Python como calculadora

Introducción a Python

Python es el lenguaje de programación que aprenderemos este semestre. Python es un lenguaje **interpretado** de propósito general que sustenta diferentes paradigmas de programación, pero nos enfocaremos solamente en el paradigma procedural-imperativo, es decir, en programas que describen **procedimientos detallados** para un computador de **cómo hacer** una tarea.

Como Python es un lenguaje interpretado, un usuario puede **interactuar** directamente con su intérprete y utilizarlo, por ejemplo, como una calculadora. Intentemos lo siguiente:

Ejemplo 1

- Escribamos 2+5 ↲
- Python responde 7, el resultado de la operación matemática 2+5

```
>>> 2+5
7
>>>
```

- Ahora hagamos algo más complejo: 7 + 10 + 100 − 4 * 2 ↲
- Python responde 109 que es el resultado de la operación matemática ingresada:

```
>>> 7 + 10 + 100 - 4 * 2
109
>>>
```

Pregunta 1

Ahora, apoyándonos en los años de experiencia que tenemos con operaciones aritméticas, respondamos la primera pregunta de la actividad.

Como se podemos darnos cuenta, las expresiones aritméticas se pueden expresar igual a como lo conocemos y hacemos habitualmente:

(Número) (Operador) (Número)

Esto se conoce como **notación infija,** y es la que usamos cada vez que escribimos una operación matemática. Notarán que una diferencia puede estar en la multiplicación, en donde su operador corresponde al símbolo * (asterisco), además el operador división corresponde al **/ (slash)** en vez de **÷ (símbolo de división matemático)** o **: (dos puntos)**.

Python provee, además, un par de operaciones que pueden ser de utilidad: **Resto o módulo (%)** y la **Potencia (**),** y también dos operaciones "unarias": **Identidad (+)** y **cambio de signo (-).**

Ejemplo 2

Probemos ingresando las siguientes expresiones a Python:

```
>>> 2 ** 3
8
>>> 2 ** 0
1
>>> 27 % 5
2
>>> 25 % 5
0
>>> 5 / 5
1
>>> 5 / 2
2
>>> 2 / 10
0
>>> 2 ** 31
2147483648L
>>>
```

Del ejemplo anterior podemos notar que ocurre algo extraño al utilizar las divisiones, la primera división entrega el resultado esperado, pero las otras dos no. Esto se debe a que el operador de división que estamos usando es el **operador de división entera.**

Esto se debe a que todos los valores con los que se está operando pertenecen al conjunto de los **números enteros (Z).** Por lo tanto, Python considera que el resultado de la operación debe entregarse necesariamente en dicho conjunto a menos que le indiquemos **explícitamente** lo contrario. Entonces **¿cómo utilizamos números distintos a los enteros?**

Enteros, no enteros y otros números

Probemos que sucede si escribimos los ejemplos anteriores (y alguno nuevos) ahora de la siguiente forma:

Ejemplo 3

```
>>> 5.0 / 5.0
1.0
>>> 5.0 / 2
2.5
>>> 2.0 / 10
0.20000000000000001
>>> 2.5 * 4
10.0
>> 0.20000000000000001 * 5
1.0
>>> 1.0/10000000
9.9999999999999995e-08
>>>
```

Como podemos ver, ahora sí Python está trabajando con **números no enteros**, pues al tener por lo menos uno de los números de la operación escrito con un punto decimal (Usamos un punto **(.)** en vez de una coma **(,)** debido a que Python fue escrito por angloparlantes) Python "comprende" que la operación ya no está siendo calculada dentro del conjunto de los enteros, sino en el conjunto de los números no enteros o como se le denomina en computación: **números de punto flotante**

Pregunta 2

Ahora es posible resolver la pregunta 2 de la actividad.

Hemos sido cuidadosos en hablar de "números no enteros" y no de "números reales". Esto se debe a que necesitaríamos **infinitos bits** para representar el conjunto de número reales, lo que no es posible. En su remplazo, los computadores usan **números de punto flotante** que los aproximan. Si lo pensamos detenidamente, entre dos números reales, independiente cuáles, siempre existirá al menos uno, y en la realidad, entre dos números reales cualquiera existen **infinitos números**, es decir, los números reales corresponden a

un conjunto denso, sin embargo, en los números de punto flotante, es posible llegar a un punto donde ésta propiedad **no se cumple**, es decir, existirán números entre los cuáles **no existen otros elementos**, pues computacionalmente, la **precisión** con la que se **está representando al número no entero** no lo permite.

