Python de cero a experto

Autor: Luis Miguel de la Cruz Salas

Pythonico es más bonito

Expresiones y declaraciones

En matemáticas se define una expresión como una colección de símbolos que juntos expresan una cantidad, por ejemplo, el perímetro de una circunferencia es $2\pi r$.

En Python una **expresión** está compuesta de una combinación válida de valores, operadores, funciones y métodos, que se puede evaluar y da como resultado al menos un valor. Una expresión puede estar del lado derecho de una asignación.

```
a = 2**32
```

Véase más en The Python language reference: Expressions y Python expressions .

En términos simples y generales se dice que **una expresión produce un valor**.

In [1]:
```
23  # Expresión simple
```

Out[1]: 23

In [2]:
```
len('Hola mundo') # Expresión que ejecuta una función
```

Out[2]: 10

In [3]:
```
# Otros ejemplos
x = 1
y = x + 2
y ** 3
```

Out[3]: 27

In [4]:
```
7 == 2 * 2 * 2
```

Out[4]: False

In [5]:
```
3.141592 * len('Luis')
```

Out[5]: 12.566368

Una **declaración** (*statement*) se puede pensar como el elemento autónomo más corto de un lenguaje de programación. Un programa se forma de una secuencia que contiene una o más

declaraciones. Una declaración contiene componentes internos, que pueden ser otras declaraciones y varias expresiones. Véase más en Simple statements, Compound statements y Python statements (wikipedia).

En términos simples y generales se dice que una declaración hace algo

In [6]:
```
# Esta declaración realiza una pregunta, no produce nada
if x < 0:
    x = 0
```

In [7]:
```
# Esta declaración ejecuta una función.
print('Hola')
```

```
Hola
```

Tipos y operadores

El tipo de un objeto se determina en tiempo de ejecución.

Tres tipos más usados:

Tipo	Ejemplo
Númerico	13, 3.1416, 1+5j
Cadena	"Frida", "Diego"
Lógico	True, False

Tipos númericos

Tres tipos de números:

1. Enteros
2. Reales
3. Complejos

1. Enteros

Son aquellos que carecen de parte decimal.

In [8]:
```
entero   = 13
print(entero)
print(type(entero))
```

```
13
<class 'int'>
```

In [9]:
```
import sys
sys.int_info
```

Out[9]:
```
sys.int_info(bits_per_digit=30, sizeof_digit=4)
```

2. Reales

Son aquellos que tienen una parte decimal.

In [10]:
```
pi = 3.141592
print(pi)
print(type(pi))
```

```
3.141592
<class 'float'>
```

In [11]:
```
sys.float_info
```

Out[11]:
```
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=5
3, epsilon=2.220446049250313e-16, radix=2, rounds=1)
```

3. Complejos

Son aquellos que tienen una parte real y una parte imaginaria, y ambas partes son números reales.

In [12]:
```
complejo = 12 + 5j # La parte imaginaria lleva una j al final
print(complejo)
print(type(complejo))
```

```
(12+5j)
<class 'complex'>
```

In [13]:
```
complejo.imag
```

Out[13]:
```
5.0
```

In [14]:
```
complejo.real
```

Out[14]:
```
12.0
```

In [15]:
```
complejo.conjugate()
```

Out[15]:
```
(12-5j)
```

In [16]:
```
complejo.imag
```

Out[16]:
```
5.0
```

Operadores Aritméticos

In [17]:
```
# Suma
1 + 2
```

Out[17]:
```
3
```

In [18]:
```
# Resta
5 - 32
```

Out[18]:
```
-27
```

In [19]:
```
# Multiplicación
3 * 3
```

Out[19]: 9

In [20]:
```
# División
3 / 2
```

Out[20]: 1.5

In [21]:
```
# Potencia
81 ** (1/2)
```

Out[21]: 9.0

Numeric types

In [22]:
```
# Precedencia de operaciones
1 + 2 * 3 + 4
```

Out[22]: 11

In [23]:
```
# Uso de paréntesis para modificar la precedencia
(1 + 2) * (3 + 4)
```

Out[23]: 21

In [24]:
```
6/2*(2+1)
```

Out[24]: 9.0

Operator precedence

In [25]:
```
# Operaciones entre tipos diferentes
a = 1
b = 2 * 3j
a + b
```

Out[25]: (1+6j)

Operadores de asignación

In [26]:
```
etiqueta = 1.0
suma = 1.0
suma += etiqueta  # Equivalente a : suma = suma + etiqueta
print(suma)
```

