Operadores aritméticos

Python como calculadora

Una vez abierta la sesión en `python`, podemos aprovechar al máximo este lenguaje: contamos con una calculadora a la mano, sólo hay que ir escribiendo las operaciones en la línea de comandos.

Podemos hacer una suma:

```
3+200
```

203

Una división entre enteros

30/1234

```
0.024311183144246355
```

Una división entre reales

```
3.0/4.0
```

```
0.75
```

Una división entera

Devuelve el cociente (sin decimales)

```
30//4
```

7

Otro ejemplo

```
4//3
```

1

La división devuelve un determinado número de dígitos luego del punto decimal

```
4/3
```

```
1.3333333333333333
```

Combinación de operadores

```
5.0 / 10 * 2 + 5
```

```
6.0
```

¿por qué obtenemos este resultado??

El resultado cambia cuando agrupamos con paréntesis

```
5.0 / (10 * 2 + 5)
```

```
0.2
```

Como podemos ver, el uso de paréntesis en las expresiones tiene una particular importancia sobre la manera en que se evalúan las expresiones

Potenciación de un número

Podemos elevar un número a una potencia en particular

```
2**3**2
```

512

Orden para las potencias

Vemos que elevar a una potencia, la manera en que se ejecuta la expresión se realiza en un sentido en particular: de derecha a izquierda

```
(2**3)**2
```

Los paréntesis

El uso de paréntesis nos indica que la expresión contenida dentro de ellos, es la que se evalúa primero, posteriormente se sigue la regla de precedencia de operadores

Operador módulo

El operador módulo (%) nos devuelve el residuo del cociente

```
17%3
```

2

Tabla de operadores

Precedencia en los operadores aritméticos

Operador	Operación	Ejemplo	Resultado
$**$	Potencia	$2 * *3$	8
$*$	Multiplicación	$7 * 3$	21
$/$	División	$10.5/2$	5.25
$//$	Div. entera	$10.5//2$	5.0
$+$	Suma	$3 + 4$	7
$-$	Resta	$6 - 8$	-2
%	Módulo	15%6	3

Precedencia de los operadores artiméticos 1

1. Las expresiones contenidas dentro de pares de paréntesis son evaluadas primero. En el caso de expresiones con paréntesis anidados, los operadores en el par de paréntesis más interno

Operadores relacionales

Se comparan dos (o más expresiones) mediante un operador, el tipo de dato que devuelve es lógico: **True** o **False**, que también tienen una representación de tipo numérico:

- **True** $= 1$
- **False** $= 0$

Operaciones aritméticas y relacionales

```
1 + 2 > 7- 3
```

```
False
```

Se pueden combinar diferentes expresiones

```
1 < 2 < 3
```

```
True
```

El doble signo igual (==) es el operador de igualdad

```
1 > 2 == 2 < 3

1 > (2 ==  2) <3
```

```
False
```

Combinación de expresiones

```
3 > 4 < 5
```

```
False
```

Hay que tener cuidado con el uso de operadores relaciones estrictos

```
1.0 / 3 < 0.3333
```

```
False
```

Las expresiones se pueden complicar cada vez más, por lo que hay que mantener atención al momento de escribirlas

```
5.0 / 3 >= 11 /7.0
```

```
True
```

El utilizar las operaciones aritméticas y relacionales, extiende por mucho el uso del lenguaje.

```
2**(2. /3) < 3**(3./4)
```

`True`

Tabla de operadores relacionales

Operador	Operación	Ejemplo	Resultado
$==$	Igual a	$4 == 5$	`False`
$!=$	Diferente	$2! = 3$	`True`
$<$	Menor que	$10 < 4$	`False`
$>$	Mayor que	$5 > -4$	`True`
$<=$	Menor o igual	$7 <= 7$	`True`
$>=$	Mayor o igual	$3.5 >= 10$	`False`

Operadores booleanos

Operadores booleanos

En el caso del operador boleano and y el operador or evalúan una expresión compuesta por dos (o más términos).

En ambas expresiones se espera que cada una tenga el valor de True, en caso de que esto ocurra, el valor que devuelve la evaluación, es `True`.

Como se verá en la tabla de verdad, se necesita una condición particular para que el valor que devuelva la comparación, sea `False`.

Operador	Operación	Ejemplo	Resultado
`and`	Conjunción	`False and True`	`False`
`or`	Disyunción	`False or True`	`True`
`not`	Negación	`not True`	`False`

Tabla de verdad de los operadores boleanos

Tipos de variables

Las variables en python sólo son ubicaciones de memoria reservadas para almacenar valores.
Esto significa que cuando se crea una variable, se reserva un poco de espacio disponible en la memoria.

Basándose en el tipo de datos de una variable, el intérprete asigna memoria y decide qué se puede almacenar en la memoria reservada. Por lo tanto, al asignar diferentes tipos de datos a las variables, se pueden almacenar **enteros**, **decimales** o **caracteres (cadenas)** en estas variables.

Asignando valores a variables

Las variables de python no necesitan una declaración explícita para reservar espacio de memoria.
La declaración ocurre automáticamente cuando se asigna un valor a una variable. *El signo igual (=) se utiliza para asignar valores a las variables*.

```
100
1000.0
Chucho
```

Asignación múltiple de valores

En python podemos asignar un valor único a varias variables simultáneamente.

```
A = b = c = 1
print (A)
print (b)
print (c)
```

```
1
1
1
```

En el ejemplo, se crea un objeto entero con el valor 1, y las tres variables se asignan a la misma ubicación de memoria.

