**INFORMÁTICA, GRUPO D**

# Un calculadora avanzada ... y algo más!

Vamos a familiarizarnos con el entorno interactivo de Python. Vamos a construir *expresiones aritméticas* y a guardar los resultados en *variables* mediante *asignaciones*.

## Operadores aritméticos

In [1]:
```
1 + 1
```

Out[1]: 2

In [2]:
```
5 - 3
```

Out[2]: 2

In [3]:
```
5*3
```

Out[3]: 15

In [4]:
```
6 / 2
```

Out[4]: 3

In [5]:
```
5 / 2 #cuidado: se opera con enteros
```

Out[5]: 2

In [6]:
```
#resto (módulo), cambio de signo, potencia
print 5%2 , -5, 5**2
```

```
1 -5 25
```

## Precedencia de los operadores. Paréntesis.

Las operaciones aritméticas básicas son **binarias**. Es decir, por ejemplo, sólo puedo sumar dos números a la vez. Si quiero calcular 1 + 2 + 3, Python ejecuta (1 + 2) + 3. La suma y la resta son operaciones **asociativas por la izquierda**.

In [7]:
```
1 + 2 + 3 == (1+ 2) + 3
```

Out[7]: True

Las multiplicación y la división también son **asociativas por la izquierda**.

In [8]:
```
3 * 4 / 2 == (3 * 4)/2
```

Out[8]: True

In [9]:
```
#Pero
2 + 4 * 5 == (2 + 4) * 5
```

Out[9]: False

No se respeta la asociatividad a la izquierda porque la multiplicación y la división tienen *mayor nivel de precedencia* que la suma y la resta.

Podemos alterar el nivel de precedencia utilizando **paréntesis**.

In [10]:
```
2 + (4 * 5) == (2 + 4) * 5
```

Out[10]: False

El operador % es asociativo por la izquierda y tiene el mismo nivel de precedencia que la multiplicación.

La potencia es asociativa *por la derecha* y tiene mayor nivel de precedencia.

In [11]:
```
2**3**2 == 2**(3**2)
```

Out[11]: True

**NIVEL DE PRECEDENCIA** 1. Exponenciación ** 2. Cambio de signo - 3. Multiplicación, división, resto. 4. Suma, resta.

# Tipos de datos

In [12]:
```
print 3/4 , 3.0/4
```

```
0 0.75
```

### Enteros y flotantes

- Cada valor calculado por Python tiene un *tipo* determinado.
- En cualquier operación, siempre se intenta respetar el tipo de los operandos.
- Para obtener resultado de *tipo real* necesitamos operandos reales o bien operaciones que generen reales.

In [13]:
```
3 == 3.0
```

Out[13]: True

In [14]:
```
3 is 3.0
```

Out[14]:
```
False
```

- Los enteros ocupan menos memoria.
- Las operaciones con enteros son más rápidas.
- Sólo utilizaremos reales cuando de verdad los necesitemos.

Python trabaja con aproximaciones de los reales: *números en formato de coma flotante* o simplemente *flotantes* (float).

Un flotante consta de dos partes, *mantisa* y *exponente*.

(mantisa)E(exponente) = mantisa*10^exponente

In [15]:
```
2E3
```

Out[15]:
```
2000.0
```

In [16]:
```
2e-3
```

Out[16]:
```
0.002
```

In [17]:
```
0.0000003021
```

Out[17]:
```
3.021e-07
```

In [18]:
```
0.9849839499349893489849738493
```

Out[18]:
```
0.9849839499349894
```

In [19]:
```
10009084984938943849384938 43.30292
```

Out[19]:
```
1.0009084984938944e+27
```

In [20]:
```
.9228
```

Out[20]:
```
0.9228
```

In [21]:
```
1.
```

Out[21]:
```
1.0
```

In [22]:
```
#Los tipos se pueden combinar
3 + 4 - 6.25
```

Out[22]:
```
0.75
```

## Valores lógicos

Existe el tipo *booleano* para guardar los valores lógicos cierto (True) y falso (False)

In [23]:
```
True
```

Out[23]: True

In [24]:
```
False
```

Out[24]: False

Existen además una serie de **operadores lógicos y de comparación.**

In [25]:
```
# and : y lógica
True and True
```

Out[25]: True

In [26]:
```
True and False
```

Out[26]: False

In [27]:
```
False and True
```

Out[27]: False

In [28]:
```
False and False
```

Out[28]: False

In [29]:
```
# or : "o lógica" o disyunción
True or False
```

Out[29]: True

In [30]:
```
# not : negación
not True
```

Out[30]: False

El tipo booleano cobra mayor importancia al utilizar los operadores de comparación.

