la memoria. La naturaleza de la información que se almacena en una variable es lo que se conoce como *tipo*. Para asignar un valor a una variable, se utiliza el símbolo de la igualdad ("=").

- Cada variable va a tener un nombre único, no podemos definir dos variables con el mismo nombre. Por el contrario, sí que podemos asignar el mismo valor y tipo a dos variables de diferente nombre.

- En programación, una constante tiene el mismo significado que, por ejemplo, en el ámbito matemático: un tipo de variable cuyo valor no puede cambiarse es siempre el mismo. Las constantes en Python se declaran en archivos independientes del archivo principal.

- Las constantes se suelen declarar en mayúsculas; si su nombre está compuesto de varias palabras, se utiliza el guion bajo para separarlas.

- Los tipos de datos se pueden separar en diferentes categorías atendiendo a si pueden o no ser modificados en tiempo de ejecución: números inmutables, secuencias inmutables, secuencias mutables, mapeos, conjuntos mutables y conjuntos inmutables.

2. Operadores matemáticos y lógicos

Vamos a ver cómo empezar a hacer cálculos y comparaciones simples con las variables en Python. Los cálculos aritméticos y las comparaciones lógicas son la base de estructuras más complejas, como algoritmos y tipos de datos (tanto simples como complejos), que se verán más adelante.

Además, para refrescar conceptos matemáticos veremos los criterios de preferencia de evaluación de elementos y operandos, ya que Python es un lenguaje del que se suele decir que "ahorra mucho código", y muchas veces las instrucciones carecen de elementos de seguimiento o encapsulamiento como paréntesis; por esa razón, hay que saber el orden interno que Python va a seguir a la hora de ejecutar una instrucción.

2.1. Operaciones básicas

Por operaciones básicas entendemos las cuatro operaciones matemáticas fundamentales:

- Suma (operador "+")
- Resta (operador "-")
- Multiplicación (operador "*")
- División (operador "/")

En las operaciones de producto (o multiplicación) en las que intervienen números decimales, el resultado siempre va a ser decimal; por ello, si en el resultado no hay parte decimal como tal, Python escribirá un 0 como parte decimal.

```
print(4.5*3)
print(4.5*2)
```

Salida:

```
13.5
9.0
```

Certificación

Es importante interiorizar los operadores matemáticos y lógicos, pues en numerosas preguntas de la certificación se hace referencia a estos.

En el caso de las divisiones, Python se comporta igual que en las multiplicaciones respecto a los decimales del resultado.

```
print(9/2)
print(9/3)
```

Salida:

```
4.5
3.0
```

En el caso de la suma o la resta de número enteros, el resultado es un número entero.

Al sumar, restar o multiplicar números enteros, el resultado es entero a no ser que en uno de los dos operandos haya un número decimal; en tal caso, el resultado será decimal (aunque, al igual que en casos anteriores, la parte decimal sea nula).

```
print(1+2)
print(3-4)
print(5*6)
print(5.0*6)
```

Salida:

```
 3
-1
30
30.0
```

Dividir por cero genera un error:

```
print(5*0)
```

Salida:

```
ZeroDivisionError                         Traceback (most recent
call last)
<ipython-input-13-fad870a50e27> in <module>()
----> 1 print(5/0)

ZeroDivisionError: division by zero
```

Si en una fórmula coinciden varias operaciones y no hay una preferencia u orden marcado mediante paréntesis, Python da prioridad a las multiplicaciones, divisiones, sumas y restas siguiendo las reglas matemáticas.

```
print(1+2*3)
```

Salida:

```
7
```

En algunas operaciones en las que el resultado contenga decimales, pueden existir errores de redondeo.

```
print(100/3)
```

Salida:

```
33.333333333333336
```

Esto se debe principalmente a que Python tiene un sistema de almacenaje de valores en binario; al pasarlo a decimal, puede darse este error, que, por otro lado, es bastante común en lenguajes de programación y se subsana utilizando bibliotecas matemáticas más específicas. Así, pues, una operación que debería dar el mismo resultado independientemente del sentido en el que se resuelva, nos da resultados diferentes:

```
print(4*3/5)
print(4/5*3)
```

Salida:

```
2.4
2.4000000000000004
```

Si se quiere prescindir de los paréntesis, se pueden escribir operaciones de suma y resta de forma seguida; pero, obviamente, no se recomienda este tipo de notación al no ser una estructura habitual.

En la secuencia “1-+-+-5”, el cálculo se realizaría de la siguiente forma:

$1-(+(-(+(-5)))) = 1-(+(-(-5))) = 1-(+(+5)) = 1-(+5) = 1-5$

Lo que daría como resultado −4.

```
print(3+-+4)
print(3-+-+4)
```

Salida:

```
-1
7
```

Este tipo de notación no se puede usar con divisiones y multiplicaciones:

```
print(3*/4)
```

Salida:

```
File "<ipython-input-18-423025211193>", line 1 print(3*/4) ^ Synta-
xError: invalid syntax
```

2.2. Casos especiales en la división

Para hallar el cociente de una división, utilizaremos el operador "//". Este operador tiene la misma prioridad que el de la división a la hora de evaluar y ejecutar las operaciones. El resultado será siempre un número entero, pero podrá ser decimal siempre que la división realmente tenga una parte decimal o que en el dividendo o en el divisor hayamos puesto una parte decimal (incluso aunque esa parte decimal sea cero o nula):

```
print(100//3)
print(42//4)
print(40.0//7)
print(10//6.0)
```

Salida:

```
30
10
2.0
1.0
```

Si en lugar de el cociente queremos hallar el resto de la división, utilizaremos el operador "%". Este operador tiene la misma prioridad que el de la división. En cuanto a los decimales que pueda mostrar (o no), sigue las mismas reglas que el operador "//".

```
print(100 % 3)
print(10 % 4)
print(20 % 4)
print(10.5 % 3)
```

Salida:

```
10
2
0
1.5
```

Cuando el resultado es decimal, pueden aparecer los problemas de redondeo comentados anteriormente.

```
print(20.2 % 7)
print(round (15.2 % 2,2))
```

Salida:

```
6.1999999999999993
1.2
```

2.3. Potencias y raíces

Para calcular una potencia, usaremos el operador "**". El equivalente matemático será: x ** y = xy.

Este operador tiene preferencia sobre las divisiones y las multiplicaciones a la hora de evaluar ecuaciones.

Utilizando exponentes negativos o decimales, se pueden calcular potencias inversas o raíces n-ésimas.

```
print(2**8)
print(4**.2)
print(5**0.3)
print(-42**0.5)
print((-7)**0.2)
```

Salida:

```
256
```

```
1.3195079107728942
1.6206565966927624
-6.48074069840786
(1.1939255675724383+0.8674377001143145j)
```

Existe una función integrada llamada “pow(x, y)” que se puede utilizar para hacer el cálculo de potencias y raíces. Si se da un tercer argumento, “pow(x, y, z)”, la función calcula primero x elevado a y, y después, el resto de la división por z.

```
print(pow(5,3))
print(pow(2,0.5))
print(pow(6,2,5))
```

Salida:

```
125
1.4142135623730951
1
```

Cuaderno Colab

Practica **este ejemplo** en Colab con los conocimientos adquiridos hasta esta unidad.

2.4. Funciones integradas

2.4.1. Cociente y resto: “divmod()”

Mediante el planteamiento “divmod(x, y)”, se obtiene una tupla formada por el cociente y el resto de *x* entre *y*

```
print(divmod(12, 5))
```

Salida:

```
(2, 2)
```

2.4.2. Redondeo: “round()”

Es una de las funciones más usadas para la representación de resultados y también una de las más complejas a la hora de entender su funcionamiento y su casuística para mostrar ciertos resultados. Para redondear un número se utiliza la función integrada “round()”. Esta función puede admitir hasta dos argumentos.

