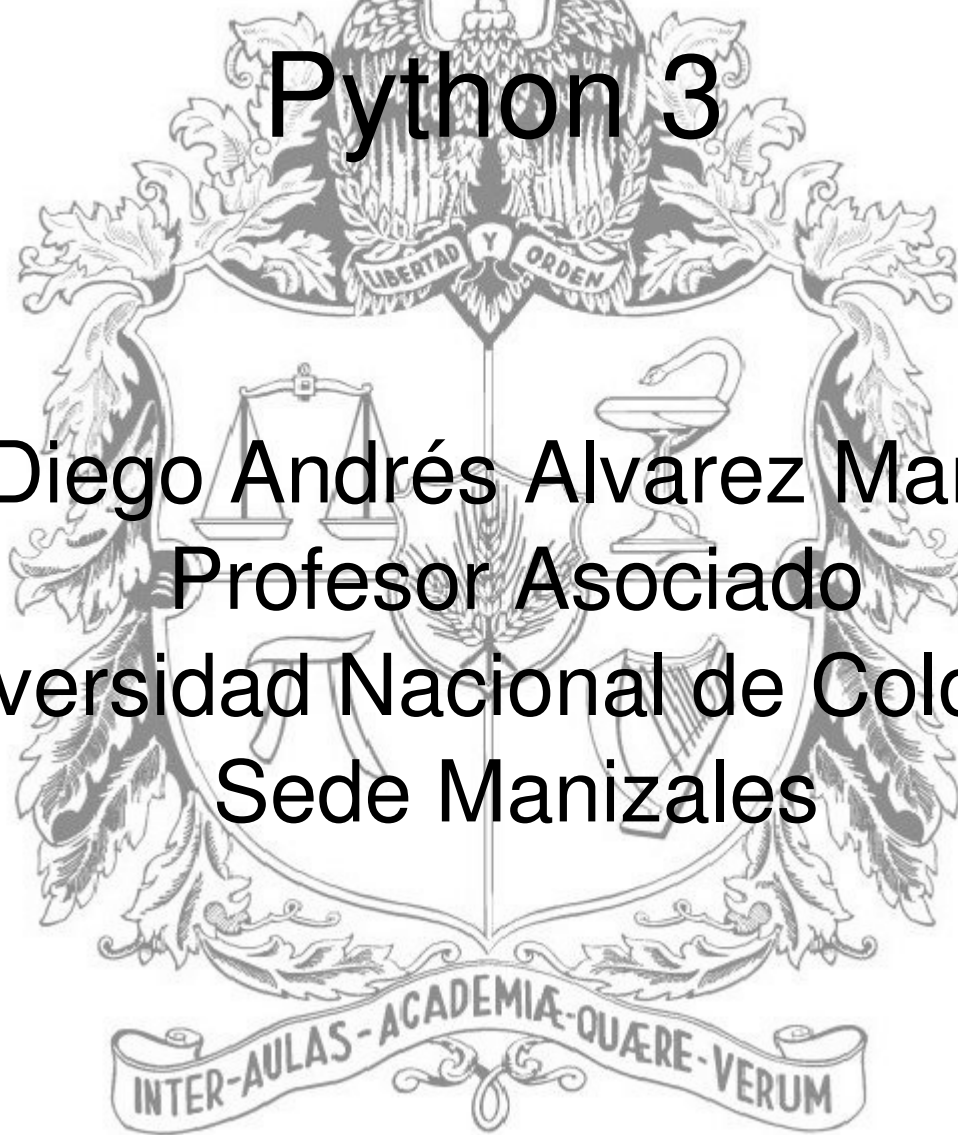


# 02 - Tipos de datos escalares en Python 3

Diego Andrés Álvarez Marín  
Profesor Asociado  
Universidad Nacional de Colombia  
Sede Manizales



# Comentarios en Python

Los comentarios en Python empiezan con el carácter #

```
>>> # este es un comentario
>>> radio = 2 # cm (este es otro comentario)
>>> cadena = "# este no es un comentario" # pero este si lo es
>>> cadena
'# este no es un comentario'
>>> |
```

# Variables

Una **variable** es un nombre o referencia a un valor guardado de la memoria del sistema

En Python no es necesario declarar las variables al principio del programa (esto es necesario en lenguajes como C, C++, Pascal, Visual Basic). Las variables se declaran automáticamente cuando se les asigna un valor por primera vez. Su tipo corresponderá al tipo de dato que contienen.

# Reglas para la creación de identificadores o nombres de variables, funciones, etc.

- Los nombres dados a las referencias de objetos se llaman identificadores o simplemente nombres
- Los nombres válidos en Python pueden tener cualquier longitud.
- El primer carácter, debe ser una letra (UNICODE), o el guión bajo `_`
- Los caracteres siguientes pueden ser también números (UNICODE), por ejemplo: '0' ... '9'. Es decir `area2`, `area_2`, `área2`, `_area2` son un identificadores válidos; `2area`, `área media` (con el espacio intermedio), `area.media`, `a(b)` no lo son.
- Python diferencia entre mayúsculas y minúsculas (es *case-sensitive* en inglés): `Arbol`, `ARBOL`, `arbol`, `ArBoL` y `árbol` son todos nombres diferentes
- Se recomienda no utilizar los nombres ya utilizados por Python, ejemplo: `int`, `float`, `list`, `tuple`, `len`, `str`, etc.
- Un nombre válido no puede ser una de las palabras claves (keywords) de Python.
- Nunca utilice la `ele` o la `“ó”` como nombre de variable, ya que se puede confundir con un uno o con un cero, respectivamente: `“1230 vs l23O”`, `“1230 vs 123O”`, `“1230 vs l23O”`, `“1230 vs 123O”`

# Palabras reservadas de Python 3

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

Y con Python 3.7:

async  
await

```
>>> import keyword
>>> keyword.kwlist
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', '
continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for',
'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal',
'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yie
ld']
>>> keyword.iskeyword('While')
False
>>> keyword.iskeyword('while')
True
>>>
```

**Nota:** las palabras clave no se pueden utilizar como identificadores. Se deben escribir exactamente como se listan aquí.

# Tipos de datos

Los tipos de datos determinan el conjunto de valores que un objeto puede tomar y las operaciones que se pueden realizar con ellas.

- Tipos de datos escalares:
  - Números enteros, flotantes, complejos, fraccionarios, lógicos(booleanos)
- Tipos de datos secuenciales:
  - Secuencias de bytes, cadenas
- Tipos de datos estructurados:
  - Listas (lists): secuencias ordenadas de valores
  - Tuplas (tuples): secuencias inmutables de valores ordenados
  - Conjuntos (sets): conjunto no ordenado de valores
  - Diccionarios (dictionaries): conjunto no ordenado de valores, que tienen una “llave” que los identifican
- Objetos: módulos, funciones, clases, métodos, archivos, código compilado, etc.
- “Constantes”

# Números

- Enteros (int): pueden ser arbitrariamente largos, es decir no hay límites MIN\_INT o MAX\_INT como en lenguaje C. Su tamaño está limitado por la memoria del computador.