2.0

In [27]:
```
suma += 1
suma
```

Out[27]: 3.0

In [28]:
```
suma = 1
suma
```

Out[28]: 1

In [29]:
```
etiqueta =  4
resta = 16
resta -= etiqueta # Equivalente a : resta = resta - etiqueta
print(resta)
```

```
12
```

In [30]:
```
etiqueta = 2
mult = 12
mult *= etiqueta  # Equivalente a : mult = mult * etiqueta
print(mult)
```

```
24
```

In [31]:
```
etiqueta = 5
divide = 50
divide /= etiqueta  # Equivalente a : divide = divide / etiqueta
print(divide)
```

```
10.0
```

In [32]:
```
etiqueta = 2
pot = 3
pot **= etiqueta # Equivalente a : pot = pot ** etiqueta
print(pot)
```

```
9
```

In [33]:
```
etiqueta = 5
modulo = 50
modulo %= etiqueta # Equivalente a : modulo = modulo % etiqueta
print(modulo)
```

```
0
```

Cadenas

Para definir una cadena se utilizan comillas simples, comillas dobles o comillas triples.

In [34]:
```
ejemplo = 'este es un ejemplo usando \' \' '
print(ejemplo)
ejemplo = "este es un ejemplo usando \" \" "
print(ejemplo)
ejemplo = '''este es un ejemplo usando \'\'\' \''' '''
print(ejemplo)
```

```
este es un ejemplo usando ' '
este es un ejemplo usando " "
este es un ejemplo usando ''' '''
```

In [35]:
```
queja = '''
Desde muy niño
tuve que interrumpir mi educuación
para ir a la escuela
'''
print(queja)
```

```

Desde muy niño
tuve que interrumpir mi educuación
```

```
para ir a la escuela
```

In [36]:
```
# La cadena puede tener ' dentro de " "
poema = "Enjoy the moments now, because they don't last forever"
print(poema)
```

```
Enjoy the moments now, because they don't last forever
```

In [37]:
```
# La cadena puede tener " dentro de ' '
titulo = 'Python "pythonico" '
print(titulo)
```

```
Python "pythonico"
```

Indexación de las cadenas

cadena :	M	u	r	c	i	é	l	a	g	o
índice +:	0	1	2	3	4	5	6	7	8	9
índice -:	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

In [38]:
```
ejemplo = 'Murciélago'
```

In [39]:
```
ejemplo[5]
```

Out[39]:
```
'é'
```

In [40]:
```
len(ejemplo)
```

Out[40]:
```
10
```

In [41]:
```
ejemplo[10]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-41-6e397be07b0b> in <module>
----> 1 ejemplo[10]

IndexError: string index out of range
```

In [42]:
```
ejemplo[-4]
```

Out[42]:
```
'l'
```

In [43]:
```
ejemplo[6]
```

Out[43]:
```
'l'
```

Inmutabilidad de las cadenas

Las cadenas no se pueden modificar:

In [44]:
```
ejemplo[5] = "e"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-44-ec417be3eaef> in <module>
----> 1 ejemplo[5] = "e"

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

Acceso a porciones de las cadenas (*slicing*)

Se puede obtener una subcadena a partir de la cadena original. La sintaxis es la siguiente:

```
cadena[Start:End:Stride]
```

Start :Índice del primer caracter para formar la subcadena.

End : Índice (menos uno) que indica el caracter final de la subcadena.

Stride: Salto entre elementos.

In [45]:
```
ejemplo[:] # Cadena completa
```

Out[45]: 'Murciélago'

In [46]:
```
ejemplo[0:3]
```

Out[46]: 'Mur'

In [47]:
```
ejemplo[::2]
```

Out[47]: 'Mrilg'

In [48]:
```
ejemplo[1:8:2]
```

Out[48]: 'ucéa'

In [49]:
```
ejemplo[::-1]
```

Out[49]: 'ogaléicruM'

Operaciones básicas con cadenas

Los operadores: `+` y `*` están definidos para las cadenas.

In [50]:
```
'Luis' + ' ' + 'Miguel' # Concatenación
```

Out[50]: 'Luis Miguel'

In [51]:
```
'ABC' * 3 # Repetición
```

Out[51]: 'ABCABCABC'

Funciones aplicables sobre las cadenas

In [52]:
```
ejemplo = 'murcielago'
```

In [53]:
```
ejemplo.capitalize()
```

Out[53]:
```
'Murcielago'
```

In [55]:
```
ejemplo.center # posicionarse en la palabra center y teclear [Shift+Tab]
```

Out[55]:
```
<function str.center(width, fillchar=' ', /)>
```

In [56]:
```
print(ejemplo)
print(ejemplo.center(20,'-'))
print(ejemplo.upper())
print(ejemplo.find('e'))
print(ejemplo.count('g'))
print(ejemplo.isprintable())
```

```
murcielago
-----murcielago-----
MURCIELAGO
5
1
True
```

Construcción de cadenas con variables

In [57]:
```
edad = 15
nombre = 'Pedro'
apellido = 'Páramo'
peso = 70.5
```

In [60]:
```
datos = nombre + apellido + 'tiene' + str(15) + 'años y pesa ' + str(70.5)
datos
```

Out[60]:
```
'PedroPáramotiene15años y pesa 70.5'
```

In [61]:
```
datos = '{} {} tiene {} años y pesa {}'.format(nombre, apellido, edad, peso)
datos
```

Out[61]:
```
'Pedro Páramo tiene 15 años y pesa 70.5'
```

In [63]:
```
# f-strings (formatted string literals)
datos = f'{nombre} {apellido} tiene {edad} años y pesa {peso}'
datos
```

Out[63]:
```
'Pedro Páramo tiene 15 años y pesa 70.5'
```

Constantes

- `False` : de tipo Booleano.
- `True` : de tipo Booleano.
- `None` : El único valor para el tipo NoneType. Es usado frecuentemente para representar la ausencia de un valor, por ejemplo cuando no se pasa un argumento a una función.

- `NotImplemented` : es un valor especial que es regresado por métodos binarios especiales (por ejemplo `__eq__()` , `__lt__()` , `__add__()` , `__rsub__()` , etc.) para indicar que la operación no está implementada con respecto a otro tipo.
- Ellipsis: equivalente a `...` , es un valor especial usado mayormente en conjunción con la sintáxis de *slicing*.
- `__debug__` : Esta constante es verdadera si Python no se inició con la opción -O.

Las siguiente constantes son usadas dentro del intérprete interactivo (no se pueden usar dentro de programas ejecutados fuera del intérprete).

- `quit` (code=None)
- `exit` (code=None)¶
- `copyright`
- `credits`
- `license`

In [64]:
```
copyright
```

Out[64]:
```
Copyright (c) 2001-2020 Python Software Foundation.
All Rights Reserved.

Copyright (c) 2000 BeOpen.com.
All Rights Reserved.

Copyright (c) 1995-2001 Corporation for National Research Initiatives.
All Rights Reserved.

Copyright (c) 1991-1995 Stichting Mathematisch Centrum, Amsterdam.
All Rights Reserved.
```

Tipos lógicos

Es un tipo utilizado para realizar operaciones lógicas y puede tomar dos valores: **True** o **False**.

In [66]:
```
bandera = True
bandera
```

Out[66]:
```
True
```

Operadores lógicos

In [67]:
```
35 > 562
```

Out[67]:
```
False
```

In [68]:
```
32 >= 21
```

Out[68]:
```
True
```

In [69]:
```
12 < 34
```

Out[69]: True

In [70]:
```
12 <= 25
```

Out[70]: True

In [71]:
```
5 == 5
```

Out[71]: True

In [72]:
```
23 != 23
```

Out[72]: False

In [73]:
```
'aaa' == 'aaa'
```

Out[73]: True

Operaciones lógicas básicas

1. and
2. or
3. not

In [74]:
```
(5 < 32) and (63 > 32)
```

Out[74]: True

In [75]:
```
(2.32 < 21) and (23 > 63)
```

Out[75]: False

In [76]:
```
(32 == 32) or (5 < 31)
```

Out[76]: True

In [77]:
```
(32 == 21) or (31 < 5)
```

Out[77]: False

In [78]:
```
not True
```

Out[78]: False

In [79]:
```
not (32 != 32)
```

Out[79]: True

In [80]:
```
(0.4 - 0.3) == 0.1
```

Out[80]: False

Fuertemente Tipado

Esta característica impide que se realizen operaciones entre tipos no compatibles.

In [82]:
```
lógico = True
real   = 220.0
entero = 284
complejo = 1+1j
cadena = 'numeros hermanos'
```

In [83]:
```
lógico + real
```

Out[83]:
```
221.0
```

In [84]:
```
lógico + complejo
```

Out[84]:
```
(2+1j)
```

In [85]:
```
cadena + real  # Tipos no compatibles
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-85-46589d476860> in <module>
----> 1 cadena + real  # Tipos no compatibles

TypeError: can only concatenate str (not "float") to str
```

Conversión entre tipos (*casting*)

Operación para transformar un tipo en otro tipo compatible.