In [31]:
```
#Operador de igualdad
2 == 3
```

Out[31]: False

In [32]:
```
True == False
```

In [32]:
```
True == False
```

Out[32]: False

In [33]:
```
2 == 1 + 1 % 3
```

Out[33]: True

In [34]:
```
# Comparación de cantidades
2 > 1
```

Out[34]: True

In [35]:
```
2 < 1
```

Out[35]: False

In [36]:
```
3 <= 2 + 1
```

Out[36]: True

In [37]:
```
# existe >=
# también podemos probar la desiguadad
9 <> 3**2
```

Out[37]: False

In [38]:
```
9 != 3**2
```

Out[38]: False

## Variables y asignaciones

Para simplificar algunos cálculos, es util guardar algunos valores en *variables*

In [39]:
```
#área de un círculo
# a = pi * radio^2
pi = 3.14
radio = 10
area = pi * radio**2
area
```

Out[39]: 314.0

**Asignación** : dar valor a una *variable* utilizando una *expresión*.

Formulación: *variable* = *expresión*

*Una asignación no es una ecuación matemática*. Primero se evalúa la expresión a la derecha del símbolo igual (=) y se guarda el valor resultante en la variable indicada a la izquierda del símbolo =

In [40]:
```
#la variable se puede usar varias veces para distintos valores.
a = 3
a = 25 + 1
a = 10**2
a = 2.5
a
```

Out[40]: 2.5

In [41]:
```
a = a + 1  #??
a
```

Out[41]: 3.5

El nombre de una variable es su *identificador*. Los identificadores siguen siguen unas reglas precisas.

Un **identificador** debe estar formado por letras minúsculas, mayúsculas, dígitos y/o el carácter de subrayado (_), con una restricción: que el primer carácter no sea un dígito.

Hay una norma más: un identificador no puede coincidir con una palabra reservada o palabra clave. Una palabra reservada es una palabra que tiene un significado predefinido y es necesaria para expresar ciertas construcciones del lenguaje.

**Palabras reservadas de Python:** 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 y yield.

Identificadores válidos: h, x, Z, velocidad, x, fuerza1, masa_2, prueba_123, desviacion_tipica.

Python distingue entre mayúsculas y minúsculas.

Cuidado con los espacios en blanco.

Dado que eres libre de llamar a una variable con el identificador que quieras, hazlo *con clase*: escoge siempre nombres que guarden relación con los datos del problema.

**Asignaciones con operador**

Podemos simplificar las operaciones cuando hay una única variable

In [42]:
```
a = 10
a = a + 1
a += 1
a
```

Out[42]: 12

Se utiliza para todos los operadores

In [43]:
```
b = 10
b *= 5 # es lo mismo que b = b * 5
b
```

Out[43]: 50

In [44]:
```
c = 10
c **= 6
c
```

Out[44]:
```
1000000
```

**El tipo de datos cadena**

En muchos lenguajes se le llama *string*

Permite guardar cadenas de símbolos

In [45]:
```
a = 'hola'
a
```

Out[45]:
```
'hola'
```

Las operaciones básicas con la concatenación (+) y la repetición (*)