- En el caso de especificar solo un argumento, “round(x)” devuelve como resultado el valor redondeado con el entero más próximo a *x*:

```
print(round(42.55))
print(round(42.45))
print(round(-42.95))
print(round(-42.82))
```

Salida:

```
43
42
-43
-43
```

- Si se escriben dos argumentos, “round()” devuelve el primer argumento redondeado en la posición indicada por el segundo argumento. Si el segundo argumento es positivo, el primer argumento se redondea con el número de decimales indicado:

```
print(round(42.3535, 2))
print(round(42.4567, 1))
print(round(-42.4527, 6))
```

Salida:

```
42.35
42.5
-42.4527
```

Si se piden más decimales de los que tiene el número, se obtiene el primer argumento, sin cambios:

```
print(round(4.4527, 9))
```

Salida:

```
4.4527
```

Si el segundo argumento es 0 y el primero es un número decimal, se redondea al entero más próximo, como cuando solo hay especificado un argumento; la principal diferencia estriba en que el resultado que obtendremos es decimal y no entero.

```
print(round(42.4532, 0))
print(round(42.4532))
```

Salida:

```
42.0
4
```

Si el segundo argumento es negativo, se redondea a decenas, centenas, etc.

```
print(round(42593, -2))
print(round(42593, -7))
print(round(42593, -4))
```

Salida:

```
42600
0
40000
```

Existe un caso especial en la gestión del redondeo en la función "round()": es el redondeo cuando el valor está justo en medio (por ejemplo, redondear 42,5 a entero, 82,85 a décimas, etc.). En matemáticas, este cálculo se hace aproximando siempre hacia el valor más alto, pero en Python es diferente: cuando se redondea a enteros, (decenas, centenas, etc.), Python redondea de manera que la última cifra (la redondeada) sea par.

```
print(round(9.5))
print(round(7.5))
print(round(8.5))
print(round(6.5))
print(round(320, -2))
print(round(420, -2))
print(round(770, -2))
print(round(950, -2))
print(round(30, -2))
```

Salida:

```
10
8
8
6
300
```

```
400
800
1000
0
```

Por otro lado, cuando el criterio de redondeo implica llegar hasta las décimas, centésimas, etc., Python hará el redondeo en algunos casos hacia el número superior y en otros casos hacia el inferior; esto, aunque parezca aleatorio, viene dado por la forma en la que se "traducen" los números desde su almacenamiento binario hasta su representación en decimal (como ya se ha dicho anteriormente).

```
print(round(2.14, 1))
print(round(31.245, 1))
print(round(33.75, 1))
print(round(43.85, 1))
print(round(34.75, 1))
print(round(76.85, 1))
print(round(0.225, 1))
print(round(0.215, 1))
print(round(0.155, 1))
print(round(0.745, 1))
print(round(0.785, 1))
print(round(0.555, 1))
```

Salida:

```
2.1
31.2
33.8
43.9
34.8
76.8
0.2
0.2
0.2
0.7
0.8
0.6
```

Cuaderno Colab

Practica **este ejemplo** en Colab con los conocimientos adquiridos hasta esta unidad.

2.4.3. Redondeo anterior y posterior: "floor()" y "ceil()"

Una forma de automatizar y decretar un criterio firme para los redondeos y no depender de las traducciones internas de Python de binario a decimal son las funciones "floor()" y "ceil()". Ambas están incluidas en

la biblioteca Math. Estas funciones solo admiten un argumento (el número o el cálculo que hay que redondear) y devuelven valores enteros.

La función “floor()” redondeará hacia el entero inferior, mientras que “ceil()” redondeará al entero superior.

Antes de utilizar estas funciones, hay que importarlas; de lo contrario, se generará un error.

```
print(floor(5/2))
```

Salida:

```
NameError
Traceback (most recent call last)
<ipython-input-32-e01668bd2985> in <module>()
----> 1 print(floor(5/2))

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

```
print(ceil(5/2))
```

Salida:

```
NameError
Traceback (most recent call last)
<ipython-input-33-23144d875ff7> in <module>()
----> 1 print(ceil(5/2))

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

```
from math import floor
from math import ceil
print(floor(5/2))
print(ceil(5/2))
```

Salida:

```
2
3
```

2.4.4. Valor absoluto: “abs()”

La función integrada “abs()” calcula el valor absoluto de un número; es decir, nos devolverá el mismo valor pero sin signo, independientemente de si es positivo o negativo.

```
print(abs(-60))
print(abs(42))
```

Salida:

```
60
42
```

2.4.5. Máximo: “max()”

La función integrada “max()” calcula el valor máximo de un conjunto de valores (números o letras). En el caso de las cadenas de texto, los valores máximos corresponden a los últimos valores en una jerarquía alfabética, independientemente de la longitud de la cadena. Las vocales acentuadas y las letras especiales, como la *ñ* o la *ç*, se consideran posteriores al resto de las consonantes y vocales.

```
print(max(4, 5, -2, 8, 3.5, -10))
print(max(“Josep”, “Claudio”))
```

Salida:

```
8
Josep
```

2.4.6. Mínimo: “min()”

La función integrada “min()” calcula el valor mínimo de un conjunto de valores (números o letras). En el caso de las cadenas de texto, los valores mínimos corresponden a los primeros valores en una jerarquía alfabética, independientemente de la longitud de la cadena. Como ya se ha mencionado, las vocales acentuadas y las letras especiales como la *ñ* o la *ç* se consideran posteriores al resto de las consonantes y las vocales.

```
print(min(4, 5, -2, 8, 3.5, -10))
print(min(“Josep”, “Claudio”))
```

Salida:

```
-10
Claudio
```

Las vocales acentuadas, la letra ñ o ç se consideran posteriores al resto de vocales y consonantes

```
print(min("Ángeles", "Roberto"))
```

Salida:

```
Roberto
```

2.4.7. Suma: "sum()"

La función integrada "sum()" calcula la suma de un conjunto de valores. Este ha de ser un tipo iterable de datos, bien sea una tupla, una lista, un rango, un conjunto o un diccionario.

```
print(sum(1, 8, 10, 42, 36))
print(sum(range(4)))
print(sum({1, 4, 48, 10, 5}))
```

Salida:

```
97
6
68
```

2.4.8. Ordenación: "sorted()"

La función integrada "sorted()" ordena un conjunto de valores. Igual que en el punto anterior, los valores sobre los que va a actuar esta función han de ser un tipo de datos iterable (tupla, rango, lista, conjunto o diccionario). A diferencia de en "sum()", los valores no se alteran, ya que no se realiza ninguna operación de asignación entre ellos; el objetivo de "sorted()" es el de comparación y ordenación de los valores, por lo que el resultado será el tipo de datos ordenados.

```
print(sorted(100, 20, 18, -53, 42))
print(sorted([100, 20, 18, -53, 42]))
print(sorted({100, 20, 18, -53, 42}))
```

Cuaderno Colab

Practica **este ejemplo** en Colab con los conocimientos adquiridos hasta esta unidad.

Salida:

```
[-53, 18, 20, 42, 100]
[-53, 18, 20, 42, 100]
[-53, 18, 20, 42, 100]
```

2.5. Operadores y expresiones lógicas

2.5.1. Operadores lógicos

A diferencia de los operadores que hemos visto hasta ahora, que trabajaban con valores aritméticos, los **operadores lógicos** trabajan con valores booleanos, es decir, binarios de verdadero (*true*) o falso (*false*).

Los valores "True" y "False" se escriben con la primera letra en mayúscula, mientras que los operadores van en minúscula. En cualquier otro caso, la evaluación de la sentencia nos dará un error.

Conviene recordar que la tabla de los distintos operadores lógicos y su comportamiento ya se expuso anteriormente.

Operador	Ejemplo	Resultado	Evaluación
and	7 == 9 and 7 < 15	False and True	False
and	90< 120 and 3>1	True and True	True
and	15<20 and 15> 13	True and False	False
or	42 == 42 or 12<37	True or False	True
or	37 > 35 or 15<22	True or True	True
xor	40 == 40 xor 6 >2	True o True	False
xor	40 == 40 xor 6<2	True o False	True

- **And:** sería equivalente a un producto. Se entiende como un operador "y" lógico. Este operador da como resultado "True" si y solo si sus dos operandos son "True":

  1 * 1 = 0
  1 * 0 = 0
  0 * 1 = 0
  0 * 0 = 0

  Pasando estas operaciones a código Python:

```
print(True and True)
print(True and False)
print(False and True)
print(False and False)
```

Salida:

```
True
False
False
False
```

- **Or:** sería equivalente a una suma binaria. Se entiende como un operador "o" lógico. Este operador da como resultado "True" si algún operando es "True":

1 + 1 = 1
1 + 0 = 1
0 + 1 = 1
0 + 0 = 0

Pasando estas operaciones a código Python:

```
print(True or True)
print(True or False)
print(False or True)
print(False or False)
```

Salida:

```
True
True
True
False
```

- **Not:** negación. Este operador da como resultado "True" si y solo si su argumento es "False":

```
print(not True)
print(not False)
```

Salida:

```
False
True
```

2.5.2. Expresiones compuestas

Para la escritura y evaluación de expresiones lógicas compuestas o complejas, es recomendable el uso, siempre que sea posible, de paréntesis para una mejor visualización y seguimiento del flujo de la ejecución. En cualquier caso, y como ya se hizo en el punto anterior con los distintos operadores lógicos, conviene recordar la tabla de preferencia de ejecución de operadores lógicos que ya se expuso anteriormente.

Operador	Definición
not	Operador lógico de negación
and	Operador lógico “Y”
or	Operador lógico “O”

El operador “not” se evalúa antes que el operador “and”:

```
print(not True and False)
print((not True) and False)
print(not (True and False))
```

Salida:

```
False
False
True
```

El operador “not” se evalúa antes que el operador “or”:

```
print(not True or False)
print((not True) or False)
print(not (True or False))
```

Salida:

```
False
False
False
```

El operador “and” se evalúa antes que el operador “or”:

```
print(True and False or True)
print((False and True) or True)
print(False and (True or True))
print(True or True and True)
print((True or True) and True))
print(True or (True and True))
```

Salida:

```
True
True
False
True
True
True
```

Si en las expresiones lógicas se utilizan valores distintos de “True” o “False”, Python utiliza esos valores en lugar de “True” o “False”.

```
print(3 or 4)
```

Salida:

```
3
```

Este tipo de resultados no parecen tener demasiada lógica; no tiene mucho sentido que conteste “3” en lugar de “4”, ya que, al tratarse de un “or”, ambos valores serían correctos. El porqué de este tipo de respuesta reside en que cuando Python evalúa el primer valor (3, es decir, “True”), ya sabe que el valor final va a ser “True” y por eso contesta “3”.

Un caso distinto es el del producto o “and”, ya que, al evaluar “3 and 5”, Python tiene que evaluar los dos valores. Como los dos son “True”, el resultado es “True”. Python contesta entonces el último valor que ha evaluado, “5”.

Si se quieren mostrar valores booleanos, se puede convertir el resultado a un valor booleano:

```
print(3 or 4)
print(bool(3 or 4))
```

Salida:

```
3
True
```

2.5.3. Comparaciones

A la hora de comparar valores, estas dan como resultado valores booleanos:

- ">": mayor que.
- "<": menor que.

```
print(30 > 12)
print(30 < 12)
```

Salida:

```
True
False
```

- ">=": mayor o igual que.
- "<=": menor o igual que.

```
print(3 >= 5 + 1)
print(3 - 5 <= 1)
```

Salida:

```
False
True
```

- "==": igual a.
- "!=": distinto a.

```
print(8 == 4 + 4)
print(6 / 3 != 2)
```

Salida:

```
True
False
```

Como recordatorio, es bueno tener presente la tabla de preferencias a la hora de evaluar operadores lógicos, que ya conocemos:

Operador	Definición
<, <=, =>, >	Comparadores lógicos de magnitud
==, !=	Comparadores lógicos de igualdad o desigualdad

Es buen momento para recordar que el signo "=" significa asignación si es único, pero significa comparación si es doble "==". Cualquier expresión mal escrita debido a esto nos dará un error.

Se pueden encadenar varias comparaciones; si el resultado es positivo (o verdadero), será entonces cuando Python lo devuelva como "True".

```
print(4 == 13 + 10 > 20)
print(7 != 55 + 3 > 0)
```

Salida:

```
False
True
```

Este tipo de comparaciones encadenadas no puede darse en otros lenguajes de alto nivel similares a Python como pueden ser Java o PHP.

2.6. Operadores de asignación

En Python, el operador "=" se utiliza para la asignación, como ya se ha dicho en repetidas ocasiones; pero, en combinación con otros operadores aritméticos, se pueden obtener operadores combinados para una serie de asignaciones aumentadas.

Certificación

Es importante interiorizar los operadores de asignación, pues en numerosas preguntas de la certificación se hace referencia a estos.

2.6.1. Operador "="

El operador "igual a", "=", es el más simple de todos y asigna a la variable del lado izquierdo el valor del lado derecho.

2.6.2. Operador “+=”

Suma a la variable del lado izquierdo el valor del lado derecho.

```
r = 12
print(r)
r += 3
print(r)
```

Salida:

```
12
15
```

De una forma más explícita, equivaldría a lo siguiente:

```
r = 12
print(r)
r = r + 3
print(r)
```

Salida:

```
12
15
```

2.6.3. Operador “-=”

Resta a la variable del lado izquierdo el valor del lado derecho.

```
r = 12
print(r)
r -= 3
print(r)
```

Salida:

```
12
9
```

De una forma más explícita, equivaldría a lo siguiente:

```
r = 12
print(r)
r = r - 3
print(r)
```

Salida:

```
12
9
```

2.6.4. Operador "*="

Multiplica la variable del lado izquierdo por el valor del lado derecho.

```
r = 12
print(r)
r *= 3
print(r)
```

Salida:

```
12
36
```

En el ejemplo anterior, si la variable *r* es igual a 5 y *r* *= 10, entonces la variable *r* será igual a 50. Su equivalente sería el siguiente:

```
r = 12
print(r)
r = r * 3
print(r)
```

Salida:

```
12
36
```

2.6.5. Operador "/="

Divide la variable del lado izquierdo por el valor del lado derecho.

```
r = 12
print(r)
r /= 3
print(r)
```

Salida:

```
12
4.0
```

De una forma más explícita, equivaldría a lo siguiente:

```
r = 12
print(r)
r = r / 3
print(r)
```

Salida:

```
12
4.0
```

2.6.6. Operador "**="

Calcula el exponente de la variable del lado izquierdo con el valor del lado derecho.

```
r = 12
print(r)
r **= 3
print(r)
```

Salida:

```
12
1728
```

De una forma más explícita, equivaldría a lo siguiente:

```
r = 12
print(r)
r = r ** 3
print(r)
```

Salida:

```
12
1728
```

Ejercicios resueltos

Practica el **ejercicio 2** con los conocimientos adquiridos hasta esta unidad.

2.6.7. Operador "//="

Calcula la división entera entre la variable del lado izquierdo y el valor del lado derecho.

```
r = 12
print(r)
r //= 3
print(r)
```

Salida:

```
12
4
```

De una forma más explícita, equivaldría a lo siguiente:

```
r = 12
print(r)
r = r // 3
print(r)
```

Salida:

```
12
4
```

2.6.8. Operador "%="

Calcula el resto de la división de la variable del lado izquierdo por el valor del lado derecho.

```
r = 12
print(r)
r %= 3
print(r)
```

Salida:

```
12
0
```

De una forma más explícita, equivaldría a lo siguiente:

```
r = 12
print(r)
r = r % 3
print(r)
```

Cuaderno Colab

Practica **este ejemplo** en Colab con los conocimientos adquiridos hasta esta unidad.

Salida:

```
12
0
```

Resumen

- La división es la operación básica que cuenta con una mayor casuística a la hora de operar con sus elementos. Además, existen algunas diferencias entre la forma de calcular en Python 2 y Python 3 que pueden llegar a afectar al resultado.

- Igual que en matemáticas, existe un orden determinado a la hora de evaluar una expresión aritmética en Python, por lo que siempre es recomendable expresar las operaciones con paréntesis.

- Para generar potencias de números, se puede utilizar tanto el operador "**" como la función "pow(base, exp)".

- Python posee funciones matemáticas ya integradas en el código que nos pueden ayudar a hacer cálculos que, de otra manera, sería bastante complejo expresar mediante un algoritmo (redondeo, valor absoluto, máximos y mínimos, etc.).

- Los operadores lógicos trabajan con valores booleanos ("verdadero" o "falso") y se utilizan para comparar valores como "mayor que", "menor que", "igual a", "distinto a", etc., así como para evaluar condiciones tales como "and", "or", "xor", "not", etc. Para todos ellos, igual que en el formato aritmético, hay una preferencia u orden a la hora de la evaluación de signos, por lo que se recomienda el uso de paréntesis para determinar el orden que quiera el usuario.

- Los operadores de asignación se generan mediante la combinación del signo "=" junto con otros operadores como "+", "-" o "*" (entre otros). Pueden simplificar cálculos y ahorrarnos muchas líneas de código cuando son usados en bucles.

3. Tipos de datos simples o básicos

Hasta ahora hemos estudiado elementos de manera aislada y los hemos ido definiendo individualmente; a partir de ahora, los definiremos con mucho más detalle y los veremos de manera más horizontal, más integrados unos con otros.

Uno de los rasgos más interesantes de Python en lo que respecta a la economía del lenguaje es el formateo de las cadenas de caracteres, muy útil para todo programa que vaya a tener cualquier interacción con el usuario.

3.1. Cadenas de texto

Las cadenas de caracteres son secuencias alfanuméricas que van entrecomilladas. En función de su longitud, pueden ser cadenas cortas o largas.

3.1.1. Cadenas cortas

Pueden ir entre comillas simples (‘) o dobles (‘’).

```
print(‘Aprender python con nosotros es fácil’)
print(“Aprender python con nosotros es fácil”)
```

Salida:

```
Aprender python con nosotros es fácil
Aprender python con nosotros es fácil
```

Certificación

Es importante interiorizar los tipos de datos simples o básicos, pues en numerosas preguntas de la certificación se hace referencia a estos.

3.1.2. Cadenas largas

Pueden contener saltos de línea, comillas internas y otras características propias de textos largos, por lo que suelen estar encerradas entre comillas triples simples (‘’’) o dobles (“””).

```
print(‘’’Mi grupo de Rock
favorito es “The Roquets”’’’)
print(“””Mi grupo de Rock
favorito es \”El Pez Volador\”””””)
```

3.1.3. Prefijo de cadenas

- **“r”** o **“R”** indica que se trata de una cadena *raw* (del inglés, “cruda”). La principal diferencia respecto a las cadenas normales es que los caracteres que escapamos con la contrabarra (o barra invertida) no son sustituidos. Esto es muy útil en la escritura de expresiones regulares.

```
variable_raw = r”\t\nPython ;)\n”
print(type(variable_raw))
```

Salida:

```
<class ‘str’>
```

- **“u”** o **“U”** indica que se trata de una cadena que utiliza codificación Unicode incluida en el tipo str.

```
variable_unicode = U’úáäïPython ;)...’
print(type(variable_unicode))
```

Salida:

```
<class ‘str’>
```

3.1.4. Cadenas de escape

Para escapar caracteres dentro de cadenas de caracteres se usa el carácter “\” seguido de cualquier carácter ASCII.

Secuencia de escape	Significado
\newline	Ignorado.
\\	Backslash (\).
\’	Comilla simple (‘).
\”	Comilla doble (“).
\a	Bell ASCII (BEL).
\b	Backspace ASCII (BS).
\f	Formfeed ASCII (FF).
\n	Linefeed ASCII (LF).

Cuaderno Colab

Practica **este ejemplo** en Colab con los conocimientos adquiridos hasta esta unidad.

Secuencia de escape	Significado
\N{name}	Carácter llamado name en base de datos Unicode (solo Unicode).
\r	Carriage Return ASCII (CR).
\t	Tabulación horizontal ASCII (TAB).
\uxxxx	Carácter con valor hex 16-bit xxxx (solo Unicode). Ver hex.
\Uxxxxxxxx	Carácter con valor hex 32-bit xxxxxxxx (solo Unicode). Ver hex.
\v	Tabulación vertical ASCII (VT).
\ooo	Carácter con valor octal ooo. Ver octal.
\xhh	Carácter con valor hex hh. Ver hex.

3.1.5. Operaciones

Las cadenas de caracteres admiten también operaciones aritméticas, como ya vimos, con las evaluaciones de las funciones “min()” y “max()”.

- La suma para concatenar cadenas de caracteres:

```
x, y, z = “Aprendemos”, “Python”, “ “
print(x+z+y)
```

Salida:

```
Aprendemos Python
```

- La multiplicación para repetir la cadena de caracteres por *N* veces definidas en la multiplicación:

```
x, y, z = “Aprendemos”, “Python”, “ “
print(x+z+(y+z)*2)
```

Salida:

```
Aprendemos Python Python
```

3.1.6. Comentarios

Aunque ya hicimos una introducción a los comentarios anteriormente, vamos a profundizar en estos elementos, su sintaxis y sus aplicaciones.

Los comentarios son cadenas de caracteres que ayudan a entender mejor el código que se está ejecutando, ya que pueden dar información adicional, apuntar variaciones y determinar una versión o procedimiento; son especialmente útiles (y casi de obligado uso) cuando el mismo código va a ser utilizado o modificado por varios programadores. Con una información robusta en los comentarios se puede ahorrar mucho tiempo en pruebas "para ver qué hace" o simplemente para hacerlo funcionar. Más allá de eso, los podemos encontrar para:

- Brindar información general sobre el programa.
- Explicar qué hace cada una de sus partes.
- Aclarar o fundamentar el funcionamiento de un bloque específico de código que no sea evidente en su propia lectura.
- Indicar aspectos pendientes de agregar o mejorar.

El signo para indicar el comienzo de un comentario es "#". A partir de este carácter y hasta el final de la línea, todo lo que haya escrito se considerará un comentario, por lo que será ignorado por el intérprete de Python durante la ejecución.

```
# Cometario en nuestro módulo 3
# No se visualiza nada en consola
```

Python no dispone de un método para determinar bloques de comentarios de varias líneas. Como ya se apuntó con anterioridad, estos van entre triples comillas (independientemente de si estas son simples o dobles). A diferencia de los comentarios de una sola línea, estos tienen el inconveniente de que, si bien no se genera ningún tipo de código ejecutable, el bloque no es ignorado del todo por Python, ya que crea un objeto como una cadena de caracteres.

```
""" Podemos
Comentar en vaira
líneas"""
```

Existen al menos dos alternativas para introducir comentarios multilínea:

- Comentar cada una de las líneas con el carácter.
- Usar triples comillas (simples o dobles) para generar una cadena multilínea.

Ejercicios resueltos

Practica el **ejercicio 3** con los conocimientos adquiridos hasta esta unidad.

3.1.7. Formateo de cadenas

Python soporta múltiples formas de formatear una cadena de caracteres.

- **Formateo "%"**

El carácter "%" es un operador integrado en Python. Con él se puede dar forma a ciertas variables para presentarlas en pantalla. Hay que tener en cuenta los tipos de formateo que puede llegar a realizar:

- %c = *str*, carácter simple.
- %s = *str*, cadena de caracteres.
- %d = *int*, enteros.
- %f = *float*, coma flotante.
- %o = octal.
- %x = hexadecimal.

```
explicacion = "Sumar cinco + siete y multiplicar el resultado por tres"
resultado = (5+7)*3.0
print("El resultado de %s es %f" % (explicacion, resultado))
```

Salida:

```
El resultado de Sumar cinco + siete y multiplicar el resultado por tres es 36.000000
```

También aquí se puede controlar el formato de salida. Por ejemplo, para obtener el valor con 8 dígitos después de la coma:

```
explicacion = "Mi resultado favorito"
resultado = 15.25365632554565854
print("%s es %8f" % (explicacion, resultado))
```

Salida:

```
Mi resultado favorito es 15.253656
```

A continuación, un ejemplo para cada tipo de datos:

```
explicacion = “Mi resultado favorito”
caracter = “S”
resultado = 15.25365632554565854
entero = 154
octal = 0o27
hexa = 0x21
print(“%s es %8f ¿Si (S) o No (N)? %c” % (explicacion, resultado, caracter))
print(“Mi identificador es %d, mi contraseña %o, y tengo %x años” % (entero, octal, hexa))
```

Salida:

```
Mi resultado favorito es 15.253656 ¿Si (S) o No (N)? S
Mi identificador es 154, mi contraseña 27, y tengo 21 años
```

Cuaderno Colab

Practica **este ejemplo** en Colab con los conocimientos adquiridos hasta esta unidad.

- **Clase “formatter”**

“Formatter” es una clase integrada de Python que hace referencia a las cadenas de caracteres. Permite crear y personalizar formatos para reescribir métodos públicos. Contiene las funciones “format()” y “vformat()”.

- **“format()”**

Este método retorna una cadena formateada. Las sustituciones que hay que llevar a cabo son identificadas entre llaves (“{}”) dentro de la cadena de caracteres original y se realizan en el orden en el que aparecen como argumentos, contando a partir de cero.

```
explicacion = “Mi resultado favorito”
caracter = “S”
resultado = 15.25365632554565854
print(“{} es {} ¿Si (S) o No (N)? {}”.format(explicacion, resultado, caracter))
```

Salida:

```
Mi resultado favorito es 15.25365632554566 ¿Si (S) o No (N)? S
```

También se puede referenciar a partir de la posición de los valores utilizando índices:

```
explicacion = “Mi resultado favorito”
caracter = “S”
resultado = 15.25365632554565854
print(“{0} es {2} ¿Si (S) o No (N)? {1}”.format(explicacion, caracter, resultado))
```

Salida:

```
Mi resultado favorito es 15.25365632554566 ¿Si (S) o No (N)? S
```

Otro uso de esta función está vinculado a los objetos, ya que estos también pueden ser referenciados utilizando un identificador (a modo de clave) para más tarde pasarlo como argumento a la función.

```
explicacion = "Mi resultado favorito"
caracter = "S"
resultado = 15.25365632554565854
print("{val} es {val2} ¿Si (S) o No (N)? {val1}".format(val=explicacion, val1=caracter,
val2=resultado))
```

Salida:

```
Mi resultado favorito es 15.25365632554566 ¿Si (S) o No (N)? S
```

Es posible además alinear una cadena con x = 30 caracteres de separación a la derecha:

```
print("{:>30}".format("Alinear cadena"))
```

Salida:

```
Alinear cadena
```

De la misma forma, se puede alinear una cadena de caracteres a la izquierda con x = 30:

```
print("{:=30}".format("Alinear cadena"))
```

Salida:

```
Alinear cadena
```

También se puede alinear una cadena de caracteres al centro en 30 caracteres, con la siguiente sentencia:

```
print("{:^30}".format("Alinear cadena"))
```

Salida:

```
Alinear cadena
```

Se puede limitar la cadena de caracteres a un número concreto de estos:

```
print("{:.9}".format("Cortar cadena"))
```

Salida:

```
Cortar ca
```

3.2. Booleanos

Una variable booleana es una variable que solo puede tomar dos posibles valores: "True" (verdadero) o "False" (falso).

Atendiendo a un criterio de nulidad o no nulidad de un objeto, en Python se podría considerar que toda variable (en realidad, todo objeto) puede ser booleano, ya que si es nulo o vacío se consideraría "False", mientras que el resto se considerarían "True", basándose en la existencia de estos.

Para saber cómo trata Python el objeto, se puede hacer una llamada al método "bool()" para obtener una respuesta:

Campus online

Accede al Campus online para ver el videotutorial *Funciones ampliadas del tipo STRING.*

```
print(bool(0.0))
print(bool(""))
print(bool(None))
print(bool(()))
print(bool([]))
print(bool({}))
print(bool(31))
print(bool(-19))
print(bool("Claudio"))
print(bool([19, "Diciembre", 2021]))
print(bool({19, "Diciembre", 2021}))
```

Salida:

```
False
False
False
False
False
False
True
True
True
True
True
True
```

3.3. Enteros y decimales

Python diferencia entre números enteros y números decimales. Esta diferenciación viene dada por el separador "." entre la parte entera y la decimal.

Uno de los errores más comunes a la hora de escribir constantes o números decimales es utilizar el carácter coma (","), ya que Python no lo identifica como un separador entre parte entera y parte decimal, sino como la separación de dos valores a modo de tupla.

Aunque no es muy común, se pueden escribir los números decimales sin su parte entera:

```
print(3)
print(4.5)
print(3, 5)
print(3.0)
print(.3)
```

Salida:

```
3
4.5
3 5
3.0
0.3
```

Un número entero es el que no tiene decimales, independientemente de si es negativo o positivo. En Python hay dos formas de representarlos, y hacerlo de una forma u otra atiende a un criterio de "longitud" del número, ya que para números enteros muy grandes se utilizará el tipo *long*, mientras que para enteros más pequeños se utilizará el tipo *int*.

No es viable declarar todos los enteros como *long* "por si acaso", para tener cubierta la posibilidad de que sea grande; se recomienda utilizar siempre el tipo *int* sobre todo para ahorrar memoria, a no ser que esté justificado utilizar el tipo *long*. Los enteros se guardan con 32 bits de precisión en Python.

3.4. *Long* y *Float*

3.4.1. Tipo *Long*

El tipo ***long*** sirve en Python para almacenar un número entero, pero sin una limitación fija en cuanto a magnitud, ya que el límite depende de la memoria de la máquina. Por defecto, al asignar un número a una variable, este siempre será definido como *int*, exceptuando cuando el número sea tan grande como para ser del tipo *long* o cuando venga declarado como tal en la propia instrucción.

```
n = 31
print(type(n))
n = 65536*65536*65536*6553665536*6553665536*6553665536*65536
print(type(n))  # Ya no existe el tipo long en python 3, se engloba en int
```

Salida:

```
<class 'int'>
<class 'int'>
```

3.4.2. Tipo *Float*

A diferencia de los números enteros, los números reales sí tienen parte decimal. En Python serán declarados mediante el tipo ***float***.

En lo que respecta a la memoria, Python utiliza lo que equivaldría a una variable de tipo *double* en C, es decir, utiliza el doble de precisión para los tipos *float* que para los tipos *int*, por lo que estaríamos hablando de 64 bits. Desde Python 2.4 se cuenta con un tipo de datos adicional llamado **decimal** para representar fracciones de forma mucho más precisa, aunque ese tipo de datos queda reservado para cálculos científicos avanzados.

El separador de parte entera y parte decimal es el punto ("."), que no debemos confundir con la coma (","), como ya se advirtió en el apartado anterior.

real=0.2703

Si queremos entrar dentro de la notación científica para indicar exponentes en base 10, podemos acompañar el número real de un carácter "e". Así:

real=0.1e-3

Sería equivalente a $0{,}1 \times 10^{-3} = 0{,}1 \times 0{,}001 = 0{,}0001$

3.5. Números complejos (tipo *complex*)

En Python también se pueden hacer cálculos con números complejos. Este tipo de números tienen dos partes bien diferenciadas: la real y la imaginaria (que vendrá acompañada de la letra *j*).

```
print(1+1j+2+3j)
print((1+1j)*1j)
print(1+j)
```

Salida:

```
(3+4j)
(-1+1j)
...
NameError
Traceback (most recent call last)
<ipython-input-88-20e7eec8a1d1> in <module>()
      1 print(1+1j+2+3j)
      2 print((1+1j)*1j)
----> 3 print(1+j)
```

Incluso cuando la parte imaginaria sea 1, este valor deberá aparecer en la definición del número acompañado de la *j*, ya que esta no puede ir sola.

El resultado de una operación con números complejos es siempre un número complejo, aunque su parte imaginaria sea nula. De la misma forma que en el ejemplo anterior, siempre ha de haber un número acompañando a la parte imaginaria.

```
print(1j*1j)
```

Salida:

```
(-1+0j)
```

La mayor parte de los lenguajes de programación carecen de este tipo de datos, ya que se reserva muchas veces a ámbitos más científicos y para esos usos se pueden importar módulos matemáticos o de mayor precisión de datos. En lo referente a la memoria, Python utiliza dos variables de tipo *double* para almacenar el tipo *complex*, que es el que hace referencia a los números complejos. Cada una de las variables double almacena una de las partes del número complejo: por un lado, la parte real, y por otro, la imaginaria.

Resumen

- En el ámbito matemático, se definen los números enteros, decimales y complejos en sus tipos de datos correspondientes, dependiendo de la longitud o del tipo de operaciones para las que están indicados. Además, hay que tener en cuenta las funciones y métodos integrados del propio Python que podemos utilizar, y cuáles, debido a la dificultad, el nivel de precisión o la complejidad de cálculo, hay que importar con un módulo o una biblioteca externa como Math.

- Se entiende por cadena de caracteres una secuencia alfanumérica que va entrecomillada. El entrecomillado puede ser simple, doble o triple, dependiendo del propósito visual que queramos darle.

- Los comentarios pueden ocupar una línea, varias o incluso pueden estar en la misma línea de la instrucción del código. Sirven para explicar o aclarar una parte del código.

- Las variables booleanas pueden tomar solo dos valores: "True" o "False". Atendiendo a un criterio de nulidad o no nulidad de un objeto, en Python se podría considerar que toda variable puede ser booleana dependiendo de si está vacía o no.

- Los números enteros y decimales se diferencian por la parte decimal, que viene dada por el separador, el punto (".."). Uno de los errores más comunes es utilizar como separador la coma (","), ya que Python no la identifica como separador entre parte entera y decimal, sino como separación de dos valores (tupla).

- El tipo *long* se utiliza para almacenar números de gran magnitud pero sin un límite bien definido, ya que este viene dado por la memoria restante de la máquina.

- Los números complejos se pueden definir mediante el tipo *complex* y tendrán siempre una parte real y otra imaginaria, aunque alguna de ellas puede ser nula.

4. Tipos de datos complejos

Después de ver los tipos de datos simples en Python y algunas de sus aplicaciones, vamos a ir un paso más allá y estudiaremos y definiremos los tipos de datos complejos. Estos pueden considerarse entidades de nivel superior a los tipos de datos simples, ya que, si bien hasta ahora teníamos como tipos de datos enteros o cadenas de caracteres, en este caso tendremos unos tipos de datos en cuyo interior pueden coexistir varios tipos simples.

Estos tipos de datos complejos son, a su vez, la base para entender lo que está por venir en el siguiente módulo: bucles, estructuras de control y condiciones de recursividad.

4.1. Rangos

Un **rango** se define como una lista inalterable de números enteros en sucesión aritmética; es decir, que la diferencia entre los diferentes valores será siempre la misma. Se define con el tipo de datos ***range***.