```
int()
```

Transforma objetos en enteros, siempre y cuando haya compatibilidad.

In [86]:
```
cadena = '1000'
print(type(cadena))
entero = int(cadena)
print(type(entero))
print(entero)
```

```
<class 'str'>
<class 'int'>
1000
```

In [87]:
```
flotante = 3.141592
entero  = int(flotante) # Trunca la parte decimal
print(entero)
```

```
3
```

In [88]:
```
complejo= 4-4j
entero = int(complejo) # Tipos NO COMPATIBLES
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
```

```
<ipython-input-88-2f4651a3398b> in <module>
      1 complejo= 4-4j
----> 2 entero = int(complejo) # Tipos NO COMPATIBLES

TypeError: can't convert complex to int
```

In [89]:
```
entero = int(True)
print(entero)
```

```
1
```

In [90]:
```
print(1 == True)
```

```
True
```

`str()`

Transforma objetos en cadenas, siempre y cuando haya compatibilidad.

In [91]:
```
entero = 1000
print(type(entero))
cadena = str(entero)
print(type(cadena))
print(cadena)
```

```
<class 'int'>
<class 'str'>
1000
```

In [92]:
```
complejo = 5+1j
print(complejo)
print(type(complejo))
cadena = str(complejo)
print(cadena)
print(type(cadena))
```

```
(5+1j)
<class 'complex'>
(5+1j)
<class 'str'>
```

`float()`

Transforma objetos en flotantes, siempre y cuando haya compatibilidad.

In [93]:
```
cadena = '3.141592'
print(cadena)
print(type(cadena))
real = float(cadena)
print(real)
print(type(real))
```

```
3.141592
<class 'str'>
3.141592
<class 'float'>
```

In [94]:
```
float(33)
```

Out[94]:
```
33.0
```

In [95]:
```
float(False)
```

Out[95]:
```
0.0
```

In [96]:
```
float(3+3j)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-96-55bff90a30a0> in <module>
----> 1 float(3+3j)

TypeError: can't convert complex to float
```

Función *Eval*

Es una función que permite evaluar una cadena str, como si se tratase de una expresión, siempre y cuando la expresión sea válida en Python.

In [97]:
```
suma = '300+800'
resultado = eval(suma)
print(resultado)
print(type(resultado))
```

```
1100
<class 'int'>
```

In [98]:
```
a = 220.1
resta = 'a - 220'
resultado = eval(resta)
print(resultado)
print(type(resultado))
```

```
0.09999999999999432
<class 'float'>
```

In [99]:
```
logica = '32 == 32'
resultado = eval(logica)
print(resultado)
print(type(resultado))
```

```
True
<class 'bool'>
```

In [100…
```
import math
formula = 'math.sin(0.25*math.pi)'
print(formula)
print(type(formula))
resultado = eval(formula)
print(resultado)
```

```
math.sin(0.25*math.pi)
<class 'str'>
0.7071067811865475
```

Formato en código ANSI

- Un código de formato ANSI lo forma el carácter Escape seguido por tres números enteros

separados por un punto y coma (;).

- El primero de estos números (un valor de 0 a 7) establece el estilo del texto (negrita, subrayado, etc); el segundo número (de 30 a 37) fija el color del texto y el último número (de 40 a 47) el color del fondo.
- El carácter Escape se puede expresar en octal "\033", en hexadecimal "\x1b", o bien, con chr(27).

Estilos	Código ANSI
Sin efecto	0
Negrita	1
Débil	2
Cursiva	3
Subrayado	4
Inverso	5
Oculto	6
Tachado	7

Color	Texto	Fondo
Negro	30	40
Rojo	31	41
Verde	32	42
Amarillo	33	43
Azul	34	44
Morado	35	45
Cian	36	46
Blanco	37	47

OJO: Algunos estilos no están soportados por todas las consolas

In [101…
```
print(chr(27)+"[0;31m"+"Texto en color rojo")
```
```
Texto en color rojo
```

In [102…
```
print("\x1b[1;32m"+"Texto en negrita de color verde")
```
```
Texto en negrita de color verde
```

In [103…
```
print("\033[4;35m"+"Texto subrayado de color morado")
```
```
Texto subrayado de color morado
```

In [104…
```
print("\033[1;34;46m"+"Texto en negrita de color azul con fondo cyan ")
```
```
Texto en negrita de color azul con fondo cyan
```

In [ ]:
```
```