In [46]:
```
a = a + a
```

In [47]:
```
a
```

Out[47]:
```
'holahola'
```

In [48]:
```
b = '-'*10
b
```

Out[48]:
```
'----------'
```

In [49]:
```
#cuidado con los espacios
b + ' '*6 + b
```

Out[49]:
```
'----------      ----------'
```

CUIDADO: Una cadena no es un identificador

In [50]:
```
hola = 3.14
```

In [51]:
```
hola == 'hola'
```

Out[51]:
```
False
```

In [52]:
```
hola + hola
```

Out[52]:
```
6.28
```

In [53]:
```
'hola' + 'hola'
```

Out[53]:
```
'holahola'
```

In [54]:
```
hola + 'hola'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-54-db7e7bbc7041> in <module>()
----> 1 hola + 'hola'

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

In [55]:
```
'12'+'12'
```

Out[55]:
```
'1212'
```

A los caracteres individuales a veces se les llama de tipo *char*. Tienen operadores propios.

In [56]:
```
a = 'a'
ord(a)
```

Out[56]:
```
97
```

In [57]:
```
chr(98)
```

Out[57]:
```
'b'
```

In [58]:
```
#Se respeta el orden alfabético
'adios'>'burro'
```

Out[58]:
```
False
```

## Funciones predefinidas

Python tiene una serie de funciones predefinidas.

*abs*: valor absoluto

In [59]:
```
abs(-4)
```

Out[59]:
```
4
```

*float*: conversión a flotante. Acepta enteros y cadenas.

In [60]:
```
float(3)
```

Out[60]:
```
3.0
```

In [61]:
```
float('3.34')
```

Out[61]:
```
3.34
```

In [62]:
```
float('3.4e11')
```

Out[62]: 340000000000.0

In [63]:
```
float('3k10')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-63-9f7d72827442> in <module>()
----> 1 float('3k10')

ValueError: invalid literal for float(): 3k10
```

*int*: conversión a entero. Acepta flotantes y cadenas.

In [64]:
```
int('29')
```

Out[64]: 29

In [65]:
```
int(3.1)
```

Out[65]: 3

In [66]:
```
int(3.9)
```

Out[66]: 3

*str*: conversión a cadena. Recibe un número y devuelve una representación como cadena.

In [67]:
```
str(10)
```

Out[67]: '10'

In [68]:
```
str(3.1e4)
```

Out[68]: '31000.0'

*round*: redondeo. Puede usarse con uno o dos argumentos. Si se usa con un argumento, redondea el número al flotante más próximo o cuya parte decimal sea nula. (¡Observa que el resultado siempre es de tipo flotante!) Si round recibe dos argumentos, estos deben ir separados por una coma y el segundo indica el número de decimales que queremos conservar tras el redondeo.

In [69]:
```
round(10.3)
```

Out[69]: 10.0

In [70]:
```
round(10.8)
```

Out[70]: 11.0

In [71]:
```
a = 45.99893843959393
```

```
a **= 2
```

In [72]:
```
a
```

Out[72]:
```
2115.902337569552
```

In [73]:
```
round(a,4)
```

Out[73]:
```
2115.9023
```

# El módulo math

Podemos utilizar numerosas funciones matemáticas utilizando, importándolas del módulo math.

In [74]:
```
from math import sin, cos
from math import pi
```

In [75]:
```
sin(pi)
```

Out[75]:
```
1.2246467991473532e-16
```

In [76]:
```
cos(pi/2) #esto es cero, no?
```

Out[76]:
```
6.123233995736766e-17
```

Más sencillo: podemos importar todo

In [77]:
```
from math import *
```

- sin(x), Seno de x, expresado en radianes.
- cos(x), Coseno de x, expresado en radianes.
- tan(x), Tangente de x, expresado en radianes.
- exp(x), el número e elevado a x.
- ceil(x), Redondeo hacia arriba de x.
- floor(x), Redondeo hacia abajo de x.
- log(x), Logaritmo en base e de x.
- log10(x), Logaritmo en base 10 de x.
- sqrt(x), Raíz cuadrada de x.

También se definen las constantes pi y e.

In [78]:
```
pi
```

Out[78]:
```
3.141592653589793
```

In [79]:
```
e
```

Out[79]:
```
2.718281828459045
```

In [80]:
```
floor(pi)
```

Out[80]:
```
3.0
```

In [81]:
```
ceil(pi)
```

Out[81]:
```
4.0
```

# Cómo crear mis propias funciones

In [82]:
```
def cuadrado(x):
    return x**2
```

In [83]:
```
cuadrado(5)
```

Out[83]:
```
25
```

In [84]:
```
def esCuadradoPerfecto(n):
    m = int(sqrt(n))
    return m*m == n
```

In [85]:
```
esCuadradoPerfecto(30)
```

Out[85]:
```
False
```

In [86]:
```
esCuadradoPerfecto(25)
```

Out[86]:
```
True
```

In [87]:
```
def cuadradoPrevio(n):
    m = int(sqrt(n))
    return m**2
```

In [88]:
```
cuadradoPrevio(30)
```

Out[88]:
```
25
```

In [89]:
```
cuadradoPrevio(25)
```

Out[89]:
```
25
```

In [89]:

In [89]:

In [89]: