CUADERNO DE EJERCICIOS DE PYTHON

Este es el cuaderno de Ejercicios de uso de python desde el CEC-EPN

USO DE OPERADORES MATEMATICOS

Las cuatro operaciones básicas Las cuatro operaciones aritméticas básicas son la suma (+), la resta (-), la multiplicación (*) y la división (/).

Al hacer operaciones en las que intervienen números enteros y decimales, el resultado es siempre decimal. En el caso de que el resultado no tenga parte decimal, Python escribe 0 como parte decimal para indicar que el resultado es un número decimal:

In [2]:

```
#SUMA

12365+1204578
```

Out[2]:

```
1216943
```

In [3]:

```
#SUMA

55987425+11103354877
```

Out[3]:

```
11159342302
```

In [4]:

```
#MULTIPLICACION

8549*1994
```

Out[4]:

```
17046706
```

In [5]:

```
#MULTIPLICACION

6521*2012
```

Out[5]:

```
13120252
```

In [6]:

```
#NUMEROS COMPLEJOS

1 + 1j + 2 + 3j
```

Out[6]:

```
(3+4j)
```

In [7]:

```
#NUMEROS COMPLEJOS

(1+1j) * 1j
```

Out[7]:

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

In [8]:

```
#MULTIPLICACION NUMEROS ENTEROS Y DECIMALES

4.5 * 3
```

Out[8]:

```
13.5
```

In [10]:

```
#MULTIPLICACION NUMEROS ENTEROS Y DECIMALES

8.9 * 2
```

Out[10]:

```
17.8
```

In [11]:

```
#RESTA

9841 - 10997
```

Out[11]:

```
-1156
```

In [12]:

```
#RESTA

11983 - 9961
```

Out[12]:

```
2022
```

In [5]:

```
#DIVISION

#Al dividir números enteros, el resultado es siempre decimal,
#aunque sea un número entero.
#Cuando Python escribe un número decimal, lo escribe siempre,
#con parte decimal, aunque sea nula.

9 / 2
```

Out[5]:

```
4.5
```

In [14]:

```
#DIVISION

9 / 3
```

Out[14]:

```
3.0
```

In [15]:

```
#NUMEROS COMPLEJOS

#El resultado de una operación en la que intervienen números,
#complejos es un número complejo,
#aunque el resultado no tenga parte imaginaria.

1j * 1j
```

Out[15]:

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

In [17]:

```
#Cuando en una fórmula aparecen varias operaciones,
#Python las efectúa aplicando,
#las reglas usuales de prioridad de las operaciones
#(primero multiplicaciones y divisiones, después sumas y restas).

#EJERCICIO#1

1 + 2 * 3
```

Out[17]:

```
7
```

In [18]:

```
#EJERCICIO#2

10 - 4 * 2
```

Out[18]:

```
2
```

In [19]:

```
#En caso de querer que las operaciones se realicen en otro orden,
#se deben utilizar paréntesis.

(5 + 8) / (7 - 2)
```

Out[19]:

```
2.6
```

EJEMPLOS DE ERRORES EN PYTHON

In [20]:

```
#Dividir por cero genera un error:

5 / 0
```

```
---------------------------------------------------------------------------
--
ZeroDivisionError                         Traceback (most recent call las
t)
~\AppData\Local\Temp\ipykernel_6368\2331073613.py in <module>
      1 #Dividir por cero genera un error:
----> 2 5 / 0

ZeroDivisionError: division by zero
```

In [23]:

```
#Al realizar operaciones con decimales, los resultados pueden,
#presentar errores de redondeo:

100 / 3
```

Out[23]:

```
33.333333333333336
```

In [24]:

```
#Debido a los errores de redondeo, dos operaciones que
#debieran dar el mismo resultado pueden dar resultados diferentes:

#EJEMPLO#1

4 * 3 / 5
```

Out[24]:

```
2.4
```

In [25]:

```
#EJEMPLO#1 CON NUMEROS CAMBIADOS DE POSICION

4 / 5 * 3
```

Out[25]:

```
2.4000000000000004
```

In [26]:

```
#Se pueden escribir sumas y restas seguidas, pero no se recomienda,
#hacerlo porque no es una notación habitual:

#EJEMPLO#1

3 + - + 4
```

Out[26]:

```
-1
```

In [27]:

```
#EJEMPLO#2

3 - + - + 4
```

Out[27]:

```
7
```

In [28]:

```
#Lo que no se puede hacer es escribir multiplicaciones y divisiones seguidas:

3 * / 4
```

```
  File "C:\Users\Kevin\AppData\Local\Temp\ipykernel_6368\1224859286.py",
line 2
    3 * / 4
        ^
SyntaxError: invalid syntax
```

Cociente de una división

In [29]:

```
#El cociente de una división se calcula en Python con el operador //.
#El resultado es siempre un número entero, pero será de tipo entero o decimal
#dependiendo del tipo de los números empleados
#(en caso de ser decimal, la parte decimal es siempre cero).
#Por ejemplo:

#EJEMPLO#1

10 // 3
```

Out[29]:

```
3
```

In [30]:

```
#EJEMPLO#2

20 // 6.0
```

Out[30]:

```
3.0
```

In [31]:

```
#El operador cociente // tiene la misma prioridad que la división:

#EJEMPLO#1

26 // 5 / 2
```

Out[31]:

```
2.5
```

In [32]:

```
#EJEMPLO#2

(26 // 5) / 2
```

Out[32]:

```
2.5
```

In [33]:

```
#EJEMPLO#3

26 // (5 / 2)
```

Out[33]:

```
10.0
```

In [34]:

```
#EJEMPLO#4

26 / 5 // 2
```

Out[34]:

```
2.0
```

In [35]:

```
#EJEMPLO#5

(26 / 5) // 2
```

Out[35]:

```
2.0
```

In [36]:

```
#EJEMPLO#6

26 / (5 // 2)
```

Out[36]:

```
13.0
```

Resto de una división

In [37]:

```
#El resto de una división se calcula en Python con el operador %.
#El resultado tendrá tipo entero o decimal, de acuerdo,
#con el resultado de la operación.

#EJEMPLO#1

10 % 3
```

Out[37]:

```
1
```

In [38]:

```
#EJEMPLO#2

10 % 4
```

Out[38]:

```
2
```

In [39]:

```
#EJEMPLO#3

10 % 5
```

Out[39]:

```
0
```

In [40]:

```
#EJEMPLO#4

10.5 % 3
```

Out[40]:

```
1.5
```

In [42]:

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

#EJEMPLO#1

10.2 % 3             #EL RESULTADO CORRECTO ES 1.2
```

Out[42]:

```
1.1999999999999993
```

In [43]:

```
#EJEMPLO#2

10 % 4.2            #EL RESULTADO CORRECTO ES 1.6
```

Out[43]:

```
1.5999999999999996
```

In [44]:

```
#EJEMPLO#3

10.1 % 5.1         #ESTE RESULTADO COINCIDE CON EL RESULTADO CORRECTO
```

Out[44]:

```
5.0
```

El operador resto % tiene la misma prioridad que la división:

In [45]:

```
#EJEMPLO#1

26 % 5 / 2
```

Out[45]:

```
0.5
```

In [46]:

```
#EJEMPLO#2

(26 % 5) / 2
```

Out[46]:

```
0.5
```

In [47]:

```
#EJEMPLO#3

26 % (5 / 2)
```

Out[47]:

```
1.0
```

In [48]:

```
#EJEMPLO#4

26 / 5 % 2
```

Out[48]:

```
1.2000000000000002
```

In [49]:

```
#EJEMPLO#5

(26 / 5) % 2
```

Out[49]:

```
1.2000000000000002
```

In [50]:

```
#EJEMPLO#6

26 / (5 % 2)
```

Out[50]:

```
26.0
```

La función integrada divmod()

In [51]:

```
#La función integrada divmod(x, y) devuelve una tupla formada
#por el cociente y el resto de la división de x entre y.

divmod(13, 4)
```

Out[51]:

```
(3, 1)
```

También se pueden calcular potencias o raíces mediante la función integrada pow(x,y). Si se da un tercer argumento, pow(x, y, z), la función calcula primero x elevado a y después calcula el resto de la división por z.

In [52]:

```
#EJEMPLO#1

pow(2, 3)
```

Out[52]:

```
8
```

In [53]:

```
#EJEMPLO#2

pow(4, 0.5)
```

Out[53]:

```
2.0
```

In [54]:

```
#EJEMPLO#3

pow(2, 3, 5)
```

Out[54]:

```
3
```

Redondear un número

Aunque no se puede dar una regla válida en todas las situaciones, normalmente es conveniente redondear el resultado de un cálculo cuando se muestra al usuario, sobre todo si tiene muchos decimales, para facilitar su lectura.

Lo que no se debe hacer nunca es redondear resultados intermedios que se vayan a utilizar en cálculos posteriores, porque el resultado final será diferente.

La función integrada round()

Para redondear un número (por ejemplo, cuando se muestra al usuario el resultado final de un cálculo), se puede utilizar la función integrada round(). La función integrada round() admite uno o dos argumentos numéricos.

In [55]:

```
#Si sólo hay un argumento, la función devuelve el argumento,
#redondeado al entero más próximo:

#EJEMPLO#1

round(4.35)
```

Out[55]:

```
4
```

In [56]:

```
#EJEMPLO#2

round(-4.35)
```

Out[56]:

```
-4
```

Si se escriben dos argumentos, siendo el segundo un número entero, la función integrada round() devuelve el primer argumento redondeado en la posición indicada por el segundo argumento.

In [57]:

```
#Si el segundo argumento es positivo, el primer argumento se,
#redondea con el número de decimales indicado:

#EJEMPLO#1

round(4.3527, 2)
```

Out[57]:

```
4.35
```

In [58]:

```
#EJEMPLO#2

round(4.3527, 1)
```

Out[58]:

```
4.4
```

In [59]:

```
#EJEMPLO#3

round(4.3527, 3)
```

Out[59]:

```
4.353
```

In [60]:

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

#EJEMPLO#1

round(4.3527, 7)
```

Out[60]:

```
4.3527
```

In [61]:

```
#EJEMPLO#2

round(435, 2)
```

Out[61]:

```
435
```

In [62]:

```
#Si el segundo argumento es 0 y el primero es un número decimal,
#se redondea al entero más próximo, como cuando no se,
#escribe segundo argumento,
#pero la diferencia es que el resultado es decimal y no entero.

#EJEMPLO#1

round(4.3527, 0)
```

Out[62]:

```
4.0
```

In [63]:

```
#EJEMPLO#2

round(4.3527)
```

Out[63]:

```
4
```

In [64]:

```
#Si el segundo argumento es 0 y el primero es un número entero,
#el resultado es entero:

#EJEMPLO

round(435, 0)
```

Out[64]:

```
435
```

In [65]:

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

#EJEMPLO#1

round(43527, -1)
```

Out[65]:

```
43530
```

In [66]:

```
#EJEMPLO#2

round(43527, -2)
```

Out[66]:

```
43500
```

In [67]:

```
#EJEMPLO#3

round(43527, -3)
```

Out[67]:

```
44000
```

In [68]:

```
#EJEMPLO#4

round(43527, -4)
```

Out[68]:

```
40000
```

In [69]:

```
#EJEMPLO#5

round(43527, -5)
```

Out[69]:

```
0
```

La función integrada round() redondea correctamente al número más próximo del orden de magnitud deseado (entero, décimas, decenas, centésimas, centenas, etc). El problema es cuando el número a redondear está justo en medio (por ejemplo, redondear 3.5 a entero, 4.85 a décimas, etc.). En Matemáticas se suele redondear siempre hacia arriba, pero en Python se sigue otro criterio:

In [70]:

```
#Cuando se redondea a enteros, decenas, centenas, etc.,
#Python redondea de manera que la última cifra (la redondeada) sea par.

#EJEMPLO#1

round(3.5)
```

Out[70]:

```
4
```

In [71]:

```
#EJEMPLO#2

round(4.5)
```

Out[71]:

```
4
```

In [72]:

```
#EJEMPLO#3

round(5.5)
```

Out[72]:

```
6
```

In [73]:

```
#EJEMPLO#4

round(6.5)
```

Out[73]:

```
6
```

In [74]:

```
#EJEMPLO#5

round(450, -2)
```

Out[74]:

```
400
```

In [75]:

```
#EJEMPLO#6

round(350, -2)
```

Out[75]:

```
400
```

In [76]:

```
#EJEMPLO#7

round(250, -2)
```

Out[76]:

```
200
```

In [77]:

```
#EJEMPLO#8

round(150, -2)
```

Out[77]:

```
200
```

In [78]:

```
#EJEMPLO#9

round(50, -2)
```

Out[78]:

```
0
```

Cuando se redondea a décimas, centésimas, etc., Python redondea en unos casos para arriba y en otros para abajo, debido a la forma en que se representan internamente los números decimales (como se explica en el apartado Representación de números decimales en binario):

In [79]:

```
#EJEMPLO#1

round(3.15, 1)
```

Out[79]:

```
3.1
```

In [80]:

```
#EJEMPLO#2

round(3.45, 1)
```

Out[80]:

```
3.5
```

In [81]:

```
#EJEMPLO#3

round(3.65, 1)
```

Out[81]:

```
3.6
```

In [82]:

```
#EJEMPLO#4

round(0.315, 2)
```

Out[82]:

```
0.32
```

In [83]:

```
#EJEMPLO#5

round(0.345, 2)
```

Out[83]:

```
0.34
```

In [84]:

```
#EJEMPLO#6

round(0.365, 2)
```

Out[84]:

```
0.36
```

Representación de números decimales en binario.

En Python los números decimales se almacenan internamente en binario con 53 bits de precisión (en concreto, se trata del formato de coma flotante de doble precisión de la norma IEEE-754). Cuando un programa pide a Python un cálculo con números decimales, Python convierte esos números decimales a binario, realiza la operación en binario y convierte el resultado de nuevo en decimal para mostrárselo al usuario.

El problema es que muchos números decimales no se pueden representarse de forma exacta en binario, por lo que los resultados no pueden ser exactos. Eso explica, por ejemplo, los resultados del ejemplo anterior:

In [85]:

```
#EJEMPLO#1

round(3.45, 1)
```

Out[85]:

```
3.5
```

In [86]:

```
#EJEMPLO#2

round(3.55, 1)
```

Out[86]:

```
3.5
```

En el primer ejemplo, al convertir 3.45 a binario con 53 bits de precisión, el valor obtenido es realmente 3.450000000000000177635683940025046467781066894531 25, es decir, ligeramente mayor que 3.45, por lo que al redondear con décimas, Python muestra el valor 3.5.

En el segundo ejemplo, al convertir 3.55 a binario con 53 bits de precisión, el valor obtenido es realmente 3.549999999999999822364316059974953532218933105468 75, es decir, ligeramente inferior a 3.55, por lo que al redondear con décimas, Python muestra también el valor 3.5.

OBSERVACION:

Para ver el valor que se obtiene al convertir un número decimal a binario se puede utilizar el tipo Decimal de la biblioteca decimal:

In [87]:

```
#EJEMPLO#1

from decimal import Decimal
Decimal(3.45)
```

Out[87]:

```
Decimal('3.450000000000000177635683940025046467781066894531250')
```

In [88]:

```
#EJEMPLO#2

from decimal import Decimal
Decimal(3.55)
```

Out[88]:

```
Decimal('3.549999999999999822364316059974953532218933105468750')
```

El problema del redondeo se agudiza cuando se hacen operaciones, puesto que los errores pueden acumularse (aunque a veces se compensan y pasan desapercibidos).

En algunos casos extremos, el error es apreciable en cálculos muy sencillos:

In [89]:

```
#EJEMPLO

0.1 + 0.1 + 0.1
```

Out[89]:

```
0.30000000000000004
```

En la mayoría de situaciones, estos errores no tienen consecuencias importantes en los resultados finales, pero si en una aplicación concreta se necesita total exactitud, se deben utilizar bibliotecas específicas. Python incluye la biblioteca decimal y existen bibliotecas como mpmath que admiten precisión arbitraria.

Otras funciones integradas (built-in functions)

El intérprete de Python incorpora varias funciones integradas que realizan operaciones matemáticas muy habituales.

Valor absoluto: abs()

La función integrada abs() calcula el valor absoluto de un número, es decir, el valor sin signo.

In [90]:

```
#EJEMPLO#1

abs(-6)
```

Out[90]:

6

In [91]:

```
#EJEMPLO#2

abs(7)
```

Out[91]:

7

Máximo: max()

La función integrada max() calcula el valor máximo de un conjunto de valores (numéricos o alfabéticos). En el caso de cadenas, el valor máximo corresponde al último valor en orden alfabético, sin importar la longitud de la cadena.

In [92]:

```
#EJEMPLO#1

max(4, 5, -2, 8, 3.5, -10)
```

Out[92]:

8

In [93]:

```
#EJEMPLO#2

max("David", "Alicia", "Tomás", "Emilio")
```

Out[93]:

'Tomás'

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

In [94]:

```
#EJEMPLO

max("Ángeles", "Roberto")
```

Out[94]:

'Ángeles'

Mínimo: min()

La función integrada min() calcula el valor mínimo de un conjunto de valores (numéricos o alfabéticos). En el caso de cadenas, el valor mínimo corresponde al primer valor en orden alfabético, sin importar la longitud de la cadena.

In [95]:

```
#EJEMPLO#1

min(4, 5, -2, 8, 3.5, -10)
```

Out[95]:

```
-10
```

In [96]:

```
#EJEMPLO#2
min("David", "Alicia", "Tomás", "Emilio")
```

Out[96]:

```
'Alicia'
```

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

In [97]:

```
min("Ángeles", "Roberto")
```

Out[97]:

```
'Roberto'
```

Suma: sum()

La función integrada sum() calcula la suma de un conjunto de valores. El conjunto de valores debe ser un tipo de datos iterable (tupla, rango, lista, conjunto o diccionario).

In [98]:

```
#EJEMPLO#1

sum((1, 2, 3, 4, 5))
```

Out[98]:

```
15
```

In [99]:

```
#EJEMPLO#2

sum([1, 2, 3, 4, 5])
```

Out[99]:

```
15
```

In [101]:

```
#EJEMPLO#3

sum(range(6))
```

Out[101]:

```
15
```

In [102]:

```
#EJEMPLO#4

sum({1, 2, 3, 4, 5})
```

Out[102]:

```
15
```

Ordenación: sorted()

La función integrada sorted() ordena un conjunto de valores. El conjunto de valores debe ser un tipo de datos iterable (tupla, rango, lista, conjunto o diccionario). El conjunto de valores no se modifica, la función devuelve una lista con los elementos ordenados.

In [103]:

```
#EJEMPLO#1

sorted((10, 2, 8, -3, 6))
```

Out[103]:

```
[-3, 2, 6, 8, 10]
```

In [104]:

```
#EJEMPLO#2

sorted([10, 2, 8, -3, 6])
```

Out[104]:

```
[-3, 2, 6, 8, 10]
```

In [105]:

```
#EJEMPLO#3

sorted({10, 2, 8, -3, 6})
```

Out[105]:

```
[-3, 2, 6, 8, 10]
```

In [106]:

```
#EJEMPLO#4

sorted(("David", "Alicia", "Tomás", "Emilio"))
```

Out[106]:

```
['Alicia', 'David', 'Emilio', 'Tomás']
```

In [107]:

```
#EJEMPLO#5

sorted(["David", "Ángeles", "Tomás", "Óscar", "Emilio"])
```

Out[107]:

```
['David', 'Emilio', 'Tomás', 'Ángeles', 'Óscar']
```

Para saber más:

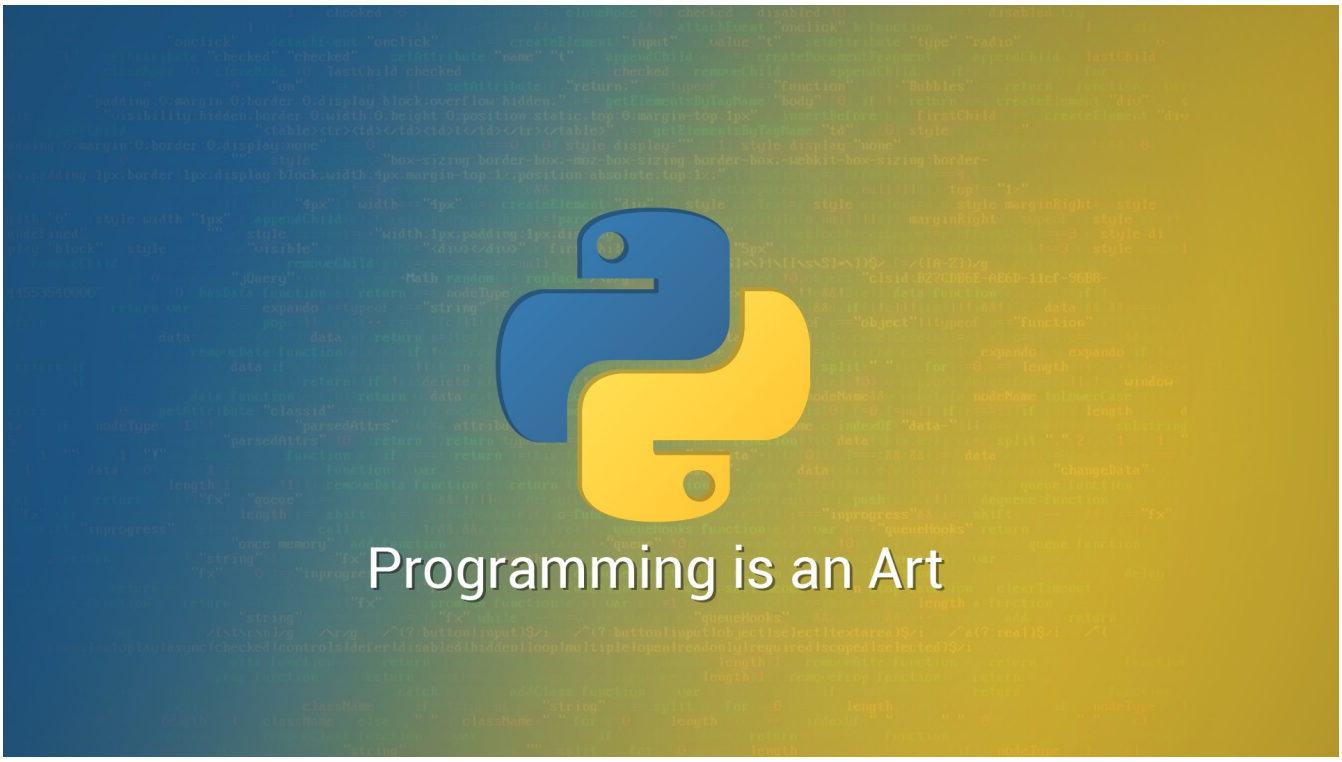
IEEE-754 precisión simple, precisión doble (Wikipedia en inglés)

Manual de Python 3: limitaciones de los decimales en Python

Artículo The perils of Floating Point, de Bruce M. Bush

Artículo What Every Computer Scientist Should Know About Floating-Point Arithmetic, de David Goldberg

Web What Every Computer Scientist Should Know About Floating-Point Arithmetic


Programming is an Art