Un *range* se define creando el tipo de datos con hasta tres valores numéricos, de la misma forma que se crearía una función. Sin embargo, aquí está una de las diferencias entre Python 2 y Python 3 que ya hemos señalado anteriormente, y es que en Python 2 "range()" se consideraba una función, mientras que en Python 3 se considera un tipo de datos (aunque su declaración y utilización se asemejen a las de una función).

- **"range(n)" con un argumento:** se crea una lista de *n* números enteros consecutivos que empieza en 0 y acaba en *n* – 1. Para poder ver los valores creados dentro del rango, se ha de convertir el rango en lista y mostrarla.

  Si *n* no es positivo, se crea un *range* vacío.

```
valor = range(10)
print(valor)
print(list(valor))
print(range(7))
print(list(range(-2)))
print(list(range(0)))
```

Certificación

Es importante interiorizar los tipos de datos complejos, pues en numerosas preguntas de la certificación se hace referencia a estos.

Salida:

```
range(0, 10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
range(0, 7)
[0, 1, 2, 3, 4, 5, 6]
[]
[]
```

- **"range(m, n)" con dos argumentos:** se crea una lista de enteros consecutivos que empieza en *m* y acaba en *n* – 1.

```
print(list(range(5, 10)))
print(list(range(-5, 1)))
print(list(range(5, 1)))
```

Salida:

```
[5, 6, 7, 8, 9]
[-5, -4, -3, -2, -1, 0]
[]
```

- **"range(m, n, p)" con tres argumentos:** se crea una lista de enteros consecutivos que empieza en *m* y acaba justo antes de superar o igualar a *n*, aumentando los valores de *p* en *p*. Si p es negativo, los valores van disminuyendo de *p* en *p*. El valor de *p* no puede ser cero:

```
print(list(range(5, 21, 3)))
print(list(range(10, 0, -2)))
print(list(range(4, 18, 0)))
```

Salida:

```
[5, 8, 11, 14, 17, 20]
[10, 8, 6, 4, 2]

ValueError
Traceback (most recent call last)
<ipython-input-92-64dcb2f33848> in <module>()
      1 print(list(range(5, 21, 3)))
      2 print(list(range(10, 0, -2)))
----> 3 print(list(range(4, 18, 0)))

ValueError: range() arg 3 must not be zero
```

Si *p* es positivo y *n* menor o igual que *m*, o si *p* es negativo y *n* mayor o igual que *m*, se crea un *range* vacío.

```
print(list(range(25, 20, 2)))
print(list(range(20, 25, -2)))
```

Salida:

```
[]
[]
```

En los "range(m, n, p)" se pueden escribir *p* rangos distintos que generan el mismo resultado. Por ejemplo:

```
print(list(range(10, 20, 3)))
print(list(range(10, 21, 3)))
print(list(range(10, 22, 3)))
```

Salida:

```
[10, 13, 16, 19]
[10, 13, 16, 19]
[10, 13, 16, 19]
```

Por resumir cada definición del tipo "range(m, n, p)":

- ***m*:** el valor inicial.
- ***n*:** el valor final (que no se alcanza nunca).
- ***p*:** el paso (la cantidad que se avanza cada vez).

Hay un par de asunciones: la primera de ellas es que, por ejemplo, en la definición con dos argumentos, Python asume que $p=1$, por lo que los valores aumentarán de 1 en 1. Por otra parte, si solo se escribe un argumento, Python sigue asumiendo que $p=1$, pero ahora también asume que $m=0$, por lo que el rango empezará en 0 y aumentará de 1 en 1.

Este tipo de datos solo admite argumentos enteros; en cualquier otro caso, se producirá un error.

```
print(range(3.5, 20, 3))
```

Salida:

```
TypeError                Traceback (most recent call last)
<ipython-input-95-d6676a62726b> in <module>()
----> 1 print(range(3.5, 20, 3))

TypeError: 'float' object cannot be interpreted as an integer
```

De forma directa, no existe la posibilidad de concatenar varios tipos de "range()", ya que no se puede asegurar que el resultado vaya a cumplir las especificaciones del tipo de datos "range()".

```
print(range(3)+range(5))
```

Salida:

```
TypeError          Traceback (most recent call last)
<ipython-input-96-e02e3f3c4fe2> in <module>()
----> 1 print(range(3)+range(5))

TypeError: unsupported operand type(s) for +: 'range' and 'range'
```

Sin embargo, de manera indirecta sí podremos concatenar o unir varios tipos de datos "range()", siempre y cuando previamente los convirtamos en listas. Lógicamente, el resultado será una lista; si bien no es un tipo "range()" ni se podrá convertir en él, al menos habremos podido unir dos "range()".

```
print(list(range(3))+list(range(5)))
print(list(range(10, 0, -2)))
print(list(range(4, 18, 0)))
```

Salida:

```
[0, 1, 2, 0, 1, 2, 3, 4]
```

Por otro lado, aunque nos asegurásemos de que estamos intentando concatenar dos tipos “range()” iguales en cuanto a tipo de sucesión, seguiría sin ser posible.

```
print(list(range(1, 3))+list(range(3, 5)))
print(list(range(10, 0, -2)))
print(list(range(4, 18, 0)))
```

Salida:

```
   [1, 2, 3, 4]
...
TypeError        Traceback (most recent call last)
<ipython-input-98-adeda5806ed8> in <module>()
      1 print(list(range(1, 3))+list(range(3, 5)))
----> 2 print(range(1, 3)+range(3, 5))

TypeError: unsupported operand type(s) for +: ‘range’ and ‘range’
```

Cuaderno Colab

Practica **este ejemplo** en Colab con los conocimientos adquiridos hasta esta unidad.

4.2. Tuplas

Las **tuplas** son conjuntos de datos inmutables. Esto significa que no pueden modificarse una vez que han sido creadas. Son muy similares a las listas y comparten varias de sus funciones y métodos integrados, y su principal diferencia respecto a estas es que las tuplas son inmutables. Algunos de los métodos integrados con los que cuentan las tuplas y que podemos usar para acceder a sus datos son los siguientes:

4.2.1. “Count()”

A este método se le pasa un elemento como argumento y su función es la de contar la cantidad de veces que este aparece en la tupla.

```
Frutas = (“Platano”, “Pera”, “Manzana”, “Pera”)
print(Frutas.count(“Kiwi”))
print(Frutas.count(“Pera”))
```

Salida:

```
0
2
```

4.2.2. "Index()"

Es el mismo método que usa el tipo de dato *lista*; y su función, tras recibir un elemento como argumento, es devolver el índice de la primera aparición de este elemento en la tupla.

El método devuelve una excepción "ValueError" si el elemento no se encuentra en la tupla.

```
Frutas = ("Platano", "Pera", "Manzana", "Pera")
print(Frutas.index("Pera"))
print(Frutas.index("Kiwi"))
```

Salida:

```
1
ValueError
Traceback (most recent call last)
<ipython-input-100-58e24b6f8535> in <module>()
      1 Frutas = ("Platano", "Pera", "Manzana", "Pera")
      2 print(Frutas.index("Pera"))
----> 3 print(Frutas.index("Kiwi"))

ValueError: tuple.index(x): x not in tuple
```

4.3. Listas

En Python, el tipo de datos **lista** se podría definir como una variable que almacena *arrays*, por lo que, internamente, cada posición puede ser un tipo de datos distinto.

```
Frutas = ("Platano", "Pera", "Manzana", "Pera")
print(Frutas)
```

Salida:

```
['Platano', 'Pera', 'Manzana', 'Pera']
```

Certificación

Es importante interiorizar las listas, pues en numerosas preguntas de la certificación se hace referencia a estas.

Las listas en Python se definen por dos características muy claras:

- **Heterogéneas:** pueden estar conformadas por elementos de distintos tipos, incluidas otras listas.

- **Mutables:** sus elementos pueden modificarse.

Puede accederse a los elementos de una lista mediante su índice, y el índice del primer elemento es 0.

```
Frutas = (“Platano”, “Pera”, “Manzana”, “Pera”)
print(Frutas[0])
print(Frutas[3])
```

Salida:

```
Platano
```

Para saber la longitud de una lista, utilizamos una función integrada llamada “**len()**”.

```
Frutas = (“Platano”, “Pera”, “Manzana”, “Pera”)
print(len(Frutas))
```

Salida:

```
4
```

Las posiciones de una lista empiezan por 0 y acaban en el tamaño de la lista menos uno: (len(factura) - 1):

```
Frutas = (“Platano”, “Pera”, “Manzana”, “Pera”)
print(len(Frutas)-1)
```

Salida:

```
3
```

Se pueden usar de igual modo índices negativos; el índice –1 se refiere al último elemento de la lista.

```
Frutas = (“Platano”, “Pera”, “Manzana”, “Pera”)
print(Frutas[-1])
```

Salida:

```
3
```

De esta definición, podemos asumir que los índices negativos que se pueden usar van desde el -1 (último elemento) a -len(factura) (primer elemento).

```
Frutas = (“Platano”, “Pera”, “Manzana”, “Pera”)
print(Frutas[-len(Frutas)])
```

Salida:

```
Platano
```

Mediante el acceso a un índice o posición concreta, podemos asignar un valor nuevo; por eso las listas son tipos de datos mutables, a diferencia de las tuplas.

Cuaderno Colab

Practica **este ejemplo** en Colab con los conocimientos adquiridos hasta esta unidad.

```
Frutas = (“Platano”, “Pera”, “Manzana”, “Pera”)
print(Frutas)
Frutas[3] = “Kiwi”
print(Frutas)
```

Salida:

```
[‘Platano’, ‘Pera’, ‘Manzana’, ‘Pera’]
[‘Platano’, ‘Pera’, ‘Manzana’, ‘Kiwi’]
```

4.3.1. Métodos integrados

A continuación, enumeramos los métodos integrados:

- **“append()”:** este método agrega un elemento al final de una lista.

```
Frutas = (“Platano”, “Pera”, “Manzana”, “Pera”)
print(Frutas)
Frutas.append(“Kiwi”)
print(Frutas)
```

Salida:

```
[‘Platano’, ‘Pera’, ‘Manzana’, ‘Pera’]
[‘Platano’, ‘Pera’, ‘Manzana’, ‘Pera’, ‘Kiwi’]
```

- **“count()”:** de igual manera que en las tuplas, este método recibe un elemento como argumento y cuenta la cantidad de veces que aparece en la lista.

```
Frutas = (“Platano”, “Pera”, “Manzana”, “Pera”)
print(Frutas)
print(Frutas.count(“Pera”))
```

Salida:

```
[‘Platano’, ‘Pera’, ‘Manzana’, ‘Pera’]
2
```

- **“extend()”:** este método extiende una lista agregando un iterable al final.

```
Frutas = (“Platano”, “Pera”, “Manzana”, “Pera”)
print(Frutas)
Frutas.extend(“Kiwi”)
print(Frutas)
```

Salida:

```
[‘Platano’, ‘Pera’, ‘Manzana’, ‘Pera’]
[‘Platano’, ‘Pera’, ‘Manzana’, ‘Pera’, ‘K’, ‘i’, ‘w’, ‘i’]
```

- **“index()”:** de la misma manera que en las tuplas, este método recibe un elemento como argumento y devuelve el índice de su primera aparición en la lista.

  Adicionalmente, esta función admite otro argumento que indica el índice inicial a partir del cual empezar la búsqueda.

El método devuelve una excepción “ValueError” si el elemento no se encuentra en la lista.

```
versiones = [1.0, 3.2, 4, 5, 6.5]
print(versiones.index(4))
print(versiones[2])
print(versiones.index(4, 2))
print(versiones.index(19))
```

Salida:

```
2
4
2
ValueError        Traceback (most recent call last)
<ipython-input-114-bae08137e986> in <module>()
      3 print(versiones[2])
      4 print(versiones.index(4, 2))
----> 5 print(versiones.index(19))

ValueError: 19 is not in list
```

- **“insert()”:** a esta función se le pasan dos argumentos con la forma “insert(i, x)” y lo que hace es insertar el elemento *x* en la posición *i*.

```
versiones = [1.0, 3.2, 4, 5, 6.5]
print(versiones)
versiones.insert(2, -5)
print(versiones)
```

Salida:

```
[1.0, 3.2, 4, 5, 6.5]
[1.0, 3.2, -5, 4, 5, 6.5]
```

- **“pop()”:** esta función es similar a la de desapilar si trabajásemos con pilas: retorna el último elemento de la lista y lo borra.

```
versiones = [1.0, 3.2, 4, 5, 6.5]
print(versiones. pop())
print(versiones)
```

Para saber más

Una pila (*stack* en inglés) es una lista ordenada o estructura de datos que permite almacenar y recuperar datos, siendo el modo de acceso a sus elementos de tipo LIFO (del inglés *Last In, FirstOut*, «último en entrar, primero en salir»).

Salida:

```
6.5
[1.0, 3.2, 4, 5]
```

Opcionalmente, se le puede pasar un argumento numérico que haría las veces de índice para devolver y borrar de la lista ese elemento en concreto. En este caso, no habría ninguna similitud con las pilas.

```
versiones = [1.0, 3.2, 4, 5, 6.5]
print(versiones.pop(2))
print(versiones)
```

Salida:

```
4
[1.0, 3.2, 5, 6.5]
```

- **"remove()":** a este método se le pasa como argumento un valor; busca su primera aparición en la lista y la borra.

  El método devuelve una excepción "ValueError" si el elemento no se encuentra en la lista.

```
versiones = [1.0, 3.2, 4, 5, 6.5]
print(versiones)
versiones.remove(3.2)
print(versiones)
versiones.remove(12)
```

Salida:

```
4
[1.0, 3.2, 5, 6.5]
```

- **"reverse()":** este método invierte el orden de los elementos en la lista; es decir, ordena de forma inversa los elementos en función de su índice o su posición.

```
versiones = [1.0, 3.2, 4, 5, 6.5]
print(versiones)
versiones.reverse()
print(versiones)
```

Salida:

```
[1.0, 3.2, 4, 5, 6.5]
[6.5, 5, 4, 3.2, 1.0]
```

- **"sort()":** este método ordena los elementos de una lista.

El método "sort()" admite la opción "reverse", aunque por defecto sea "False"; esto es, hay que especificar el valor "True" en la definición para invertir la ordenación.

```
versiones = [1.0, 3.2, 4, 5, 6.5]
print(versiones)
versiones.sort()
print(versiones)
versiones.sort(reverse=True)
print(versiones)
```

Cuaderno Colab

Practica **este ejemplo** en Colab con los conocimientos adquiridos hasta esta unidad.

Salida:

```
[1.0, 3.2, 8, 5, 6.5]
[1.0, 3.2, 5, 6.5, 8]
[8, 6.5, 5, 3.2, 1.0]
```

4.4. Diccionarios

Los **diccionarios** definen una relación uno a uno entre una serie de claves y sus respectivos valores.

Los diccionarios se pueden definir colocando una lista de parejas de datos separadas por comas al estilo "key:value", y todo ello entre llaves ("{}").

```
>>>diccionario={
... "clave1":42,
... "clave2":GDAG,
```

Se puede acceder a los valores del diccionario usando la clave pertinente del valor que se quiera saber.

Al poder almacenar distintos tipos de datos, podemos saber de qué tipo es cada dato pasándole a la función "type" la clave del diccionario.

```
dic = {
    “Nombre”: “Josep”,
    “Tel”: 688777555,
    “Material”: [“PC”, “Teclado”, “WebCam”]
}
print(dic, type(dic))
print(dic[“Nombre”])
print(type(dic[“Material”]))
```

Salida:

```
{‘Nombre’: ‘Josep’, ‘Tel’: 688777555, ‘Material’: [‘PC’, ‘Teclado’, ‘WebCam’]} <class ‘dict’>
Josep
<class ‘list’>
```

4.4.1. Operaciones

Los objetos de tipo **diccionario** permiten un gran número de operaciones sobre ellos mediante el uso de operadores integrados en el intérprete de Python.

- **Acceder a valor de clave:** permite acceder a un valor especifico mediante su clave.

```
dic = dict(
    Nombre=”Josep”,
    Tel=688777555,
    Material=[“PC”, “Teclado”, “WebCam”]
)

print(dic[“Nombre”])
```

Salida:

```
Josep
```

- **Asignar valor a clave:** permite asignar el valor específico del diccionario mediante su clave.

```
dic = dict(
```

Ejercicios resueltos

Practica el **ejercicio 4** con los conocimientos adquiridos hasta esta unidad.

```
    Nombre="Josep",
    Tel=688777555,
    Material=["PC", "Teclado", "WebCam"]
)
print(dic, type(dic))
dic["Nombre"] = "Claudio"
print(dic, type(dic))
```

Salida:

```
{'Nombre': 'Josep', 'Tel': 688777555, 'Material': ['PC', 'Teclado', 'WebCam']} <class 'dict'>
{'Nombre': 'Claudio', 'Tel': 688777555, 'Material': ['PC', 'Teclado', 'WebCam']} <class 'dict'>
```

- **Iteración "in":** con este operador obtendremos un valor "True" si el valor por el que preguntamos se encuentra en el diccionario.

```
dic = dict(
    Nombre="Josep",
    Tel=688777555,
    Material=["PC", "Teclado", "WebCam"]
)
print(dic, type(dic))
print('Nombre' in dic)
print('Apellidos' in dic)
```

Cuaderno Colab

Practica **este ejemplo** en Colab con los conocimientos adquiridos hasta esta unidad.

Salida:

```
{'Nombre': 'Josep', 'Tel': 688777555, 'Material': ['PC', 'Teclado', 'WebCam']} <class 'dict'>
True
False
```

4.4.2. Métodos integrados

- **"clear()":** borra todos los valores del diccionario.

```
dic = dict(
    Nombre="Josep",
    Tel=688777555,
    Material=["PC", "Teclado", "WebCam"]
)
print(dic, type(dic))
dic.clear()
print(dic, type(dic))
```

Salida:

```
{'Nombre': 'Josep', 'Tel': 688777555, 'Material': ['PC', 'Teclado', 'WebCam']} <class
'dict'>
{} <class 'dict'>
```

- **"copy()"**: replica el objeto diccionario.

```
dic = dict(
    Nombre="Josep",
    Tel=688777555,
    Material=["PC", "Teclado", "WebCam"]
)
dic2 = dict()
print(dic, dic == dic2)
dic2 = dic.copy()
print(dic2, dic == dic2)
```

Salida:

```
{'Nombre': 'Josep', 'Tel': 688777555, 'Material': ['PC', 'Teclado', 'WebCam']} False
{'Nombre': 'Josep', 'Tel': 688777555, 'Material': ['PC', 'Teclado', 'WebCam']} True
```

- **"fromkeys()":** en ocasiones es necesario generar un diccionario a partir de las claves proporcionadas. Su implementación a mano nos tomaría mucho tiempo y sería un trabajo muy tedioso; esta función nos ayuda a realizar esta tarea de forma sencilla y utilizando un solo método.

También se puede inicializar en un valor en concreto.

```
variables = ('Nombre', 'Tel', 'Material')
dic = dict.fromkeys(variables)
print(dic)
dic = dict.fromkeys(variables, "Rellenar")
print(dic)
```

Salida:

```
{'Nombre': None, 'Tel': None, 'Material': None}
{'Nombre': 'Rellenar', 'Tel': 'Rellenar', 'Material': 'Rellenar'}
```

- **“get()”:** nos devuelve el valor en función de la clave que busquemos.

```
dic = dict(
    Nombre=”Josep”,
    Tel=688777555,
    Material=[“PC”, “Teclado”, “WebCam”]
)
print(dic.get(“Nombre”))
```

Salida:

```
Josep
```

- **“items()”:** este método devuelve una lista de pares de diccionarios (clave, valor) como dos tuplas.

```
dic = dict(
    Nombre=”Josep”,
    Tel=688777555,
    Material=[“PC”, “Teclado”, “WebCam”]
)
print(dic.items())
```

Salida:

```
dict_items([(‘Nombre’, ‘Josep’), (‘Tel’, 688777555), (‘Material’,
[‘PC’, ‘Teclado’, ‘WebCam’])])
```

- **“keys()”**: este método devuelve una lista de las claves del diccionario:

```
dic = dict(
    Nombre=”Josep”,
    Tel=688777555,
    Material=[“PC”, “Teclado”, “WebCam”]
)
print(dic.keys())
```

Salida:

```
dict_keys([‘Nombre’, ‘Tel’, ‘Material’])
```

- **“pop()”**: este método nos muestra el valor de una clave concreta que le especifiquemos como argumento y la borra del diccionario, junto con su valor. Lanza una excepción “KeyError” si la clave no se encuentra.

```
dic = dict(
    Nombre=”Josep”,
    Tel=688777555,
    Material=[“PC”, “Teclado”, “WebCam”]
)
print(dic.pop(‘Nombre’))
print(dic.items())
```

Salida:

```
Josep
dict_items([(‘Tel’, 688777555), (‘Material’, [‘PC’, ‘Teclado’, ‘WebCam’])])
```

- **“popitem()”**: este método muestra la clave y el valor del primer elemento del diccionario y lo borra. Lanza una excepción “KeyError” si el diccionario está vacío.

```
dic = dict(
    Nombre=”Josep”,
    Tel=688777555,
    Material=[“PC”, “Teclado”, “WebCam”]
)
print(dic.popitem())
print(dic.items())
print(dic.popitem())
print(dic.items())
print(dic.popitem())
print(dic.items())
print(dic.popitem())
print(dic.items())
```

Salida:

```
(‘Material’, [‘PC’, ‘Teclado’, ‘WebCam’])
dict_items([(‘Nombre’, ‘Josep’), (‘Tel’, 688777555)])
(‘Tel’, 688777555)
dict_items([(‘Nombre’, ‘Josep’)])
(‘Nombre’, ‘Josep’)
dict_items([])
```

```
KeyError        Traceback (most recent call last)
<ipython-input-132-fece31d59e4e> in <module>()
     10 print(dic.popitem())
     11 print(dic.items())
---> 12 print(dic.popitem())
     13 print(dic.items())

KeyError: 'popitem(): dictionary is empty'
```

- **"setdefault()":** este método es similar a "get(key, default_value)", pero asigna además la clave "key" al valor por "default_value" para la clave si esta no se encuentra en el diccionario.

```
dic = dict(
    Nombre="Josep",
    Tel=688777555,
    Material=["PC", "Teclado", "WebCam"]
)
nombre = dic.setdefault('Nombre')
print(nombre)
```

Salida:

```
Josep
```

Por otro lado, también podemos usar "setdefault()" si la clave no está en el diccionario:

```
dic = dict(
    Nombre="Josep",
    Tel=688777555,
    Material=["PC", "Teclado", "WebCam"]
)
dic.setdefault('Apellidos')
print(nombre)
```

Salida:

```
{'Nombre': 'Josep', 'Tel': 688777555, 'Material': ['PC', 'Teclado',
'WebCam'], 'Apellidos': None}
```

- **“update()”**: este método actualiza un diccionario añadiendo parejas de clave/valor en un diccionario paralelo. Como tal, este método solo actualiza, pero no retorna nada; sin embargo, sí nos muestra de nuevo el diccionario “versiones” y podemos ver que este fue actualizado con el otro diccionario “versiones_adicional”.

```
dic = dict(
    Nombre=”Claudio”,
    Tel=688777555,
    Material=[“PC”, “Teclado”, “WebCam”]
)
dic_extra = dict(
    Apellidos=”Gomez”
)
print(dic)
print(dic_extra)
dic.update(dic_extra)
print(dic)
```

Salida:

```
{‘Nombre’: ‘Claudio’, ‘Tel’: 688777555, ‘Material’: [‘PC’, ‘Tecla-
do’, ‘WebCam’]}
{‘Apellidos’: ‘Gomez’}
{‘Nombre’: ‘Claudio’, ‘Tel’: 688777555, ‘Material’: [‘PC’, ‘Tecla-
do’, ‘WebCam’], ‘Apellidos’: ‘Gomez’}
```

- **“values()”**: este método devuelve una lista de los valores del diccionario:

```
dic = dict(
    Nombre=”Claudio”,
    Tel=688777555,
    Material=[“PC”, “Teclado”, “WebCam”]
)
print(dic.values())
```

Salida:

```
dict_values([‘Claudio’, 688777555, [‘PC’, ‘Teclado’, ‘WebCam’]])
```

Para saber más

Para convertir a tipo *diccionario*, debe usarse la función “dict()”, que está integrada en el intérprete de Python.

Cuaderno Colab

Practica **este ejemplo** en Colab con los conocimientos adquiridos hasta esta unidad.

Ejercicios resueltos

Practica **los ejercicios de autocomprobación** en Colab con los conocimientos adquiridos en este módulo.

Resumen

- Un rango se define como una lista inalterable de números enteros en sucesión aritmética. Así, la diferencia entre los diferentes valores será siempre la misma. Se define con el tipo de datos *range*.

- Las tuplas son conjuntos de datos inmutables. Esto significa que no pueden modificarse una vez que han sido creados. Son muy similares a las listas y comparten varias de sus funciones y métodos integrados, mientras que su principal diferencia es que son inmutables.

- En Python, el tipo de datos lista se podría definir como una variable que almacena *arrays*, por lo que internamente cada posición puede ser un tipo de datos distinto.

- Los diccionarios definen una relación uno a uno entre una serie de claves y sus respectivos valores.

- Los diccionarios se pueden definir colocando una lista de parejas de datos separadas por coma al estilo “key:value”, y todo ello entre llaves (“{}”).

- Tanto las listas como los diccionarios tienen funciones integradas para que recorrerlos y realizar operaciones con ellos sea más sencillo.

Índice