También puede asignar varios objetos a varias variables.

```
A, b, c = 1, 2, "Alicia"
print (A)
print (b)
print (c)
```

```
1
2
Ana
```

Aquí, dos objetos enteros con valores 1 y 2 se asignan a las variables A y b respectivamente, y un objeto de cadena con el valor **Alicia** se asigna a la variable c.

Tipos de Datos Estándar

Los datos almacenados en la memoria pueden ser de varios tipos. Por ejemplo, la edad de una persona se almacena como un valor numérico y su dirección se almacena como caracteres alfanuméricos.

En python se cuenta con varios tipos de datos estándar que se utilizan para definir las operaciones posibles entre ellos y el método de almacenamiento para cada uno de ellos.

Los tipos de datos son cinco:

1. Números.
2. Cadena.
3. Lista.
4. Tupla.
5. Diccionario.

Números

Los tipos de datos numéricos almacenan valores numéricos.

```
10
10
```

También se puede eliminar la referencia a un objeto numérico utilizando la sentencia del
La sintaxis de la sentencia del es:
*del var*1[, *var*2[, *var*3[...., *varN*]]]]

Se puede eliminar un solo objeto o varios objetos utilizando la sentencia del
Por ejemplo:

```
del var


del variable1, variable2
```

En python se soportan tres tipos numéricos diferentes:

1. Int (enteros con signo)
2. Flotante (valores reales de punto flotante)

```
Hola Mundo!
H
la
la Mundo!
Hola Mundo!Hola Mundo!
Hola Mundo!PUMAS
```

Listas

Las listas es el tipo de dato más versátil de los tipos de datos compuestos de python.
Una lista contiene elementos separados por comas y entre corchetes ([ ]).

En cierta medida, las listas son similares a los arreglos (arrays) en el lenguaje C.
Una de las diferencias entre ellos es que todos los elementos pertenecientes a una lista pueden ser de tipo de datos diferente.

Los valores almacenados en una lista se pueden acceder utilizando el

```
['abcd', 786, 2.23, 'salmon', 70.2]
abcd
[786, 2.23]
[2.23, 'salmon', 70.2]
[123, 'pizza', 123, 'pizza']
['abcd', 786, 2.23, 'salmon', 70.2, 123, 'pizza']
```

Tuplas

Una tupla es otro tipo de datos de secuencia que es similar a la lista.

Una tupla consiste en un número de valores separados por comas. Sin embargo, a diferencia de las listas, las tuplas se incluyen entre paréntesis.

La principal diferencia entre las listas y las tuplas son:

1. Las listas están entre corchetes [ ] y sus elementos y tamaño pueden cambiarse.
2. Las tuplas están entre paréntesis ( ) y *no se pueden actualizar*.

```
('abcd', 786, 2.23, 'arena', 70.2)
abcd
(786, 2.23)
(2.23, 'arena', 70.2)
(123, 'playa', 123, 'playa')
('abcd', 786, 2.23, 'arena', 70.2, 123, 'playa')
```

El siguiente código es inválido con la tupla, porque intentamos actualizar una tupla, que la acción no está permitida. El caso es similar con las listas.

```
mitupla = ( 'abcd', 786 , 2.23, 'edificio', 70.2  )
milista = [ 'abcd', 786 , 2.23, 'energia', 70.2  ]
mitupla[2] = 1000     # Sintaxis invalida para la tupla
milista[2] = 1000      # Sintaxis invalida para la lista
```

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

TypeError                                 Traceback (most r

<ipython-input-19-a99f473d7b8f> in <module>()
      1 mitupla = ( 'abcd', 786 , 2.23, 'edificio', 70.2  )
      2 milista = [ 'abcd', 786 , 2.23, 'energia', 70.2  ]
----> 3 mitupla[2] = 1000    # Sintaxis invalida para la tu
      4 milista[2] = 1000     # Sintaxis invalida para la l


TypeError: 'tuple' object does not support item assignment
```

Pero en la lista podemos agregar nuevos elementos que se colocan al final de la misma:

```
print(milista)
milista.append('hola')
print(milista)
```

```
['abcd', 786, 2.23, 'energia', 70.2]
['abcd', 786, 2.23, 'energia', 70.2, 'hola']
```

Diccionarios

Los diccionarios de python son de tipo tabla-hash.
Funcionan como arrays asociativos y consisten en pares *clave-valor*.

Una clave de diccionario puede ser casi cualquier tipo de python, pero suelen ser números o cadenas.
Los valores, por otra parte, pueden ser cualquier objeto arbitrario de python.

Los diccionarios están encerrados por llaves { } y los valores se pueden asignar y acceder mediante llaves cuadradas [ ].

```
fisicos = dict()

fisicos ={
```

```
{1: 'Eistein', 2: 'Bohr', 3: 'Pauli', 4: 'Schrodinger', 5:
dict_keys([1, 2, 3, 4, 5])
dict_values(['Eistein', 'Bohr', 'Pauli', 'Schrodinger', 'Ha
{1: 'Eistein', 2: 'Bohr', 3: 'Pauli', 4: 'Schrodinger', 5:
```

Regla para los identificadores

Los identificadores son nombres que hacen referencia a los objetos que componen un programa: **constantes**, **variables**, **funciones**, etc.

Reglas para construir identificadores:

- El primer carácter debe ser una letra o el carácter de subrayado (guión bajo)
- El primer carácter puede ir seguido de un número variable de dígitos numéricos, letras o carácteres de subrayado.
- No pueden utilizarse espacios en blanco, ni símbolos de puntuación.
- En python se distingue de las mayúsculas y minúsculas.