- Base 10: 1, 2

```
>>> 0xFF - 250 + 0b10 - 0o10  
-1
```

- Base 2: 0b101110110, 0B001001001110

- Base 8: 0o232573, 0017321577

- Base 16: 0x23AF57BA, 0XFF23AB3C

- Flotantes (double en lenguaje C): 1.2, 7.43e4, 1.2E-3, 1.0
- Números complejos: 7+3j, 32+4J
- Fracciones (fraction): 1/21, 2/423
- Decimales (decimal): Decimal('-0.2')

# Números

- A partir de Python 3.6 se permite escribir los números con guión bajo para mejorar su legibilidad:

```
>>> 1_000_000_000_000_000
1000000000000000
>>> 0xFF_FF_FF_FF
4294967295
```

```
>>> programmer_error = 0xbad_c0ffee
>>> flags = 0b_0111_0101_0001_0101
```



# Enteros vs. Flotantes

```
>>> entero = 2          # este es un int
>>> flotante = 2.0      # este es un float
>>> entero
2
>>> flotante
2.0
>>> type(entero)
<class 'int'>
>>> type(flotante)
<class 'float'>
>>> |

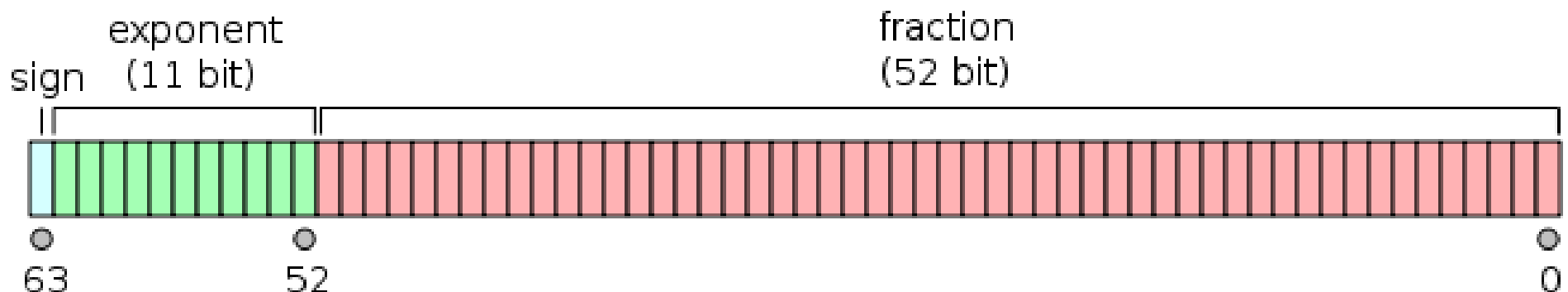
>>> type(1e5)
<class 'float'>
>>> type(100000)
<class 'int'>
```

# Números flotantes

- Se codifican en memoria utilizando el estándar IEEE Standard 754 for floating point arithmetic:

[http://en.wikipedia.org/wiki/IEEE\\_floating\\_point](http://en.wikipedia.org/wiki/IEEE_floating_point)

- Poseen una precisión de 64 bits.
- Tienen 15 dígitos de precisión
- Es el mismo “double” de lenguaje C, C++



## sys.float\_info

A *struct sequence* holding information about the float type. It contains low level information about the precision and internal representation. The values correspond to the various floating-point constants defined in the standard header file float.h for the 'C' programming language; see section 5.2.4.2.2 of the 1999 ISO/IEC C standard [C99], 'Characteristics of floating types', for details.

attribute	float.h macro	explanation
epsilon	DBL_EPSILON	difference between 1 and the least value greater than 1 that is representable as a float
dig	DBL_DIG	maximum number of decimal digits that can be faithfully represented in a float; see below
mant_dig	DBL_MANT_DIG	float precision: the number of base-radix digits in the significand of a float
max	DBL_MAX	maximum representable finite float
max_exp	DBL_MAX_EXP	maximum integer e such that $\text{radix}^{e-1}$ is a representable finite float
max_10_exp	DBL_MAX_10_EXP	maximum integer e such that $10^e$ is in the range of representable finite floats
min	DBL_MIN	minimum positive normalized float
min_exp	DBL_MIN_EXP	minimum integer e such that $\text{radix}^{e-1}$ is a normalized float
min_10_exp	DBL_MIN_10_EXP	minimum integer e such that $10^e$ is a normalized float
radix	FLT_RADIX	radix of exponent representation
rounds	FLT_ROUNDS	integer constant representing the rounding mode used for arithmetic operations. This reflects the value of the system FLT_ROUNDS macro at interpreter startup time. See section 5.2.4.2.2 of the C99 standard for an explanation of the possible values and their meanings.

```

>>> import sys
>>> sys.float_info
sys.float_info(max=1.7976931348623157e+308, max_exp=1024, max_10
_exp=308, min=2.2250738585072014e-308, min_exp=-1021, min_10_exp
=-307, dig=15, mant_dig=53, epsilon=2.220446049250313e-16, radix
=2, rounds=1)
>>> sys.float_info.max
1.7976931348623157e+308
>>> sys.float_info.min
2.2250738585072014e-308
>>> sys.float_info.dig
15
>>> sys.float_info.epsilon
2.220446049250313e-16
>>> |

```

$\longrightarrow$  `math.log10(2**52)`  
15.653559774527022

Como una curiosidad, el IEEE standard 754, permite los llamados números **subnormales** (o **denormales**), los cuales son números incluso más pequeños que `sys.float_info.min`. Ver: [http://en.wikipedia.org/wiki/Denormal\\_number](http://en.wikipedia.org/wiki/Denormal_number). Todo depende del procesador de su PC. Estos números están en el intervalo entre `sys.float_info.min*sys.float_info.epsilon` y `sys.float_info.min` es decir [4.94e-324, 2.23e-308]. Cualquier número más pequeño que este se redondea a cero. Tenga en cuenta, que esto no tiene importancia práctica alguna en sus cálculos. Además hay una pérdida de precisión asociada a su uso.

```

>>> 2e-324
0.0
>>> 3e-324
5e-324
>>> 7e-324
5e-324
>>> 8e-324
1e-323

```

**sys.float\_info.epsilon** representa el número más pequeño posible que puede sumársele a 1.0 para que  $(1.0 + \text{sys.float\_info.epsilon}) > 1.0$ . En MATLAB esta constante se llama "eps". **sys.float\_info.epsilon** representa la exactitud relativa de la aritmética del computador. Observe que un double tiene 52 bits en su parte de fracción, por lo que  $\text{sys.float\_info.epsilon} = 2^{-52} = 2.220446049250313 \times 10^{-16}$  es la mayor precisión posible.

Ver: [http://en.wikipedia.org/wiki/Machine\\_epsilon](http://en.wikipedia.org/wiki/Machine_epsilon)

```
1 import sys
2
3 print('sys.float_info.epsilon = ', sys.float_info.epsilon)
4
5 if 1.0 + sys.float_info.epsilon > 1.0:
6     print('1.0+eps > 1.0');
7 if 1.0 + sys.float_info.epsilon/2 > 1.0:
8     print('1.0+eps/2 > 1.0');
```

Line: 10 of 10 Col: 1

LINE INS

```
daa@heimdall ~ $ python3 02_epsilon.py
sys.float_info.epsilon = 2.220446049250313e-16
1.0+eps > 1.0
daa@heimdall ~ $
```

No todos los números tienen una representación exacta en el formato de coma flotante. Un número como 0.1 no puede representarse exactamente como flotante. Su mantisa, que vale  $1/10$ , corresponde a la secuencia periódica de bits

0.00011001100110011001100110011001100110011001100110011

No hay, pues, forma de representar  $1/10$  con los 52 bits del formato de doble precisión. En base 10, los 52 primeros bits de la secuencia nos proporcionan el valor

0.10000000000000000055511151231257827021181583404541015625

Es lo más cerca de  $1/10$  que podemos estar.