La codificación de los números de punto flotante, o simplemente flotantes, se parece mucho a la **notación científica**.

Ejemplo 4

El número 112.625 puede escribirse como

$$1.12625 \times 10^2$$

No confunda la notación exponencial con la operación exponenciación

1.12625×10^2 lo que equivale a 112.625 en **`notación de punto flotante`**

112.625^2 lo que equivale a $12.684.390.625$ de la **`operación exponenciación`**

Pero el número 1.12625×10^2 escrito en notación de punto flotante está en base decimal:

$$1.12625 \times 10^2 = 1 \times 100 + 1 \times 10 + 2 \times 1 + 6 \times \frac{1}{10} + 2 \times \frac{1}{100} + 5 \times \frac{1}{1000}$$

o

$$1.12625 \times 10^2 = 1 \times 10^2 + 1 \times 10^0 + 2 \times 10^1 + 6 \times 10^{-1} + 2 \times 10^{-2} + 5 \times 10^{-3}$$

Pregunta 3

Intentemos ahora resolver la pregunta 3 de la actividad.

En este punto hemos visto que el intérprete de Python ha arrojado valores que siguen 3 patrones distintos:

- **Números enteros:** Aquellos números que están representados sin un punto y ningún otro caracter más que el valor que debieran tener (ejemplo: 100), éstos

números son conocidos en Python como enteros (integer) y en computación se les denomina comúnmente **int**.

- **Números no enteros:** Aquellos números con un punto decimal, representados como decimales simples o en notación de flotante (ejemplo: 9.5367431e-07), a éstos números (que no son números reales), se les conoce como números de punto flotante y comúnmente se les denomina **float**.

- **Números con una "L" misteriosa al final:** Existen números cuyo resultado tenía una extraña letra L al final, como por ejemplo 2147483648L.

  Esto se debe a que los integer en Python también son un **conjunto finito**, pues se representan con una cantidad fija de **4 bytes** que permiten representar los enteros entre -2147483648 y 2147483647 (2^{32} combinaciones de 0 y 1 distintas), de éste modo, los números que caen fuera de este rango, son números que utilizarán **más memoria** en el computador y se les denomina enteros largos o **long** y Python los muestra al usuario con una "L" o "l" al final.

Precedencia de operadores

En los ejercicios anteriores quizás notaron algunos problemas con los resultados esperados y los obtenidos. Esto ocurre por una sencilla razón: la **precedencia de operadores**.

Ejemplo 5

Si tenemos que resolver **5 + 5 ** 3 / 5 * 4 – 10 * 3**, ¿con qué operación comenzamos?

Si las operaciones se realizaran por el orden de aparición de los operadores, de izquierda a derecha, el resultado sería: 5 + 5 = 10, 10 ** 3 = 1000, 1000 / 5 = 200, 200 * 4 = 800, 800 - 10 = 790, y 790 * 3 = **2370**. Pero Python nos entrega un resultado distinto:

```
>>> 5 + 5 ** 3 / 5 * 4 - 10 * 3
75
>>>
```

La **precedencia** de operadores son reglas sencillas que permiten a Python "ordenar" la evaluación de expresiones. Este orden **se puede modificar con el uso de paréntesis**, tal y como se realiza en matemáticas.

Pregunta 4

Ahora, resolvamos la cuarta pregunta del día de hoy.

Podemos ver que Python usa los paréntesis como en la aritmética tradicional. Pero el uso de paréntesis conlleva una preocupación extra: que éstos estén **balanceados**.

Ejemplo 6

Miremos lo que hace el intérprete de Python cuando nos equivocamos en el balanceo de paréntesis:

```
>>> ((((((2+((3**2)/5*3)+3*2-(-3*4))+5)+5)-3)-2)*2)-1)
SyntaxError: invalid syntax
>>>
```

El intérprete no “interpreta” nuestra expresión, sólo nos señala que existe un error y que éste corresponde a un error sintáctico.

Ahora que hemos usado Python como una calculadora básica, supongamos que necesitamos calcular el perímetro y área de un círculo. Como sabemos, las fórmulas respectivas son:

$$Perímetro = 2\pi r$$

$$Área = 2\pi r^2$$

Si trabajamos con una buena precisión decimal, nos tardaremos bastante en escribir cada cálculo, y es muy fácil que nos equivoquemos en algún decimal.

Ejemplo 7

Supongamos que calculamos el perímetro y el área de un círculo con radio de 2.452335634384712 cm, y que usamos el valor π con 18 decimales.:

```
>>> 2 * 3.141592653589793238 * 2.452335634384712
15.408479226238953
>>> 3.141592653589793238 * 2.452335634384712 ** 2
18.893381339091178
>>>
```

¿Y si ahora queremos calcular el perímetro de una círculo con el doble del radio?

```
>>> 2 * 3.141592653589793238 * 2 * 2.452335634384712
30.816958452477905
>>> 3.141592653589793238 * (2 * 2.452335634384712) ** 2
75.573525356364712
>>>
```

¿Qué sucede si lo que necesitamos es calcular repetidamente valores, o si requerimos de valores previos obtenidos para continuar con nuestros cálculos? Pues la digitación se torna tediosa.

Para mejorar esta situación, podemos pedirle a Python que **"recuerde"** valores vistos anteriormente.

Ejemplo 8

```
>>> PI = 3.141592653589793238
>>> radio = 2.452335634384712
>>> 2 * PI * radio
15.408479226238953
>>> PI * radio ** 2
18.893381339091178
>>> 2 * PI * (2 * radio)
30.816958452477905
>>> PI * (2 * radio) ** 2
75.573525356364712
>>>
```

Con esto, indicamos a Python que **PI** contiene el valor 3,1415… y que **radio** contiene 2.452… Noten que el símbolo **igual (=)** no es una comparación ni una equivalencia matemática, sino que una **asignación**, es decir una asociación de un valor (en la memoria

del computador) a un nombre. Cuando Python encuentra un nombre en alguna expresión, éste se evalúa como el valor que le fue asignado previamente.

Habremos notado que escribimos el nombre **PI** en mayúsculas, mientras que el nombre `radio` en minúsculas. Este no ha sido por capricho, sino que por una **convención** entre programadores, quienes decidieron usar mayúsculas para identificar **constantes**, nombres cuyo valor se asigna una sola vez en el programa (no se cambian), y minúsculas para identificar **variables**, nombres cuyo valor puede ir cambiando a medida que avanza un programa.

También debemos notar que el intérprete de Python hizo diferencia con la sentencias de asignación del ejemplo. Cuando escribimos expresiones aritméticas, el intérprete nos mostró los resultados inmediatamente. Sin embargo, al asignar, no se muestra inmediatamente el contenido de la variable o constante. Podemos conocer el valor de una constante o variable de dos formas.

Ejemplo 8

```
>>> radio
2.452335634384712
>>> PI
3.141592653589793238
>>> print "El radio del círculo es", radio
El radio del círculo es 2.45233563438
>>>
```

Una opción entonces es escribir una expresión que sólo contiene el nombre de la variable o constante. La otra, es usar la sentencia **print**, que toma una lista de mensajes y expresiones (separador con comas) y los despliega por pantalla en el mismo orden en que se especifican.

Reglas para identificadores

Las variables y constantes pueden tener cualquier nombre, siempre y cuando se cumpla con algunas sencillas reglas:

1. Un identificador debe estar formado por:
    a. **Letras:** minúsculas, mayúsculas.
    b. **Dígitos**
    c. **Carácter de subrayado (_)**
2. El **primer** caracter no puede ser un dígito.
3. No puede coincidir con una palabra reservada del lenguaje Python las cuáles son: **and, assert, break, class, continue, def, del, elif, else, except, exec, finally, for, from, global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, while, yield.**
4. Debe ser representativo, ya que hace más entendible el programa. Por ejemplo: “radio”, “pi”, “sumando1”, “nombre”, “edad”.

Pregunta 5

Para terminar desarrollemos la pregunta 5 de la actividad de hoy