Una peculiaridad adicional de los números codificados con la norma IEEE 754 es que su precisión es diferente según el número representado: cuanto más próximo a cero, mayor es la precisión. Para números muy grandes se pierde tanta precisión que no hay decimales (¡ni unidades, ni decenas...!). Por ejemplo, el resultado de la suma  $100000000.0+0.000000001$  es  $100000000.0$ , y no  $100000000.000000001$ , como cabría esperar.

A modo de conclusión, has de saber que al trabajar con números flotantes es posible que se produzcan pequeños errores en la representación de los valores y durante los cálculos. Probablemente esto te sorprenda, pues es *vox populi* que «los ordenadores nunca se equivocan».

```
In [1]: 0.1+0.1+0.1
Out[1]: 0.30000000000000004
```

```
In [8]: 10000000+0.0000000001
Out[8]: 1000000.0000000001
```

```
In [9]: 10000000+0.0000000001
Out[9]: 10000000.0
```



If I run:

```
>>> import math
>>> print(math.pi)
3.141592653589793
```

Then pi is printed with 16 digits,

However, according to:

```
>>> import sys
>>> sys.float_info.dig
15
```

My precision is 15 digits.

So, should I rely on the last digit that print() is showing? (i.e. the last 3)

Respuesta corta: `sys.float_info.dig` reporta el número de dígitos que son siempre correctos. El dígito #16 casi siempre es correcto, pero no siempre lo es.

Respuesta larga:

<http://stackoverflow.com/questions/28493114/precision-of-repr-strf-printf-when-f-is-float>

<http://stackoverflow.com/questions/18409496/is-it-52-or-53-bits-of-floating-point-precision>

# Flotantes con mayor precisión

- La librería `numpy` cuenta con el tipo de dato `Float128`: tiene 34 dígitos de precisión. El número más pequeño///grande que puede representar es  $3.3621 \times 10^{-4932}$ /// $1.1897 \times 10^{4932}$ .
- Si se requieren mayores precisiones se pueden utilizar las librerías:
  - `bigfloat`
  - `mpmath`
  - `gmpy`



# Representación de números flotantes en el PC

Ver detalles en:

- David Goldberg (1991). What Every Computer Scientist Should Know About Floating-Point Arithmetic. Computing Surveys, March 1991.
  - <https://ece.uwaterloo.ca/~dwharder/NumericalAnalysis/02Numerics/Double/paper.pdf>
- [http://en.wikipedia.org/wiki/Floating\\_point](http://en.wikipedia.org/wiki/Floating_point)
- [http://en.wikipedia.org/wiki/IEEE\\_floating\\_point](http://en.wikipedia.org/wiki/IEEE_floating_point) (IEEE 754, 1985)
- [http://en.wikipedia.org/wiki/Single-precision\\_floating-point\\_format](http://en.wikipedia.org/wiki/Single-precision_floating-point_format)
- [http://en.wikipedia.org/wiki/Double-precision\\_floating-point\\_format](http://en.wikipedia.org/wiki/Double-precision_floating-point_format)
- [http://en.wikipedia.org/wiki/Quadruple-precision\\_floating-point\\_format](http://en.wikipedia.org/wiki/Quadruple-precision_floating-point_format)

# Infinito y NaN

```
>>> infinito = float('inf')
>>> infinito
inf
>>> infinito = float('Infinity')
>>> infinito
inf
>>> infinito/1000 + 10
inf
>>> 1/0
Traceback (most recent call last):
  File "<pyshell#21>", line 1, in <module>
    1/0
ZeroDivisionError: division by zero
>>> infinito = float('infinito')
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    infinito = float('infinito')
ValueError: could not convert string to float: 'infinito'
>>> NaN = float('nan')
>>> NaN
nan
>>> 1 + NaN
nan
>>> NaN == NaN
False
>>> NaN != NaN
True
>>> 1 > NaN
False
>>> NaN > 1
False
>>> 1e308
1e+308
>>> 1e309
inf
>>>
```

NOTA: en MATLAB 1/0 da infinito, no una excepción "ZeroDivisionError" como en Python

# Verificando el tipo de una variable

```
>>> type(1)
<class 'int'>
>>> type(2+3j)
<class 'complex'>
>>> type('a')
<class 'str'>
>>> type("a")
<class 'str'>
>>> type(21.0)
<class 'float'>
>>> 1+2
3
>>> 1+2.0
3.0
>>> isinstance(1, int)
True
```

```
>>> (1+2j).real
1.0
>>> (1+2j).imag
2.0
>>> type(1+2j)
<class 'complex'>
>>> type(1j.real)
<class 'float'>
```

# Convirtiendo flotantes a enteros y viceversa

```
>>> int(2.32)
2
>>> int(4.87)
4
>>> int(-5.66)
-5
>>> float(2)
2.0
>>> type(_)
<class 'float'>
>>> |
```

Observe que la función `int()` trunca el número (lo redondea hacia cero sin decimales). La función `math.trunc()` hace lo mismo que `int()` en este caso.

La variable `_` funciona como el Ans de la calculadora.  
En este caso se refiere al 2

# Convirtiendo entre bases numéricas

```
>>> bin(17)
'0b10001'
>>> bin(127)
'0b1111111'
>>> oct(529)
'0o1021'
>>> oct(-529)
'-0o1021'
>>> hex(2569)
'0xa09'
>>> hex(25)
'0x19'
>>> hex(0x3FAB34C)
'0x3fab34c'
>>> hex(0b011000011110)
'0x61e'
>>> oct(0b011000011110)
'0o3036'
>>> |
```

```
>>> int('11110000', base = 2)
240
>>> int('11110000', base = 3)
3240
>>> int('11110000', base = 5)
97500
>>> int('11110000', base = 10)
11110000
>>> int('11110000', base = 20)
1347360000
>>> int('11110000', base = 360)
Traceback (most recent call last):
  File "<pyshell#42>", line 1, in <module>
    int('11110000', base = 360)
ValueError: int() base must be >= 2 and <= 36
```

# Operadores

- Los **operadores** especifican como se puede manipular un objeto/variable
  - Aritméticos: `+` `-` `*` `/` `//` `%`
  - De asignación: `=` `+=` `-=` `*=` `/=` `%=` `&=` `|=`
  - Relacionales: `>` `<` `>=` `<=` `==` `!=`
  - Lógicos: `and` `or` `not`
  - Bit a bit: `&` `|` `^` `~` `>>` `<<`
  - Especiales: `.` `**` `in` `not in` `is` `is not`

# Operaciones aritméticas, asignaciones

- Operaciones aritméticas binarias

$x+y$      $x-y$      $x*y$      $x/y$      $x\%y$

- Operaciones aritméticas binarias con  
asignación

$x += y$      $x -= y$      $x *= y$      $x /= y$   
                   $x \%= y$

- Asignaciones (variable = expresión):

$y = x + 4*y/(x - 2) + y$

# Operadores de asignación simplificada “a = a operador b”

```
>>> a = 3
>>> a += 2
>>> a
5
>>> a -= 3
>>> a
2
>>> a /= 4
>>> a
0.5
>>> a *= 10
>>> a
5.0
>>> a **= 3
>>> a
125.0
>>> b = 'Buenos '
>>> b += 'días!'
>>> b
'Buenos días!'
```

Otros operadores de  
asignación son: **+=**, **-=**,  
**\*=**, **/=**, **&=**, **//=**, **<<=**,  
**>>=**, **%=**, **|=**, **\*\*=**, **^=**

a += b	es lo mismo que	a = a + b
a -= b		a = a - b
a *= b		a = a * b
a /= b		a = a / b
a //= b		a = a // b
a **= b		a = a ** b
etc...		



# Operadores aritméticos

Syntax	Description
$x + y$	Adds number $x$ and number $y$
$x - y$	Subtracts $y$ from $x$
$x * y$	Multiplies $x$ by $y$
$x / y$	Divides $x$ by $y$ ; always produces a float (or a complex if $x$ or $y$ is complex)
$x // y$	Divides $x$ by $y$ ; truncates any fractional part so always produces an int result; see also the <code>round()</code> function
$x \% y$	Produces the modulus (remainder) of dividing $x$ by $y$
$x ** y$	Raises $x$ to the power of $y$ ; see also the <code>pow()</code> functions
$-x$	Negates $x$ ; changes $x$ 's sign if nonzero, does nothing if zero
$+x$	Does nothing; is sometimes used to clarify code
<code>abs(x)</code>	Returns the absolute value of $x$
<code>divmod(x, y)</code>	Returns the quotient and remainder of dividing $x$ by $y$ as a tuple of two ints
<code>pow(x, y)</code>	Raises $x$ to the power of $y$ ; the same as the <code>**</code> operator
<code>pow(x, y, z)</code>	A faster alternative to $(x ** y) \% z$
<code>round(x, n)</code>	Returns $x$ rounded to $n$ integral digits if $n$ is a negative int or returns $x$ rounded to $n$ decimal places if $n$ is a positive int; the returned value has the same type as $x$ ; see the text

```
>>> 100/3
33.333333333333336
>>> 100//3
33
>>> -100//3
-34
>>> divmod(100,3)
(33, 1)
>>> 10**3
1000
>>> pow(10,3)
1000
>>> round(12.3486,2)
12.35
>>> (10 ** 3) % 6
4
>>> pow(10, 3, 6)
4
```

```
>>> round(123456.789,2)
123456.79
>>> round(123456.789,-2)
123500.0
```

# División entera //

```
>>> 27*7
189
>>> 27//7
3
>>> -27//7
-4
>>> -27.0//7
-4.0
>>> -27//7.0
-4.0
>>> 27//7.0
3.0
>>> 27/7
3.857142857142857
>>> -27/7
-3.857142857142857
```

Observe que este operador redondea hacia menos infinito, no hacia 0 como lo hace el lenguaje C. Si ambos números son enteros retorna un entero. Si un número es un float, retorna un float.

```

>>> radio = 2
>>> math.pi
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    math.pi
NameError: name 'math' is not defined
>>> import math
>>> math.pi
3.141592653589793
>>> radio = 2
>>> area = math.pi*radio**2
>>> area
12.566370614359172
>>> _ + 100
112.56637061435917
>>> 1+3i
SyntaxError: invalid syntax
>>> 1+3*i
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    1+3*i
NameError: name 'i' is not defined
>>> 1+3j
(1+3j)
>>> 1+3J
(1+3j)
>>> (1+3j)*(1-3j)
(10+0j)

```

La constante pi

La variable \_ funciona como el Ans de la calculadora

Python soporta números complejos

```

>>> x, y, z = 1, 2, 3
>>> x
1
>>> y
2
>>> z
3
>>> |

```

Múltiples  
asignaciones en  
una sola línea

# Fraccionarios

Las operaciones con fraccionarios son más lentas que con flotantes.

```
>>> import fractions
>>> x = fractions.Fraction(1, 3)
>>> x
Fraction(1, 3)
>>> 2*x
Fraction(2, 3)
>>> x*3
Fraction(1, 1)
>>> x + x
Fraction(2, 3)
>>> fractions.Fraction(6,4)
Fraction(3, 2)
>>> fractions.Fraction(6,0)
Traceback (most recent call last):
  File "<pyshell#96>", line 1, in <module>
    fractions.Fraction(6,0)
  File "/usr/lib/python3.4/fractions.py", line 167, in __new__
    raise ZeroDivisionError('Fraction(%s, 0)' % numerator)
ZeroDivisionError: Fraction(6, 0)
>>> fractions.Fraction(0,0)
Traceback (most recent call last):
  File "<pyshell#97>", line 1, in <module>
    fractions.Fraction(0,0)
  File "/usr/lib/python3.4/fractions.py", line 167, in __new__
    raise ZeroDivisionError('Fraction(%s, 0)' % numerator)
ZeroDivisionError: Fraction(0, 0)
>>> |
```

```

1 import time
2 import fractions
3 from math import log
4
5 suma = 0
6 tic = time.time()
7 ▼ for i in range(0, 10000):
8     suma += fractions.Fraction(1,2*i + 1) - fractions.Fraction(1,2*i + 2)
9 toc = time.time()
10 print('ln(2) = ', float(suma))
11 print('El cálculo se realizó en ', toc-tic, 'segundos')
12
13 suma = 0
14 tic = time.time()
15 ▼ for i in range(0,10000):
16     suma += 1/(2*i + 1) - 1/(2*i + 2);
17 toc = time.time()
18 print('ln(2) = ', suma)
19 print('El cálculo se realizó en ', toc-tic, 'segundos')
20
21 print('ln(2) = ', log(2))

```

$$\frac{1}{1} - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5} \cdots = \ln(2)$$

El cálculo con fractions es mucho más lento que el cálculo con floats: en este ejemplo aproximadamente 1430 veces más lento

Line 22, Column 1

INSERT Soft Tabs: 4 UTF-8 Python

```

daalvarez@eredron ~ $ python3 02_tiempo_calculo.py
ln(2) = 0.6931221811849453
El cálculo se realizó en 4.225690603256226 segundos
ln(2) = 0.6931221811849471
El cálculo se realizó en 0.002948284149169922 segundos
ln(2) = 0.6931471805599453
daalvarez@eredron ~ $

```

```

1  #include <stdio.h>
2  #include <sys/time.h>
3  #include <unistd.h>
4
5  double intervalo_tiempo_ms(struct timeval toc, struct timeval tic);
6
7  int main() {
8      struct timeval tic, toc;
9      double x, dif, suma = 0;
10
11     gettimeofday(&tic, NULL);...
12     for (int i=0; i < 100000000; i++)
13         suma += 1.0/(2*i + 1) - 1.0/(2*i + 2);
14     gettimeofday(&toc, NULL);
15
16     dif = intervalo_tiempo_ms(toc, tic);
17
18     printf("ln(2) = %g\n", suma);
19     printf("El cálculo se realizó en %g segundos.\n", dif);
20
21     return 0;
22 }
23
24 double intervalo_tiempo_ms(struct timeval toc, struct timeval tic)
25 {
26     return (toc.tv_sec - tic.tv_sec) + (toc.tv_usec - tic.tv_usec)/1000000.0;
27 }

```

En este ejemplo lenguaje C es 44.9 veces más rápido que Python. Se usaron  $10^7$  iteraciones

Line 28, Column 1

INSERT Soft Tabs: 4 UTF-8 C

```

ln(2) = 0.6931471555618245
El cálculo se realizó en 3.3613147735595703 segundos

```

```
ln(2) = 0.6931471805599453
```

```
daalvarez@eredron ~ $ gcc 02_tiempo_calculo.c -o 02_tiempo_calculo -O3
```

```
daalvarez@eredron ~ $ ./02_tiempo_calculo
```

```
ln(2) = 0.693147
```

```
El cálculo se realizó en 0.074768 segundos.
```

```
daalvarez@eredron ~ $ python3 -c 'print(3.3613148/0.074768)'
```

```
44.95659640487909
```



# El módulo math

```
>>> import math
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos',
'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign',
'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs',
'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'hypot', 'isfinite',
'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2',
'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
>>> math.pi
3.141592653589793
>>> math.sin(math.pi/2)
1.0
>>> math.tan(math.pi/4)
0.9999999999999999
>>> 0/0
Traceback (most recent call last):
  File "<pyshell#111>", line 1, in <module>
    0/0
ZeroDivisionError: division by zero
>>> |
```

NOTA: utilice el módulo **cmath** si  
piensa trabajar con números complejos.

Syntax	Description
<code>math.acos(x)</code>	Returns the arc cosine of $x$ in radians
<code>math.acosh(x)</code>	Returns the arc hyperbolic cosine of $x$ in radians
<code>math.asin(x)</code>	Returns the arc sine of $x$ in radians
<code>math.asinh(x)</code>	Returns the arc hyperbolic sine of $x$ in radians
<code>math.atan(x)</code>	Returns the arc tangent of $x$ in radians
<code>math.atan2(y, x)</code>	Returns the arc tangent of $y / x$ in radians
<code>math.atanh(x)</code>	Returns the arc hyperbolic tangent of $x$ in radians
<code>math.ceil(x)</code>	Returns $\lceil x \rceil$ , i.e., the smallest integer greater than or equal to $x$ as an int; e.g., <code>math.ceil(5.4) == 6</code>
<code>math.copysign(x, y)</code>	Returns $x$ with $y$ 's sign
<code>math.cos(x)</code>	Returns the cosine of $x$ in radians
<code>math.cosh(x)</code>	Returns the hyperbolic cosine of $x$ in radians
<code>math.degrees(r)</code>	Converts float $r$ from radians to degrees
<code>math.e</code>	The constant $e$ ; approximately 2.718 281 828 459 045 1
<code>math.exp(x)</code>	Returns $e^x$ , i.e., <code>math.e ** x</code>
<code>math.fabs(x)</code>	Returns $ x $ , i.e., the absolute value of $x$ as a float



<code>math.factorial(x)</code>	Returns $x!$
<code>math.floor(x)</code>	Returns $\lfloor x \rfloor$ , i.e., the largest integer less than or equal to $x$ as an int; e.g., <code>math.floor(5.4) == 5</code>
<code>math.fmod(x, y)</code>	Produces the modulus (remainder) of dividing $x$ by $y$ ; this produces better results than <code>%</code> for floats
<code>math.frexp(x)</code>	Returns a 2-tuple with the mantissa (as a float) and the exponent (as an int) so, $x = m \times 2^e$ ; see <code>math.ldexp()</code>
<code>math.fsum(i)</code>	Returns the sum of the values in iterable $i$ as a float
<code>math.hypot(x, y)</code>	Returns $\sqrt{x^2 + y^2}$
<code>math.isinf(x)</code>	Returns True if float $x$ is $\pm \text{inf}$ ( $\pm \infty$ )
<code>math.isnan(x)</code>	Returns True if float $x$ is nan (“not a number”)
<code>math.ldexp(m, e)</code>	Returns $m \times 2^e$ ; effectively the inverse of <code>math.frexp()</code>
<code>math.log(x, b)</code>	Returns $\log_b x$ ; $b$ is optional and defaults to <code>math.e</code>
<code>math.log10(x)</code>	Returns $\log_{10} x$
<code>math.log1p(x)</code>	Returns $\log_e(1 + x)$ ; accurate even when $x$ is close to 0
<code>math.modf(x)</code>	Returns $x$ ’s fractional and whole parts as two floats

Syntax	Description
<code>math.pi</code>	The constant $\pi$ ; approximately 3.1415926535897931
<code>math.pow(x, y)</code>	Returns $x^y$ as a float
<code>math.radians(d)</code>	Converts float <code>d</code> from degrees to radians
<code>math.sin(x)</code>	Returns the sine of <code>x</code> in radians
<code>math.sinh(x)</code>	Returns the hyperbolic sine of <code>x</code> in radians
<code>math.sqrt(x)</code>	Returns $\sqrt{x}$
<code>math.tan(x)</code>	Returns the tangent of <code>x</code> in radians
<code>math.tanh(x)</code>	Returns the hyperbolic tangent of <code>x</code> in radians
<code>math.trunc(x)</code>	Returns the whole part of <code>x</code> as an int; same as <code>int(x)</code>

# La razón de ser de las funciones `expm1()`, `log1p()` y `hypot()`

Se sugiere consultar los siguientes links:

- <https://www.johndcook.com/blog/2010/06/07/math-library-functions-that-seem-unnecessary/>
- <https://www.johndcook.com/blog/2010/06/02/whats-so-hard-about-finding-a-hypotenuse/>

```
>>> from math import hypot, sqrt
>>> x = 10**150
>>> y = 20**150
>>> sqrt(x**2 + y**2)
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    sqrt(x**2 + y**2)
OverflowError: int too large to convert to float
>>> hypot(x, y)
1.4272476927059599e+195
```

```
>>> sin(0)
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    sin(0)
NameError: name 'sin' is not defined
>>> from math import sin, cos
>>> sin(0)
0.0
>>> cos(0)
1.0
```

```
In [57]: math.fsum([0.1, 0.6, 0.3])
Out[57]: 1.0
```

```
>>> from math import pi, e
>>> pi
3.141592653589793
>>> e
2.718281828459045
>>> math.sin(pi/2)
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    math.sin(pi/2)
NameError: name 'math' is not defined
>>> import math
>>> math.sin(pi/2)
1.0
>>> |
```

```
>>> pi = 5
>>> pow = 3
>>> from math import *
>>> pi
3.141592653589793
>>> pow += 1
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    pow += 1
TypeError: unsupported operand type(s) for +=:
'builtin_function_or_method' and 'int'
>>> |
```

```
>>> from math import *
>>> sin(60*pi/180)
0.8660254037844386
>>> sin(radians(60))
0.8660254037844386
>>> sqrt(3)/2
0.8660254037844386
>>> |
```

# Redondeando números

```
>>> import math
>>> x = 2.4812345
>>> print(int(x), math.floor(x), math.ceil(x), round(x), round(x,3))
2 2 3 2 2.481
>>> x = 2.9123125
>>> print(int(x), math.floor(x), math.ceil(x), round(x), round(x,3))
2 2 3 3 2.912
>>> x = -2.4812345
>>> print(int(x), math.floor(x), math.ceil(x), round(x), round(x,3))
-2 -3 -2 -2 -2.481
>>> x = -2.9123125
>>> print(int(x), math.floor(x), math.ceil(x), round(x), round(x,3))
-2 -3 -2 -3 -2.912
>>> x = -2.5
>>> print(int(x), math.floor(x), math.ceil(x), round(x), round(x,3))
-2 -3 -2 -2 -2.5
>>> x = 2.5
>>> print(int(x), math.floor(x), math.ceil(x), round(x), round(x,3))
2 2 3 2 2.5
>>> x = 2
>>> print(int(x), math.floor(x), math.ceil(x), round(x), round(x,3))
2 2 2 2 2
>>> x = -2
>>> print(int(x), math.floor(x), math.ceil(x), round(x), round(x,3))
-2 -2 -2 -2 -2
```

# round() usa “round half to even”

En <https://docs.python.org/3/library/stdtypes.html#numeric-types-int-float-complex> dice:

```
round(x[, n])
```

`x` rounded to `n` digits, **rounding half to even**. If `n` is omitted, it defaults to 0.

Esto quiere decir que Python 3 (no sucede con Python 2) redondea al número par más proximo.

Según <https://en.wikipedia.org/wiki/Rounding> este tipo de redondeo es el definido por defecto en la norma IEEE 754 y lo recomiendan los estadistas para evitar sesgos en el tratamiento estadístico de datos.

```
for i in range(-6,6):  
    x = i + 0.5  
    print('% 3.1f ---> % 4.1f' % (x, round(x)))
```

Existen otros modos de redondeo, los cuales se pueden consultar en:

<https://en.wikipedia.org/wiki/Rounding>

```
-5.5 ---> -6.0  
-4.5 ---> -4.0  
-3.5 ---> -4.0  
-2.5 ---> -2.0  
-1.5 ---> -2.0  
-0.5 ---> 0.0  
0.5 ---> 0.0  
1.5 ---> 2.0  
2.5 ---> 2.0  
3.5 ---> 4.0  
4.5 ---> 4.0  
5.5 ---> 6.0
```

Este comando a veces no funciona adecuadamente:

**Note:** The behavior of `round()` for floats can be surprising: for example, `round(2.675, 2)` gives `2.67` instead of the expected `2.68`. This is not a bug: it's a result of the fact that most decimal fractions can't be represented exactly as a float. See [Floating Point Arithmetic: Issues and Limitations](#) for more information.

Ver:

- <https://stackoverflow.com/questions/33019698/how-to-properly-round-up-half-float-numbers-in-python>

# round() en otros lenguajes

Tenga en cuenta que el comportamiento de round() con el x.5 depende del lenguaje de programación empleado:

Language	a) Ties away from zero	b) Ties to even	Other
C (1999+) / C++ (2007+)	<code>round</code> and variants	via <code>rint</code> / <code>nearbyint</code> (depends on global rounding mode) , <code>roundeven</code> in future	
Python	2.x: <code>round</code>	3+: <code>round</code>	
Matlab/Octave	<code>round</code>		
R		<code>round</code>	
Mathematica		<code>Round</code>	
.NET (C#, F#, VB)	via <code>optional argument</code>	<code>Math.Round</code>	
Java			<code>round</code> : ties to +Inf
JavaScript			<code>round</code> : ties to +Inf
Fortran (77+)	<code>ANINT</code>		



# Implementando el “round half away from zero”

```
1 from math import copysign, floor
2
3 ▼ def mi_round(y):
4 ▼     '''
5     Implementa el método de redondeo
6     "Round half away from zero"
7     Ver:
8     https://en.wikipedia.org/wiki/Rounding
9     '''
10    return copysign(floor(abs(y)+0.5), y)
11    ....
12 ▼ for i in range(-6,6):
13     x = i + 0.5
14     print('% 3.1f ---> % 4.1f' % (x, mi_round(x)))
```

Line 16, Column 1

INSERT

```
daalvarez@eredron ~ $ python3 02_round_half_away_from_zero.py
```

```
-5.5 ---> -6.0
-4.5 ---> -5.0
-3.5 ---> -4.0
-2.5 ---> -3.0
-1.5 ---> -2.0
-0.5 ---> -1.0
0.5 ---> 1.0
1.5 ---> 2.0
2.5 ---> 3.0
3.5 ---> 4.0
4.5 ---> 5.0
5.5 ---> 6.0
```

Este tipo de redondeo se usa especialmente en aplicaciones financieras y es usado en MATLAB y MS EXCEL.

Ver: <https://en.wikipedia.org/wiki/Rounding>

$$q = \text{sgn}(y) \lfloor |y| + 0.5 \rfloor$$



# Precisión de los flotantes

Hemos dicho que los argumentos de las funciones trigonométricas deben expresarse en radianes. Como sabrás,  $\sin(\pi) = 0$ . Veamos qué opina Python:

```
>>> from math import sin, pi↵  
>>> sin(pi)↵  
1.2246467991473532e-16
```

El resultado que proporciona Python no es cero, sino un número muy próximo a cero: 0.000000000000000012246467991473532. ¿Se ha equivocado Python? No exactamente. Ya dijimos antes que los números flotantes tienen una precisión limitada. El número  $\pi$  está definido en el módulo matemático como 3.141592653589793115997963468544185161590576171875, cuando en realidad posee un número infinito de decimales. Así pues, no hemos pedido exactamente el cálculo del seno de  $\pi$ , sino el de un número próximo, pero no exactamente igual. Por otra parte, el módulo matemático hace cálculos mediante algoritmos que pueden introducir errores en el resultado. Fíjate en el resultado de esta sencilla operación:

```
>>> 0.1 - 0.3↵  
-0.19999999999999998
```

Los resultados con números en coma flotante deben tomarse como meras aproximaciones de los resultados reales.

# El módulo decimal

El módulo decimal da una solución al problema de la imprecisión de los flotantes. Esta imprecisión es inaceptable cuando manejamos, por ejemplo, dinero. Usar decimales es más lento que usar flotantes.

[illegible]

# Booleans

**True**=1 o **False**=0

```
>>> True + False
```

```
1
```

```
>>> False*True
```

```
0
```

```
>>> True - False
```

```
1
```

```
>>> True/False
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#10>", line 1, in <module>
```

```
    True/False
```

```
ZeroDivisionError: division by zero|
```

```
>>>
```

# Booleanos

```
1 def es_verdadero(x):
2     if x:
3         print(x, "es verdadero")
4     else:
5         print(x, "es falso")
6
7 es_verdadero(1)
8 es_verdadero(-1)
9 es_verdadero(1.1)
10 es_verdadero(0)
11 es_verdadero(0.0)
12
13 import fractions
14 es_verdadero(fractions.Fraction(1, 2))
15 es_verdadero(fractions.Fraction(0, 2))
```

Al igual que en lenguaje C, 0 es falso y cualquier valor diferente de 0 es verdadero

Line: 17 of 17 Col: 1      LINE    INS

daalvarez@eredron:~ > python3 02\_verdadero\_falso.py

```
1 es verdadero
-1 es verdadero
1.1 es verdadero
0 es falso
0.0 es falso
1/2 es verdadero
0 es falso
```

# bool()

```
>>> bool(0)
False
>>> bool(0.0)
False
>>> bool('') —————> Cadena vacía
False
>>> bool([]) —————> Lista vacía
False
>>> bool(()) —————> Tupla vacía
False
>>> bool({}) —————> Diccionario vacío
False
>>> bool(1)
True
>>> bool('Hola') —————> Cadena
True
>>> bool([1, 2, 3]) —————> Lista
True
>>> bool({1, 'x'}) —————> Conjunto
True
>>> bool({'nombre': 'Elia', 'código': 123})
True
      ↙
      Diccionario
```

NOTA: las listas, las cadenas, los conjuntos y los diccionarios los veremos más adelante.

# Condicionales

En Python cualquier valor diferente de 0 es verdadero; el cero es falso. Cuando se ponen cadenas o listas, cualquier lista con una longitud diferente de cero es verdadera. Cadenas, listas, tuplas, conjuntos o diccionarios vacíos son falsos.

Los operadores relacionales (de comparación) son los mismos que en lenguaje C: `>` `<` `>=`  
`<=` `==` `!=`

Los operadores lógicos son: `and` `or` `not`

# Comparaciones

```
>>> 10 > 2
True
>>> 10 < 2
False
>>> 10 == 10
True
>>> 10 != 10
False
>>> 'Hola' == 'hola'
False
>>> 'Hola' == 'Hola'
True
>>> 'Hola' != 'hola'
True
>>> 10 >= 10
True
```

# Ejemplo con los operadores lógicos and y or

Suponga que se ha disparado una bala, y esta está en la posición (pos\_bala\_x,pos\_bala\_y). Suponga que la nave se encuentra en las coordenadas (x,y). Si una bala impacta la nave, su vida se reduce en 1:

```
if (x == pos_bala_x) and (y == pos_bala_y):  
    vida -= 1
```

Suponga que el jugador se encuentra en la posición (x,y). El tablero tiene de 0 a XMAX-1 columnas y de 0 a YMAX-1 filas. Se verifica que la nave no se haya salido del tablero así:

```
if (x<0) or (x>=XMAX) or (y<0) or (y>=YMAX):  
    print('Se ha salido del tablero de juego')
```



# Tablas de verdad

and		
operandos		resultado
izquierdo	derecho	
True	True	True
True	False	False
False	True	False
False	False	False

or		
operandos		resultado
izquierdo	derecho	
True	True	True
True	False	True
False	True	True
False	False	False

not	
operando	resultado
True	False
False	True

```
>>> True and False
False
>>> True or False
True
>>> not False
True
>>> (False and True) or True
True
>>> |
```

# Encadenación de comparaciones

Los operadores se pueden encadenar, por lo que  $a < b == c$  es lo mismo que  $(a < b) \text{ and } (b == c)$

```
>>> 2 < 3 < 4
True
>>> 2 < 5 > 3
True
>>> 1 < 2 < 3 < 10 > 5
True
>>> 2 < 1 < 4
False
>>> 2 == 2 > 1 == 1
True
>>> 2 == 2 > 1 != 10
True
>>> 2 == 2 > -1 != -1
False
>>> |
```

NOTA: este tipo de encadenación no es posible realizarla en lenguaje C, C++, MATLAB, Pascal, entre otros. Esta notación propia de Python es muy elegante y seguro le costará deshabituarle de ella cuando aprenda otro lenguaje de programación.

# Condicionales

- Recuerde que según la precedencia de operadores los operadores se ejecutan en el siguiente orden (de mayor a menor precedencia):
  - in, not in, is, is not, <, <=, >, >=, !=, ==
  - not x
  - and
  - or

por lo que **A and not B or C**

es equivalente a **(A and (not B)) or C**

**NOTA:** el operador de comparación **==** es diferente del operador de asignación **=**

```
>>> a = 10
>>> a == 1
False
>>> a
10
```

# No compare floats con == compárelos con math.isclose()

```
math.isclose(a, b, *, rel_tol=1e-09, abs_tol=0.0)
```

Retorna **True** si los a y b son aproximadamente iguales, es decir si se cumple que:

$$|a - b| \leq \max(\text{rel\_tol} * \max(|a|, |b|), \text{abs\_tol})$$

Este comando  
apareció en  
Python 3.5

**rel\_tol** y **abs\_tol** son números positivos

**rel\_tol** es la tolerancia relativa. Por ejemplo un **rel\_tol**=0.05 dice que la diferencia entre ambos números es máximo del 5%. Un **rel\_tol**=1e-6 establece que aproximadamente 6 dígitos son iguales.

**abs\_tol** es la tolerancia absoluta. Se utiliza para comparaciones cerca a cero. Su valor depende del problema en particular; por ejemplo, si  $x - y \approx 0$ , **atol**=1e-9 es un valor muy pequeño si x es el radio del planeta Tierra medido en metros ( $6.3781 \times 10^6$  m), pero es un valor absurdamente grande si x es el radio del átomo de hidrógeno medido en metros ( $5.2918 \times 10^{-11}$  m).

Ver: <https://www.python.org/dev/peps/pep-0485/>

# No compare floats con ==

```
1 # Este programa enseña que no debemos comparar flotantes con ==
2
3 import math
4
5 # (x - sqrt(5))^2 = x^2 - 2*sqrt(5) x + 5
6 a = 1
7 b = -2*math.sqrt(5)          # a*x^2 + b*x + c = 0
8 c = 5;                      # x^2 - 2*sqrt(5) + 5 = 0
9
10 d2 = b**2 - 4.0*a*c; print("d2 = ", d2) # debe dar 0
11 d = math.sqrt(d2); print("d = ", d) # debe dar 0
12 x1 = (-b + d)/(2*a); print("x1 = ", x1)
13 x2 = (-b - d)/(2*a); print("x2 = ", x2)
14
15 ▼ if (x1 == x2):
16     print('x1 == x2')
17 ▼ else:
18     print('x1 != x2')
19
20 ▼ if (abs(x1-x2) < 1e-6): # esto es decir que abs_tol = 1e-6
21     print('x1 == x2')
22 ▼ else:
23     print('x1 != x2')
24
25 print(math.isclose(x1, x2, rel_tol=1e-6))
26 print(math.isclose(x1, x2, rel_tol=1e-7))
27 print(math.isclose(x1, x2, rel_tol=1e-8))
```

Compare floats  
utilizando una de  
estas soluciones

Line 28, Column 1

INSERT Soft Tabs: 4 UTF-8 Python

```
daalvarez@eredron ~ $ python3 02_compara_reales.py
```

```
d2 = 3.552713678800501e-15
```

```
d = 5.960464477539063e-08
```

```
x1 = 2.236068007302112
```

```
x2 = 2.2360679476974674
```

```
x1 != x2
```

```
x1 == x2
```

```
True
```

```
True
```

```
False
```

```
daalvarez@eredron ~ $
```

# Precedencia de operadores

Operación	Operador	Aridad	Asociatividad	Precedencia
Exponenciación	<b>**</b>	Binario	Por la derecha	1
Identidad	<b>+</b>	Unario	—	2
Cambio de signo	<b>-</b>	Unario	—	2
Multiplicación	<b>*</b>	Binario	Por la izquierda	3
División	<b>/</b>	Binario	Por la izquierda	3
División entera	<b>//</b>	Binario	Por la izquierda	3
Módulo (o resto)	<b>%</b>	Binario	Por la izquierda	3
Suma	<b>+</b>	Binario	Por la izquierda	4
Resta	<b>-</b>	Binario	Por la izquierda	4
Igual que	<b>==</b>	Binario	—	5
Distinto de	<b>!=</b>	Binario	—	5
Menor que	<b>&lt;</b>	Binario	—	5
Menor o igual que	<b>&lt;=</b>	Binario	—	5
Mayor que	<b>&gt;</b>	Binario	—	5
Mayor o Igual que	<b>&gt;=</b>	Binario	—	5
Negación	<b>not</b>	Unario	—	6
Conjunción	<b>and</b>	Binario	Por la izquierda	7
Disyunción	<b>or</b>	Binario	Por la izquierda	8

- Cualquier expresión diferente de cero es verdadero en C, mientras que si es cero se considera falsa.
- Con **and**: si la primera expresión es falsa, la segunda no se evalúa (esto se conoce como **evaluación en cortocircuito**):

```
if (x == 5) and (y == 10): print('x=5 y y=10')
```

- Con **or**: si la primera expresión es verdadera, la segunda no se evalúa:

```
if (x == 5) or (y == 10)): print('x=5 o y=10')
```

- Negación (**not**):

```
if not (x == 5): print('x es diferente de 5')
```

# Otras operaciones con secuencias de bits

Solo te hemos presentado los operadores que utilizaremos en el texto y que ya estás preparado para manejar. Pero has de saber que hay más operadores. Hay operadores, por ejemplo, que están dirigidos a manejar las secuencias de bits que codifican los valores enteros. El operador binario `&` calcula la operación «y» bit a bit, el operador binario `|` calcula la operación «o» bit a bit, el operador binario `^` calcula la «o exclusiva» (que devuelve cierto si y solo si los dos operandos son distintos), también bit a bit, y el operador unario `~` invierte los bits de su operando. Tienes, además, los operadores binarios `<<` y `>>`, que desplazan los bits a izquierda o derecha tantas posiciones como le indiques. Estos ejemplos te ayudarán a entender estos operadores:

En decimal		En binario	
Expresión	Resultado	Expresión	Resultado
5 & 12	4	00000101 & 00001100	00000100
5   12	13	00000101   00001100	00001101
5 ^ 12	9	00000101 ^ 00001100	00001001
5 << 1	10	00000101 << 00000001	00001010
5 << 2	20	00000101 << 00000010	00010100
5 << 3	40	00000101 << 00000011	00101000
5 >> 1	2	00000101 >> 00000001	00000010

**!Cuidado!**

No confunda

^ con \*\*



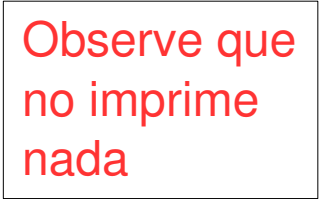
# None

**None** es una constante en Python que significa un valor nulo. Se utiliza para simbolizar que una variable no tiene un valor, o que el valor no existe, o que la referencia no apunta a ningún lado (en este caso es como el NULL del lenguaje C). No es lo mismo que False, no es 0, no es una cadena vacía, no es una lista vacía, etc. No puede compararse contra nada diferente de **None** y siempre retornará falso.

Se debe escribir **None**, no none, NONE, NoNe, etc.

```
>>> type(None)
<class 'NoneType'>
>>> None == False
False
>>> None == 0
False
>>> None == ''
False
>>> None == None
True
>>> x = None
>>> y = None
>>> x == None
True
>>> x == y
True
```

```
>>> x = None
>>> x
>>> print(x)
None
>>> type(x)
<class 'NoneType'>
```



# None en un contexto booleano

```
1 ▼ def es_verdadero(x):
2 ▼     if x:
3         print(x, "es verdadero")
4 ▼     else:
5         print(x, "es falso")
6
7     es_verdadero(None)
8     es_verdadero(not None)
```

< Line: 11 of 60 Col: 1 LINE INS

```
daa@heimdall ~ $ python3 02_verdadero_falso.py
None es falso
True es verdadero
```

# Constantes

- En programación, una constante es un valor que no puede (o no debe) ser alterado durante la ejecución de un programa. Esto en comparación a las variables, cuyo valor pueden cambiar durante la ejecución normal del programa.
- Las constantes no existen en Python, pero si son comunes en otros lenguajes de programación como Pascal, C o C++.
- Se sugiere (no es obligatorio, pero es una costumbre) escribir el nombre de las constantes en MAYÚSCULAS, para distinguirlo del resto de variables.

# Tamaño de las variables en memoria

Varían de implementación a implementación del interpretador, por lo que no se puede fiar de este número para los cálculos.

```
>>> import sys
>>> x = 123
>>> sys.getsizeof(x)
28
>>> sys.getsizeof(123)
28
>>> sys.getsizeof('abc')
52
>>> sys.getsizeof('abcd')
53
>>> x = 2**1000
>>> sys.getsizeof(x)
160
>>> sys.getsizeof([1, 2, 3, 'r'])
96
```

# Precedencia de operadores

Operator	Description
<code>lambda</code>	Lambda expression
<code>if - else</code>	Conditional expression
<code>or</code>	Boolean OR
<code>and</code>	Boolean AND
<code>not x</code>	Boolean NOT
<code>in, not in, is, is not, &lt;, &lt;=, &gt;, &gt;=, !=, ==</code>	Comparisons, including membership tests and identity tests
<code> </code>	Bitwise OR
<code>^</code>	Bitwise XOR
<code>&amp;</code>	Bitwise AND
<code>&lt;&lt;, &gt;&gt;</code>	Shifts
<code>+, -</code>	Addition and subtraction
<code>*, /, //, %</code>	Multiplication, division, remainder <a href="#">[5]</a>
<code>+X, -X, ~X</code>	Positive, negative, bitwise NOT
<code>**</code>	Exponentiation <a href="#">[6]</a>
<code>x[index], x[index:index], x(arguments...), x.attribute</code>	Subscription, slicing, call, attribute reference
<code>(expressions...), [expressions...], {key: value...}, {expressions...}</code>	Binding or tuple display, list display, dictionary display, set display

Operators in the same box group left to right (except for exponentiation, which groups from right to left).

The power operator `**` binds less tightly than an arithmetic or bitwise unary operator on its right, that is, `2**-1` is 0.5.

# Continuación de línea

```
1 x = 2 +  
2     3
```

Line: 4 of 4 Col: 1 LINE INS

```
daalvarez@eredron:~ > python3 02_continuacion.py
```

```
File "02_continuacion.py", line 1
```

```
x = 2 +  
      ^
```

SyntaxError: invalid syntax

```
1 x = 2 + \  
2     3  
3  
4 print(x)
```

\ e inmediatamente  
salto de línea  
(ENTER)

Line: 6 of 6 Col: 1 LINE INS

```
daalvarez@eredron:~ > python3 02_continuacion.py
```

```
5
```

```
1 x = 2 + \  
2     3  
3  
4 print(x)
```

Line: 6 of 6 Col: 1 LINE INS

```
daalvarez@eredron:~ > python3 02_continuacion.py
```

```
File "02_continuacion.py", line 1
```

```
x = 2 + \  
      ^
```

SyntaxError: unexpected character after line continuation character

```
daalvarez@eredron:~ >
```

Este tipo de error sucede si hacemos \  
espacio salto de línea (ENTER)

# Recolección de basura (garbage collection)

Es un mecanismo de gestión automática de la memoria de algunos lenguajes de programación el cual:

- Reserva los espacios de memoria
- Libera espacios de memoria previamente reservados (cuando ya no se necesitan)
- Compacta los espacios de memoria libre
- Lleva cuenta de qué espacios de memoria están libres y cuáles no.

## **Ventajas y desventajas**

- El programador no puede cometer errores y queda liberado de la tediosa tarea de gestionar la memoria.
- La memoria permanece retenida durante más tiempo del estrictamente necesario.
- El recolector de basura tarda cierto tiempo en hacer su tarea y produce pausas que pueden hacer la técnica incompatible con sistemas de tiempo real.

# Recolección de basura (garbage collection)

- En Python existe la librería `gc` que permite manejar el recolector de basura de forma manual.
- Like other memory management techniques, garbage collection may take a significant proportion of total processing time in a program and, as a result, can have significant influence on performance.
- El recogedor de basura utiliza un **contador de referencias** (reference counting) para saber cuando se debe borrar un objeto de la memoria. Reference counting is a simple technique in which objects are deallocated when there is no reference to them in a program.



# Referencias

- Wikipedia
- <http://www.inventwithpython.com/>
- <http://www.diveintopython3.net/>
- Documentación de Python:
  - <https://docs.python.org/3/tutorial/index.html>
  - <https://docs.python.org/3/>
- Marzal Varó, Andrés; Gracia Luengo, Isabel; García Sevilla, Pedro (2014). Introducción a la programación con Python 3. Disponible en: <http://dx.doi.org/10.6035/Sapientia93>
- <https://realpython.com/python-data-types/>
- <https://realpython.com/python-variables/>
- <https://realpython.com/python-rounding/>