Python Básico

Luis Miguel de la Cruz Salas

2023-01-12

Table of contents

Introducción

Este es un primer curso sobre las funcionalidades básicas de Python. Es un complemento al curso "Diseño de Cursos Interactivos con la Plataforma MACTI".

Los Jupyter Notebooks de este curso se pueden obtener en https://github.com/repomacti/macti/tutoriales/python_basico.

1 Definición de variables.

Objetivo. Explicar el concepto de variable, etiqueta, objetos y como se usan mediante algunos ejemplos.

Funciones de Python: - `print()`, `type()`, `id()`, `chr()`, `ord()`, `del()`

1.1 Variables.

- Son **símbolos** que permiten identificar la información que se almacena en la memoria de la computadora.
- Son **nombres** o **etiquetas** para los objetos que se crean en Python.
- Se crean con ayuda del operador de asignación `=`.
- No se tiene que establecer explícitamente el tipo de dato de la variable, pues esto se realiza de manera dinámica (tipado dinámico).

1.1.1 Ejemplos de variables válidas.

Los nombres de las variables: * pueden contener **letras**, **números** y **guiones bajos**, * deben comenzar con una letra o un guion bajo, * se distingue entre mayúsculas y minúsculas, es decir, `variable` y `Variable` son nombres diferentes.

A continuación se muestran algunos ejemplos.

```
_luis = "Luis Miguel de la Cruz"    # El nombre de la variable es _luis, el contenido es una
LuisXV = "Louis Michel de la Croix"
luigi = 25
luis_b = 0b01110 # Binario
luis_o = 0o12376 # Octal
luis_h = 0x12323 # Hexadecimal

# Sensibilidad a mayúsculas y minúsculas
# los siguientes nombres son diferentes
```

```
pi = 3.14
PI = 31416e-4
Pi = 3.141592
```

Podemos ver el contenido de la variable usando la función `print()`:

```
# El contenido de cada variable se imprime en renglones
# diferentes debido a que usamos el argumento sep='\n'
print(_luis, LuisXV, luigi, luis_b, luis_o, luis_h, pi, PI, Pi, sep='\n')
```

```
Luis Miguel de la Cruz
Louis Michel de la Croix
25
14
5374
74531
3.14
3.1416
3.141592
```

NOTA. Para saber más sobre la función `print()` revisa la sección XXX.

Para saber el tipo de objeto que se creo cuando se definieron las variables anteriores, podemos hacer uso de la función `type()`:

```
print(type(_luis), type(LuisXV), type(luigi),
      type(luis_b), type(luis_o), type(luis_h),
      type(pi), type(PI), type(Pi), sep = '\n')
```

```
<class 'str'>
<class 'str'>
<class 'int'>
<class 'int'>
<class 'int'>
<class 'int'>
<class 'float'>
<class 'float'>
<class 'float'>
```

También es posible usar la función `id()` para conocer el identificador en la memoria de cada objeto como sigue:

```
print(id(_luis), id(LuisXV), id(luigi),
      id(luis_b), id(luis_o), id(luis_h),
      id(pi), id(PI), id(Pi), sep = '\n')
```

```
139672517675408
139672517886160
94157725269672
94157725269320
139672517639248
139672517630576
139672517623920
139672517638992
139672518259056
```

Observa que cada objeto tiene un identificador diferente. Es posible que un objeto tenga más de un nombre, por ejemplo

```
luiggi = _luis
```

La etiqueta o variable `luiggi` hace referencia al mismo objeto que la variable `_luis`, y eso lo podemos comprbar usando la función `id()`:

```
print(id(luiggi))
print(id(_luis))
```

```
139672517675408
139672517675408
```

1.1.2 Ejemplos con Unicode.

Unicode: estándar para la codificación de caracteres, que permite el tratamiento informático, la transmisión y visualización de textos de muchos idiomas y disciplinas técnicas. Unicode intenta tener universalidad, uniformidad y unicidad. Unicode define tres formas de codificación bajo el nombre UTF (Unicode transformation format): UTF8, UTF16, UTF32. Véase https://es.wikipedia.org/wiki/Unicode

Python 3 utiliza internamente el tipo de datos `str` para representar cadenas de texto Unicode, lo que significa que se puede escribir y manipular texto en cualquier idioma sin preocuparte por la codificación. La compatibilidad con UTF-8 en Python significa que se puede leer, escribir y manipular archivos de texto en cualquier idioma, y también trabajar con datos provenientes de

fuentes diversas, como bases de datos, API web, etc., que pueden contener texto en diferentes idiomas y codificaciones.

A continuación se muestran algunos ejemplos.

```
compañero = 'Luismi' # puedo usar la ñ como parte del nombre de la variable
print(compañero)
```

```
Luismi
```

Los códigos Unicode de cada caracter se pueden dar en decimal o hexadecimal, por ejemplo para el símbolo π se tiene el código decimal `120587` y hexadecimal `0x1D70B`. La función `chr()` convierte ese código en el caracter correspondiente:

```
chr(0x1D70B)
```

```
' '
```

```
chr(120587)
```

```
' '
```

La función `ord()` obtiene el código Unicode de un caracter y lo regresa en decimal:

```
ord(' ')
```

```
120587
```

Podemos usar la función `print()` para realizar una impresión con formato como sigue:

```
  = 3.141592
print('{:04d} \t {} = {}'.format(ord(' '), ' ',  )) # Impresión en decimal
print('{:04o} \t {} = {}'.format(ord(' '), ' ',  )) # Impresión en octal
print('{:04x} \t {} = {}'.format(ord(' '), ' ',  )) # Impresión en hexadecimal
```

```
120587     = 3.141592
353413     = 3.141592
1d70b      = 3.141592
```

Podemos usar acentos:

```
México = 'El ombligo de la luna'
print(México)
```

```
El ombligo de la luna
```

Puedo saber el tipo de codificación que usa Python de la siguiente manera:

```
import sys
sys.stdout.encoding
```

```
'UTF-8'
```

También es posible obtener más información de los códigos unicode como sigue:

```
import unicodedata

u = chr(233) + chr(0x0bf2) + chr(6000) + chr(13231)
print('cadena : ', u)
print()
for i, c in enumerate(u):
    print('{} {:>5x} {:>3}'.format(c, ord(c), unicodedata.category(c)), end=" ")
    print(unicodedata.name(c))
```

```
cadena :  é

é    e9  Ll LATIN SMALL LETTER E WITH ACUTE
    bf2  No TAMIL NUMBER ONE THOUSAND
   1770  Lo TAGBANWA LETTER SA
   33af  So SQUARE RAD OVER S SQUARED
```

Véase: https://docs.python.org/3/howto/unicode.html

1.2 Asignación múltiple.

Es posible definir varias variables en una sola instrucción:

```
x = y = z = 25
```

```
print(type(x), type(y), type(z))
```

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

```
print(id(x), id(y), id(z))
```

```
94157725269672 94157725269672 94157725269672
```

Observa que se creó el objeto `25` de tipo `<class 'int'>` y los nombres `x`, `y` y `z` son etiquetas al mismo objeto, como se verifica imprimiendo el identificador de cada variable usando la función `id()`.

Podemos eliminar la etiqueta `x` con la función `del()`:

```
del(x)
```

Ahora ya no es posible hacer referencia al objeto `25` usando `x`:

```
print(x)
```

```
NameError: name 'x' is not defined
```

Pero si es posible hacer referencia al objeto `25` con los nombres `y` y `z`:

```
print(y,z)
```

```
25 25
```

Podemos hacer una asignación múltiples de objetos diferentes a variables diferentes:

```
x, y, z = 'eje x', 3.141592, 50
```

```
print(type(x), type(y), type(z))
```

```
<class 'str'> <class 'float'> <class 'int'>
```

```
print(id(x), id(y), id(z))
```

```
139672217233008 139672517638832 94157725270472
```

Como se observa, ahora las variables `x`, `y` y `z` hacen referencia a diferentes objetos, de distinto tipo.

1.2.1 Ejemplos de nombres NO válidos.

Los siguientes son ejemplos NO VALIDOS para el nombre de variables. Al ejecutar las celdas se obtendrá un error en cada una de ellas.

```
1luis = 20      # No se puede iniciar con un número
```

```
SyntaxError: invalid decimal literal (953519616.py, line 1)
```

```
luis$ = 8.2323  # No puede contener caractéres especiales
```

```
SyntaxError: invalid syntax (2653363214.py, line 1)
```

```
for = 35        # Algunos nombres ya están reservados
```

```
SyntaxError: invalid syntax (2521306807.py, line 1)
```

1.3 Palabras reservadas.

Tampoco es posible usar las palabras reservadas para nombrar variables. Podemos conocer las palabras reservadas como sigue:

```
help('keywords')
```

```
Here is a list of the Python keywords.  Enter any keyword to get more help.

False               class               from                or
None                continue            global              pass
```

```
True            def             if              raise
and             del             import          return
as              elif            in              try
assert          else            is              while
async           except          lambda          with
await           finally         nonlocal        yield
break           for             not
```

2 Expresiones y declaraciones.

Objetivo. Explicar el concepto de variable, etiqueta, objetos y como se usan mediante algunos ejemplos.

Funciones de Python: - `print()`, `type()`, `id()`, `chr()`, `ord()`, `del()`

2.1 Expresiones

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, variables, operadores, funciones y métodos, que se puede evaluar y **da como resultado al menos un valor**.

En esencia, una **expresión es cualquier cosa que pueda ser evaluada y producir un resultado**.

Las expresiones pueden ser simples o complejas, pero en general, representan un valor único, por ejemplo:

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

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

Veamos algunos ejemplos:

Expresiones simples

```
23
```

```
23
```

```
5 + 3
```

```
8
```

```
a = 5
a ** 2
```

```
25
```

Expresión que ejecuta una función

```
len('Hola mundo')
```

```
10
```

Expresiones usando operadores

```
# Otros ejemplos
x = 1
y = x + 2
z = y ** 3

print(x)
print(y)
print(z)

# Operación Booleana
7 == 2 * 2 * 2
```

```
1
3
27

False
```

Expresiones más complicadas

```
# Se combinan varias operaciones matemáticas
b = 2.14
c = 0.1 + 4j

(3.141592 * c + b) / a
```

```
(0.4908318400000001+2.5132736j)
```

Observa que en todos los ejemplos anteriores se produce al menos un valor como resultado de la ejecución de cada expresión.

2.2 Declaraciones

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.

En términos simples, una **declaración** es una **instrucción que realiza una acción**.

Puede ser una asignación de valor a una variable, una llamada a una función, una estructura de control de flujo (como un ciclo o una condición), una definición de función, etc.

Véase más en Simple statements, Compound statements y Python statements (wikipedia).

Veamos algunos ejemplos:

Declaración que hace una asignación

```
x = 0
```

Declaración usando un condicional

```
if x < 0:
    pass
```

Declaración que realiza un ciclo

```
for i in range(0,5):
    pass
```

Declaración de una función

```
def mult(a, b):
    return a * b
```

```
Here is a note
```

3 Tipos de datos básicos y operadores.

Objetivo. Explicar el concepto de variable, etiqueta, objetos y como se usan mediante algunos ejemplos.

Funciones de Python: - `print()`, `type()`, `id()`, `chr()`, `ord()`, `del()`

3.1 Tipos y operadores

En Python se tienen tres tipos de datos básicos principales:

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

3.2 Tipos númericos

En Python se tienen tres tipos de datos númericos: 1. Enteros 2. Reales 3. Complejos

A continuación se realiza una descripción de estos tipos numéricos. Más información se puede encontrar aquí: Numeric types.

1. Enteros

Son aquellos que carecen de parte decimal. Para definir un entero hacemos lo siguiente:

```
entero = 13
```

Cuando se ejecuta la celda anterior, se crea el objeto `13` cuyo nombre es `entero`. Podemos imprimir el valor de `entero` y su tipo como sigue:

```
print(entero)
print(type(entero))
```

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

Es posible obtener más información del tipo `int` usando la biblioteca `sys`:

```
import sys
sys.int_info
```

```
sys.int_info(bits_per_digit=30, sizeof_digit=4, default_max_str_digits=4300, str_digits_chec
```

2. Reales

Son aquellos que tienen una parte decimal. Para definir un número real (flotante) se hace como sigue:

```
pi = 3.141592
```

Cuando se ejecuta la celda anterior, se crea el objeto `3.141592` cuyo nombre es `pi`. Podemos imprimir el valor de `pi` y su tipo como sigue:

```
print(pi)
print(type(pi))
```

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

```
# para obtener más información:
sys.float_info
```

```
sys.float_info(max=1.7976931348623157e+308, max_exp=1024, max_10_exp=308, min=2.225073858507
```

3. Complejos

Son aquellos que tienen una parte real y una parte imaginaria, y ambas partes son números reales. Para definir un número complejo se hace como sigue:

```
complejo = 12 + 5j # La parte imaginaria lleva una j al final
```

Cuando se ejecuta la celda anterior, se crea el objeto `12 + 5j` cuyo nombre es `complejo`. En este caso, el contenido de `complejo` tiene dos partes: la real y la imaginaria. Podemos imprimir el valor de `complejo` y su tipo como sigue:

```
print(complejo)
print(type(complejo))
```

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

```
complejo.imag # accedemos a la parte imaginaria
```

```
5.0
```

```
complejo.real # accedemos a la parte real
```

```
12.0
```

```
complejo.conjugate() # calculamos el conjugado del número complejo.
```

```
(12-5j)
```

Nota: observa que hemos aplicado el método `conjugate()` al objeto `complejo`, esto es posible debido a que existe la clase `<class 'complex'>` en Python, y en ella se definen atributos y métodos para los objetos de esta clase. Más acerca de programación orientada a objetos la puedes ver en esta sección XXX.

3.2.1 Operadores aritméticos

Para los tipos numéricos descritos antes, existen operaciones aritméticas que se pueden aplicar sobre ellos. Veamos:

```
# Suma
1 + 2
```

```
# Resta
5 - 32
```

```
# Multiplicación
3 * 3
```

```
# División
3 / 2
```

```
# Potencia
81 ** (1/2)
```

3.2.2 Precedencia de operadores.

La aplicación de los operadores tiene cierta precedencia que está definida en cada implementación del lenguaje de programación. La siguiente tabla muestra el orden en que se aplicarán los operadores en una expresión.

Nivel	Categoría	Operadores
7	exponenciación	`**`
6	multiplicación	`*`,`/`,`//`,`%`
5	adición	`+`,`-`
4	relacional	`==`,`!=`,`<=`,`>=`,`>`,`<`
3	logicos	`not`
2	logicos	`and`
1	logicos	`or`

Como se puede ver, siempre se aplican primero los operadores aritméticos (niveles 7,6, y 5), luego los relacionales (nivel 4) y finalmente los lógicos (3, 2 y 1).

Más acerca de este tema se puede ver aquí: Operator precedence.

A continuación se muestran ejemplos de operaciones aritméticas en donde se resalta est precedencia:

```
# Precedencia de operaciones: primero se realiza la multiplicación
1 + 2 * 3 + 4
```

```
11
```

```
# Es posible usar paréntesis para modificar la precedencia: primero la suma
(1 + 2) * (3 + 4)
```

```
21
```

```
# ¿Puedes explicar el resultado de acuerdo con la precedencia
# descrita en la tabla anterior?
6/2*(2+1)
```

```
9.0
```

3.2.3 Operaciones entre tipos diferentes

Es posible combinar operaciones entre tipos de números diferentes. Lo que Python hará es promover cada número al tipo más sofisticado, siendo el orden de sofisticación, de menos a más, como sigue: `int`, `float`, `complex`.

```
a = 1 # un entero
b = 2 * 3j # un complejo
a + b  # resultará en un complejo
```

3.2.4 Operadores de asignación

Existen varios operadores para realizar asignaciones: `=`, `+=`, `-=`, `*=`, `/=`, `**=`, `%=`. La forma de uso de estos operadores se muestra en los siguientes ejemplos:

```
etiqueta = 1.0
suma = 1.0
suma += etiqueta  # Equivalente a : suma = suma + etiqueta
print(suma)
```

```
etiqueta =  4
resta = 16
resta -= etiqueta # Equivalente a : resta = resta - etiqueta
print(resta)
```

```
etiqueta = 2
mult = 12
mult *= etiqueta  # Equivalente a : mult = mult * etiqueta
print(mult)
```

```
etiqueta = 5
div = 50
div /= etiqueta  # Equivalente a : divide = divide / etiqueta
print(div)
```

```
etiqueta = 2
pot = 3
pot **= etiqueta # Equivalente a : pot = pot ** etiqueta
print(pot)
```

```
etiqueta = 5
modulo = 50
modulo %= etiqueta # Equivalente a : modulo = modulo % etiqueta
print(modulo)
```

3.3 Tipos lógicos

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

```
bandera = True
print(type(bandera))
```

3.3.1 Operadores relacionales

Cuando se aplica un operador relacional a dos expresiones, se realiza una comparación entre dichas expresiones y se obtiene como resultado un tipo lógico. `True` o `False`.

Los operadores relacionales que se pueden usar son: `==`,`!=`,`<=`,`>=`,`>`,`<`. A continuación se muestran algunos ejemplos:

```
35 > 562 # ¿Es 35 mayor que 562?
```

```
False
```

```
32 >= 21 # ¿Es 32 mayor o igual que 21?
```

```
True
```

```
12 < 34 # ¿Es 12 menor que 34?
```

```
True
```

```
12 <= 25 # ¿Es 12 menor o igual que 25?
```

```
True
```

```
5 == 5 # ¿Es 5 igual a 5?
```

```
True
```

```
23 != 23 # ¿Es 23 diferente de 23?
```

```
False
```

```
'aaa' == 'aaa' # Se pueden comparar otros tipos de datos
```

```
True
```

```
5 > len('5')
```

```
True
```

3.3.2 Operaciones lógicas.

Los operadores lógicos que se pueden usar son: `not`, `and` y `or`. Veamos algunos ejemplos

```
(5 < 32) and (63 > 32)
```

```
True
```

Debido a la precedencia de operadores, no son necesarios los paréntesis en la operaciones relacionales de la expresión anterior (véase tabla …):

```
5 < 32 and 63 > 32
```

```
True
```

Aunque a veces el uso de paréntesis hace la lectura del código más clara:

```
(2.32 < 21) and (23 > 63)
```

```
False
```

```
(32 == 32) or (5 < 31)
```

```
True
```

```
(32 == 21) or (31 < 5)
```

```
False
```

```
not True
```

```
False
```

```
not (32 != 32)
```

```
True
```

3.3.2.1 Comparación entre números flotantes.

La comparación entre números de tipo flotante debe reaizarse con cuidado, veamos el siguiente ejemplo:

```
(0.4 - 0.3) == 0.1
```

```
False
```

El cálculo a mano de `(0.4 -0.3)` da como resultado `0.1`; pero en una computadora este cálculo es aproximado y depende de las características del hardware (exponente, mantisa, base, véase). En Python el resultado de la operación `(0.4 -0.3)` es diferente de `0.1`, veamos:

```
print(0.4 -0.3)
```

```
0.10000000000000003
```

Python ofrece herramientas que permiten realizar una mejor comparación entre números de tipo flotante. Por ejemplo la biblioteca `math` contiene la función `isclose(a, b)` en donde se puede definir una tolerancia mínima para que las dos expresiones, `a` y `b` se consideren iguales (*cercanas*), por ejemplo:

```
import math
math.isclose((0.4 - 0.3), 0.1)
```

```
True
```

Se recomienda revisar el manual de `math.isclose()` y el de `numpy.isclose()`para comparación de arreglos con elementos de tipo flotante.

3.4 Fuertemente Tipado.

Python es fuertemente tipado, lo que significa que el tipo de un objeto no puede cambiar repentinamente; se debe realizar una conversión explícita si se desea cambiar el tipo de un objeto.

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

Veamos unos ejemplos:

```
lógico = True
real   = 220.0
entero = 284
complejo = 1+1j
cadena = 'numeros hermanos'
```

```
lógico + real # Los tipos son compatibles
```

```
221.0
```

```
lógico + complejo # Los tipos son compatibles
```

```
(2+1j)
```

```
cadena + real  # Los tipos no son compatibles
```

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

3.5 Conversión entre tipos (*casting*)

Es posible transformar un tipo en otro tipo compatible; a esta operación se lo conoce como *casting*.

3.5.1 Función `int()`

Transforma objetos en enteros, siempre y cuando haya compatibilidad.

```
cadena = '1000'
print(type(cadena))
entero = int(cadena)
print(type(entero))
print(entero)
```

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

```
flotante = 3.141592
entero  = int(flotante) # Trunca la parte decimal
print(entero)
```

```
3
```

```
complejo= 4-4j
entero = int(complejo) # Tipos NO COMPATIBLES
```

```
TypeError: int() argument must be a string, a bytes-like object or a real number, not 'comple
```

```
entero = int(True)
print(entero)
```

```
1
```

```
print(1 == True)
```

```
True
```

Función `str()`

Transforma objetos en cadenas, siempre y cuando haya compatibilidad.

```
entero = 1000
print(type(entero))
cadena = str(entero)
print(type(cadena))
print(cadena)
```

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

```
complejo = 5+1j
print(complejo)
print(type(complejo))
cadena = str(complejo)
print(cadena)
print(type(cadena))
```

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

Función `float()`

Transforma objetos en flotantes, siempre y cuando haya compatibilidad.

```
cadena = '3.141592'
print(cadena)
print(type(cadena))
real = float(cadena)
print(real)
print(type(real))
```

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

```
float(33)
```

```
33.0
```

```
float(False)
```

```
0.0
```

```
float(3+3j) # NO hay compatibilidad
```

```
TypeError: float() argument must be a string or a real number, not 'complex'
```

En general, si existe el tipo `<class 'MiClase'>`, donde `MiClase` puede ser un tipo de dato definido dentro de Python, alguna biblioteca o creada por el usuario, es posible realizar el *casting* del objeto `a` al tipo `<class 'MiClase'>` haciendo : `MiClase(a)` siempre y cuando haya compatibilidad.

3.6 Constantes

Python contiene una serie de constantes integradas a las que no se les puede cambiar su valor.

Más detalles se pueden encontrar en: Built-in Constants

Las principales constantes son las siguientes:

- `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* de arreglos.
- `__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`

```
copyright()
```

```
Copyright (c) 2001-2023 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.
```

```
license()
```

```
A. HISTORY OF THE SOFTWARE
==========================

Python was created in the early 1990s by Guido van Rossum at Stichting
Mathematisch Centrum (CWI, see https://www.cwi.nl) in the Netherlands
```

```
as a successor of a language called ABC.  Guido remains Python's
principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for
National Research Initiatives (CNRI, see https://www.cnri.reston.va.us)
in Reston, Virginia where he released several versions of the
software.

In May 2000, Guido and the Python core development team moved to
BeOpen.com to form the BeOpen PythonLabs team.  In October of the same
year, the PythonLabs team moved to Digital Creations, which became
Zope Corporation.  In 2001, the Python Software Foundation (PSF, see
https://www.python.org/psf/) was formed, a non-profit organization
created specifically to own Python-related Intellectual Property.
Zope Corporation was a sponsoring member of the PSF.

All Python releases are Open Source (see https://opensource.org for
the Open Source Definition).  Historically, most, but not all, Python


Hit Return for more, or q (and Return) to quit:  q
```

4 Cadenas.

Objetivo. Explicar el concepto de variable, etiqueta, objetos y como se usan mediante algunos ejemplos.

Funciones de Python: - `print()`, `type()`, `id()`, `chr()`, `ord()`, `del()`

4.1 Definición de cadenas

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

```
simples = 'este es un ejemplo usando \' \' '
print(simples)

dobles = "este es un ejemplo usando \" \" "
print(dobles)

triples1 = '''este es un ejemplo usando \''' \''' '''
print(triples1)

triples2 = """este es un ejemplo usando \""" \""" """
print(triples2)
```

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

Observa que para poder imprimir `'` dentro de una cadena definida con `' '` es necesario usar el caracter `\` antes de `'` para que se imprima correctamente. Lo mismo sucede en los otros ejemplos.

Es posible imprimir ' sin usar el caracter \ si la cadena se define con " y viceversa, veamos unos ejemplos:

```
# 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
```

```
# La cadena puede tener " dentro de ' ... '
titulo = 'Python "pythonico" '
print(titulo)
```

```
Python "pythonico"
```

```
# La cadena puede tener " y ' dentro de ''' ... '''
queja = """
Desde muy niño
tuve que "interrumpir" 'mi' educación
para ir a la escuela
"""
print(queja)
```

```
Desde muy niño
tuve que "interrumpir" 'mi' educación
para ir a la escuela
```

```
# La cadena puede tener " y ' dentro de """ ... """
queja = """
Desde muy niño
tuve que "interrumpir" 'mi' educación
para ir a la escuela
"""
print(queja)
```

```
Desde muy niño
tuve que "interrumpir" 'mi' educación
para ir a la escuela
```

4.2 Indexación de las cadenas.

La indexación de las cadenas permite acceder a diferentes elementos, o rangos de elementos, de una cadena.

- Todos los elementos de una cadena se numeran empezando en `0` y terminando en `N`, el cual representa el último elemento de la cadena.
- También se pueden usar índices negativos donde `-1` representa el último elemento y `-(N+1)` el primer elemento.

Veamos la siguiente tabla:

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

```
ejemplo = 'Murciélago'
```

```
ejemplo[0]
```

```
'M'
```

```
ejemplo[5]
```

```
'é'
```

```
ejemplo[9]
```

```
'o'
```

```
len(ejemplo) # Longitud total de la cadena
```

```
10
```

```
ejemplo[-1]
```

```
'o'
```

```
ejemplo[-5]
```

```
'é'
```

```
ejemplo[-10]
```

```
'M'
```

4.3 Inmutabilidad de las cadenas

Los elementos de las cadenas no se pueden modificar:

```
ejemplo[5] = "e"
```

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

```
cadena='''
esta es una
oración
larga
'''
```

```
print(type(cadena))
```

```
<class 'str'>
```

```
len(cadena)
```

```
27
```

```
cadena[0]
```

```
'\n'
```

```
cadena[-1]
```

```
'\n'
```

```
cadena[5] = 'h'
```

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

4.4 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.

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

```
'Murciélago'
```

```
ejemplo[0:5] # Elementos del 0 al 4
```

```
'Murci'
```

```
ejemplo[::2] # Todos los elementos, con saltos de 2
```

```
'Mrilg'
```

```
ejemplo[1:8:2] # Los elementos de 1 a 7, con saltos de 2
```

```
'ucéa'
```

```
ejemplo[::-1] # La cadena en reversa
```

```
'ogaléicruM'
```

4.5 Operaciones básicas con cadenas

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

```
'Luis' + ' ' + 'Miguel' # Concatenación
```

```
'Luis Miguel'
```

```
'ABC' * 3 # Repetición
```

```
'ABCABCABC'
```

4.6 Funciones aplicables sobre las cadenas

Existen métodos definidos que se pueden aplicar a las cadenas. Véase Common string operations para más información.

```
ejemplo = 'murcielago'
```

```
ejemplo.capitalize()
```

```
'Murcielago'
```

```
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
```

4.7 Construcción de cadenas con variables

```
edad = 15
nombre = 'Pedro'
apellido = 'Páramo'
peso = 70.5
```

Concatenación y casting.

```
datos = nombre + apellido + 'tiene' + str(15) + 'años y pesa ' + str(70.5)
datos
```

```
'PedroPáramotiene15años y pesa 70.5'
```

Método `format()`

```
datos = '{} {} tiene {} años y pesa {}'.format(nombre, apellido, edad, peso)
datos
```

```
'Pedro Páramo tiene 15 años y pesa 70.5'
```

Cadenas formateadas (*f-string*, *formatted string literals*)

```
datos = f'{nombre} {apellido} tiene {edad} años y pesa {peso}'
datos
```

```
'Pedro Páramo tiene 15 años y pesa 70.5'
```

5 Estructura de datos.

Objetivo. ...

Funciones de Python: ...

6 Introducción

Hay cuatro tipos de estructuras de datos, también conocidas como *colecciones*. La siguiente tabla resume estos cuatro tipos:

Tipo	Ordenada	Inmutable	Indexable	Duplicidad
List	SI	NO	SI	SI
Tuple	SI	SI	SI	SI
Sets	NO	NO	NO	NO
Dict	NO	NO	SI	NO

Cuando se selecciona un tipo de colección, es importante conocer sus propiedades para incrementar la eficiencia y/o la seguridad de los datos.

7 Listas

- Consisten en una secuencia **ordenada** y **mutable** de elementos.
  - Ordenadas significa que cada elemento dentro de la lista está indexado y mantiene su orden definido en su creación.
  - Mutable significa que los elementos de la lista se pueden modificar, y además que se pueden agregar o eliminar elementos.
- Las listas pueden tener elementos **duplicados**, es decir, **elementos del mismo tipo y con el mismo contenido**.

7.1 Ejemplo 1.

Creamos 4 listas:

- `gatos` : Razas de gatos.
- `origen` : Origen de cada raza de gatos.
- `pelo_largo`: Si tienen pelo largo o no.
- `pelo_corto`: Si tienen pelo corto o no.
- `peso_minimo`: El peso mínimo que pueden tener.
- `peso_maximo`: El peso máximo que pueden tener.

```
# Las lista se definen usando corchetes []
gatos = ['Persa', 'Sphynx', 'Ragdoll','Siamés']
origen = ['Irán', 'Toronto', 'California', 'Tailandia']
pelo_largo = [True, False, True, True]
pelo_corto = [False, False, False, True]
peso_minimo = [2.3, 3.5, 5.4, 2.5]
peso_maximo = [6.8, 7.0, 9.1, 4.5]
```

Observaciones: * Cada lista contiene 4 elementos. * Los elementos de cada lista son del mismo tipo. * Los elementos son cadenas, tipos lógicos y flotantes.

Se puede obtaner el tipo de las listas como sigue:

```
print(type(gatos))
```

```
print(gatos)
```

7.2 Indexado

Se puede acceder a cada elemento de las listas de manera similar a como se hace con las cadenas, veáse la notebook ...

Por ejemplo:

```
gatos[0] # Primer elemento
```

```
gatos[1:4] # Todos los elementos, desde el 1 hasta el 3
```

```
gatos[-1] # Último elemento
```

```
gatos[::-1] # Todos los elementos en reversa
```

Para conocer el tipo de objeto de uno de los elementos podemos hacer lo siguiente:

```
print(type(gatos[0]))
```

```
print(type(peso_maximo[2]))
```

7.3 Operaciones sobre las listas

Existen muchas operaciones que se pueden realizar sobre las listas. A continuación se muestran unos ejemplos

```
len(gatos) # Determinar la longitud de la lista
```

```
max(gatos) # Determinar el máximo elemento de la lista
```

```
min(gatos) # Determinar el mínimo elemento de la lista
```

```
# Operación lógica elemento a elemento.
# Produce una lista con elementos lógicos.
sin_pelo = pelo_largo or pelo_corto
print(sin_pelo)
```

```
gatos + peso_maximo # Concatenación de dos listas
```

```
origen * 2 # Duplicación de la lista, intenta multiplicar por 3
```

```
'Siamés' in gatos # ¿Está el elemento `Siamés` en la lista gatos?
```

7.4 Métodos de las listas (comportamiento)

En términos de Programación Orientada a Objetos, la clase `<class 'list'>` define una serie de métodos que se pueden aplicar sobre los objetos del tipo `list`. Veamos algunos ejemplos:

```
print(gatos) # Imprimimos la lista original
```

```
gatos.append('Siberiano') # Se agrega un elemento al final de la lista
```

```
print(gatos)
```

```
gatos.append('Persa') # Se agrega otro elemento al final de la lista, repetido
```

```
print(gatos)
```

```
gatos.remove('Persa') # Eliminamos el elemento 'Persa' de la lista
```

```
print(gatos)
```

Observa que solo se elimina el primer elemento 'Persa' que encuentra.

```
gatos.insert(0,'Persa') # Podemos insertar un elemento en un lugar específico de la lista
```

```
print(gatos)
```

```
gatos.pop() # Extrae el último elemento de la lista
```

```
print(gatos)
```

```
gatos.sort() # Ordena la lista
```

```
print(gatos)
```

```
gatos.reverse() # Modifca la lista con los elementos en reversa
```

```
print(gatos)
```

Una descripción detallada de los métodos de la listas se puede ver en More on Lists.

7.5 Copiando listas

Una lista es un objeto que contiene varios elementos. Para crear una copia de una lista, se debe generar un espacio de memoria en donde se copien todos los elementos de la lista original y asignar un nuevo nombre para esta nueva lista. Lo anterior no se puede hacer simplemente con el operador de asignación. Veamos un ejemplo:

```
gatitos = gatos
```

```
print(gatos)
print(gatitos)
```

Podemos observar que al imprimir la lista mediante los nombres `gatos` y `gatitos` obtenemos el mismo resultado. Ahora, modifiquemos el primer elemento usando el nombre `gatitos`:

```
gatitos[0] = 'Singapur'
```

```
print(gatos)
print(gatitos)
```

Observamos que al imprimir la lista usando `gatos` y `gatitos` volvemos a obtener el mismo resultado. Lo anterior significa que el operador de asignación solamente creó un nuevo nombre para el mismo objeto en memoria, por lo que en realidad no hizo una copia de la lista. Lo anterior lo podemos verificar usando la función `id()` para ver la dirección en memoria del objeto:

```
print(id(gatitos))
print(id(gatos))
```

7.5.1 Copiando con `[:]`

Crear una nueva lista copiando todos los elementos podemos hacer lo siguiente:

```
gatitos = gatos[:]
```

```
print(type(gatitos))
print(id(gatitos))
print(gatitos)

print(type(gatos))
print(id(gatos))
print(gatos)
```

Observa que el identificador en memoria de cada lista es diferente.

7.5.2 Copiando con el método `copy()`

La clase `<class 'list'>` contiene un método llamado `copy()` que efectivamente realiza una copia de la lista:

```
gatitos = gatos.copy()
```

```
print(type(gatitos))
print(id(gatitos))
print(gatitos)

print(type(gatos))
print(id(gatos))
print(gatos)
```

Observa que el identificador en memoria de cada lista es diferente.

7.5.3 Copiando con el constructor

La función `list()` transforma un objeto *iterable* en una lista. La podemos usar para copiar una lista como sigue:

```
gatitos = list(gatos)
```

```
print(type(gatitos))
print(id(gatitos))
print(gatitos)

print(type(gatos))
print(id(gatos))
print(gatos)
```

Observa que el identificador en memoria de cada lista es diferente.

NOTA. Lo que sucede en este último caso, es que se ejecuta el constructor de la clase `<class 'list'>`, el cual recibe un objeto iterable (lista, tupla, diccionario, entre otros), copia todos los elementos de ese iterable y los pone en una lista que se almacena en un espacio en memoria diferente al iterable original.

7.5.4 Copiando con la biblioteca `copy`

```
import copy
gatitos = copy.copy(gatos)
```

```
print(type(gatitos))
print(id(gatitos))
print(gatitos)

print(type(gatos))
print(id(gatos))
print(gatos)
```

Observa que el identificador en memoria de cada lista es diferente.

Más información sobre el uso de esta biblioteca se puede ver en Shallow and deep copy operations.

7.6 Listas mas complejas.

Las listas pueden tener elementos de distintos tipos. Por ejemplo:

```
superlista = ['México', 3.141592, 20, 1j, [1,2,3,'lista']]
```

```
superlista
```

```
superlista[0] # El elemento 0 de la lista
```

```
superlista[4] # El elemento 4 de la lista (este elemento es otra lista)
```

```
superlista[4][2] # El elemento 2 del elemento 4 de la lista original
```

8 Tuplas

- Consisten en una secuencia **ordenada** e **inmutable** de elementos.
  - Ordenadas significa que cada elemento dentro de la tupla está indexado y mantiene su orden definido en su creación.
  - Inmutable significa que los elementos de la tupla **NO se pueden modificar**, tampoco que se pueden agregar o eliminar elementos.
- Las tuplas pueden tener elementos **duplicados**, es decir, **elementos del mismo tipo y con el mismo contenido**.

Veamos algunos ejemplos:

```
# Las tuplas se definen usando paréntesis ()
tupla1 = () # tupla vacía

print(type(tupla1))
print(id(tupla1))
print(tupla1)
```

La clase `<class 'tuple'>` solo contiene dos funciones: * `index(o)`, determina el índice dentro de la tupla del objeto `o`. * `count(o)`, determina el número de objetos iguales a `o` existen dentro de la tupla.

```
tupla = ('a', 'b', 'c', 'b', 'd', 'e', 'f', 'b')
print(tupla)
```

```
tupla.index('a')
```

```
tupla.count('b')
```

Si deseamos una tupla de un solo elemento debemos realizar lo siguiente:

```
tupla_1 = (1,)
print(type(tupla_1))
print(tupla_1)
```

La siguiente expresión no construye una tupla, si no un entero:

```
tupla_1 = (1)
print(type(tupla_1))
print(tupla_1)
```

8.1 Indexado.

El indexado de las tuplas es similar al de las listas.

```
print(tupla)
```

```
tupla[0]
```

```
tupla[-1]
```

```
tupla[2:5]
```

```
tupla[::-1]
```

8.2 Inmutabilidad

Los elementos de las tuplas no se pueden modificar.

```
tupla[2]
```

```
tupla[2] = 'h'
```

8.3 ¿Copiando tuplas?

No es posible crear una copia de una tupla en otra. Lo que se recomienda es transformar la tupla en otra estructura de datos compatible (por ejemplo `list` o `set`).

9 Conjuntos

- Consisten en una secuencia **NO ordenada**, **modificable**, **NO indexable** y **NO** permite miembros duplicados.

Veamos algunos ejemplos:

```
# Los conjuntos se definen con {}
conjunto = {4,1,8,0,4,20}
```

```
print(type(conjunto))
print(id(conjunto))
print(conjunto)
```

9.1 Funciones y operaciones sobre conjuntos

```
# Adiccionar un elemento
conjunto.add(-8)
print(conjunto)
```

```
# Eliminar un elemento del conjunto (el elemento debe existir dentro del conjunto)
conjunto.remove(4)
print(conjunto)
```

```
# ¿El elemento 0 está en el conjunto?
0 in conjunto
```

```
# ¿El elemento 10 está en el conjunto?
10 in conjunto
```

```
# Limpiar todos los elementos del conjunto
conjunto.clear()
print(type(conjunto))
print(id(conjunto))
print(conjunto)
```

Observa que el identificador no ha cambiado, solo se eliminaron todos los elementos.

```
# Definimos dos conjuntos
A = {'Taza', 'Vaso', 'Mesa'}
B = {'Casa', 'Mesa', 'Silla'}
```

```
print(A)
print(B)
```

```
A - B # elementos en A, pero no en B
```

```
A | B # elementos en A o en B o en ambos
```

```
A & B # elementos en ambos conjuntos
```

```
A ^ B # elementos en A o en B, pero no en ambos
```

9.2 Copiando conjuntos

```
conjunto = {4,1,8,0,4,20}
```

9.2.1 Copiando con el método `copy()`

```
# Crear otro conjunto haciendo una copia
conjunto2 = conjunto.copy()

print(type(conjunto2))
print(id(conjunto2))
print(conjunto2)
```

```
print(type(conjunto))
print(id(conjunto))
print(conjunto)
```

9.2.2 Copiando con la biblioteca `copy()`

```
import copy
conjunto2 = conjunto.copy()

print(type(conjunto2))
print(id(conjunto2))
print(conjunto2)

print(type(conjunto))
print(id(conjunto))
print(conjunto)
```

10 Diccionarios

- Diccionarios son colecciones que **NO** son ordenadas, son **modificables**, **indexables** y **NO** permiten miembros duplicados.
- Las colecciones están compuesta por pares `clave:valor`.
- Se accede a los valores mediante las claves en lugar de índices.

Veamos algunos ejemplos:

```
dicc = {'Luis': 20, 'Miguel': 25}
```

```
print(type(dicc))
print(dicc)
```

10.1 Operaciones sobre diccionarios

```
dicc['Luis'] # acceder a un elemento del diccionario
```

Se puede acceder a las claves y a los valores de manera independiente como sigue:

```
dicc.keys() # Obtener todas las claves
```

```
dicc.values() # Obtener todos los valores
```

```
# ¿Existe el elemento 'Miguel' en el diccionario?
'Miguel' in dicc
```

```
# ¿Existe el valor 25 en los valores del diccionario?
25 in dicc.values()
```

```
# ¿Existe la clave 'Luis' en las claves del diccionario?
'Luis' in dicc.keys()
```

```
len(dicc) # Calcular la longitud (el número de parejas)
```

```
dicc['fulano'] = 100 # Agregar el par `'fulano':100`
```

```
print(dicc)
```

Observa que cuando el elemento no existe, la expresión `dicc['fulano'] = 100` agrega el par `'fulano': 100` al diccionario.

```
del dicc['Miguel'] # Eliminar el par `'Miguel':25`
```

```
print(dicc)
```

Podemos agregar un diccionario en otro:

```
dicc_otro = {'nuevo':'estrellas', 'viejo':'cosmos', 'edad':15000000}
```

```
print(dicc_otro)
```

```
dicc.update(dicc_otro) # Agregamos el diccionario dicc_otro al diccionario dicc
```

```
print(dicc)
```

Si los elementos ya existen, solo se actualizan los valores:

```
nuevo = {'Luis':512, 'viejo':2.1}
```

```
dicc.update(nuevo)
```

```
print(dicc)
```

10.2 Copiando diccionarios

```
print(dicc)
```

10.2.1 Copiando con el método `copy()`

```
dicc_cp = dicc.copy()

print(type(dicc_cp))
print(id(dicc_cp))
print(dicc_cp)

print(type(dicc))
print(id(dicc))
print(dicc)
```

10.2.2 Copiando con la biblioteca `copy()`

```
import copy
dicc_cp = dicc.copy()

print(type(dicc_cp))
print(id(dicc_cp))
print(dicc_cp)

print(type(dicc))
print(id(dicc))
print(dicc)
```

11 Transformación entre colecciones

11.0.1 listas → tuplas

```
lista = ['a', 'b', 'c']
```

```
tupla_l = tuple(lista)

print(type(tupla_l))
print(id(tupla_l))
print(tupla_l)

print(type(lista))
print(id(lista))
print(lista)
```

```
<class 'tuple'>
2215532231808
('a', 'b', 'c')
<class 'list'>
2215532334976
['a', 'b', 'c']
```

11.0.2 listas → conjuntos

```
lista = ['a', 'b', 'c', 'a']
```

```
conj_l = set(lista)

print(type(conj_l))
print(id(conj_l))
print(conj_l)
```

```
print(type(lista))
print(id(lista))
print(lista)
```

```
<class 'set'>
2215531882848
{'c', 'a', 'b'}
<class 'list'>
2215532336832
['a', 'b', 'c', 'a']
```

Observa que en esta transformación el conjunto elimina los elementos repetidos.

11.0.3 listas → diccionarios

```
key_l = ['a','b','c','d', 'e']
val_l = [1, 2, 3, 4, 5]
```

```
dicc = dict(zip(key_l, val_l))

print(key_l)
print(val_l)

print(type(dicc))
print(id(dicc))
print(dicc)
```

```
['a', 'b', 'c', 'd', 'e']
[1, 2, 3, 4, 5]
<class 'dict'>
2215509691520
{'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
```

11.0.4 tuplas → listas

```
tupla = (1,2,3,4)
```

```
lista_t = list(tupla)

print(type(lista_t))
print(id(lista_t))
print(lista_t)

print(type(tupla))
print(id(tupla))
print(tupla)
```

```
<class 'list'>
2215537704832
[1, 2, 3, 4]
<class 'tuple'>
2215537784000
(1, 2, 3, 4)
```

11.0.5 tuplas → conjuntos

```
tupla = (1,2,3,1,2)
```

```
conj_t = set(tupla)

print(type(conj_t))
print(id(conj_t))
print(conj_t)

print(type(tupla))
print(id(tupla))
print(tupla)
```

```
<class 'set'>
2215537722528
{1, 2, 3}
<class 'tuple'>
2215537786320
(1, 2, 3, 1, 2)
```

11.0.6 tuplas → diccionarios

```
key_t = ('a','b','c','d', 'e')
val_t = (1, 2, 3, 4, 5)
```

```
dicc = dict(zip(key_t, val_t))

print(key_t)
print(val_t)

print(type(dicc))
print(id(dicc))
print(dicc)
```

```
('a', 'b', 'c', 'd', 'e')
(1, 2, 3, 4, 5)
<class 'dict'>
2215537751808
{'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
```

11.0.7 conjunto → lista y tupla

```
conj = {1,2,3,'a','b','c'}
```

```
lista_s = list(conj)
tupla_s = tuple(conj)
```

```
print(type(conj))
print(id(conj))
print(conj)

print(type(lista_s))
print(id(lista_s))
print(lista_s)

print(type(tupla_s))
print(id(tupla_s))
print(tupla_s)
```

```
<class 'set'>
2215537723424
{1, 2, 3, 'b', 'c', 'a'}
<class 'list'>
2215537788864
[1, 2, 3, 'b', 'c', 'a']
<class 'tuple'>
2215534718752
(1, 2, 3, 'b', 'c', 'a')
```

11.0.8 conjuntos → diccionarios

```
conj1 = {'a','b','c'}
conj2 = {1,2,3}

dicc = dict(zip(conj1, conj2))

print(conj1)
print(conj2)

print(type(dicc))
print(id(dicc))
print(dicc)
```

```
{'c', 'a', 'b'}
{1, 2, 3}
<class 'dict'>
2215537993856
{'c': 1, 'a': 2, 'b': 3}
```

11.0.9 diccionarios → listas, tuplas y conjuntos

```
dicc = {'x':3, 'y':4, 'z':5}
```

Conversión directa:

```
lista = list(dicc)
tupla = tuple(dicc)
conj = set(dicc)
```

```
print(dicc)

print(type(lista))
print(lista)
print(type(tupla))
print(tupla)
print(type(conj))
print(conj)
```

```
{'x': 3, 'y': 4, 'z': 5}
<class 'list'>
['x', 'y', 'z']
<class 'tuple'>
('x', 'y', 'z')
<class 'set'>
{'y', 'x', 'z'}
```

Conversión desde las claves:

```
lista = list(dicc.keys())
tupla = tuple(dicc.keys())
conj = set(dicc.keys())
```

```
print(dicc)

print(type(lista))
print(lista)
print(type(tupla))
print(tupla)
print(type(conj))
print(conj)
```

```
{'x': 3, 'y': 4, 'z': 5}
<class 'list'>
['x', 'y', 'z']
<class 'tuple'>
('x', 'y', 'z')
<class 'set'>
{'y', 'x', 'z'}
```

Conversión desde las valores:

```
lista = list(dicc.values())
tupla = tuple(dicc.values())
conj = set(dicc.values())
```

```
print(dicc)

print(type(lista))
print(lista)
print(type(tupla))
print(tupla)
print(type(conj))
print(conj)
```

```
{'x': 3, 'y': 4, 'z': 5}
<class 'list'>
[3, 4, 5]
<class 'tuple'>
(3, 4, 5)
<class 'set'>
{3, 4, 5}
```

12 Control de flujo.

Objetivo. ...

Funciones de Python: ...

En Python existen declaraciones que permiten controlar el flujo de un programa para realizar acciones complejas. Entre estas declaraciones tenemos las siguientes:

- `while`
- `for`
- `if`
- `match`

Junto con estas declaraciones generalmente se utilizan las siguientes operaciones lógicas cuyo resultado puede ser `True` o `False`:

Python	Significado
`a == b`	¿son iguales `a` y `b`?
`a != b`	¿son diferentes `a` y `b`?
`a < b`	¿`a` es menor que `b`?:
`a <= b`	¿`a` es menor o igual que `b`?
`a > b`	¿`a` es mayor que `b`?
`a >= b`	¿`a` es mayor o igual que `b`?
`not A`	El inverso de la expresión `A`
`A and B`	¿La expresión `A` y la expresión `B` son verdaderas?
`A or B`	¿La expresión `A` o la expresión `B` es verdadera?:

13 `while`

Se utiliza para repetir un conjunto de instrucciones mientras una expresión sea verdadera:

```
while expresión:
    código ...
```

Por ejemplo:

```
a = 0 # Inicializamos a en 0

print('Inicia while') # Instrucción fuera del bloque while

while a < 5: # Mientras a sea menor que 5 realiza lo siguiente:
    print(a) # Imprime el valor de a
    a += 1   # Incrementa el valor de a en 1

print('Finaliza while')  # Instrucción fuera del bloque while
```

- Como se observa, el código después de `while` tiene una sangría (*indentation*): las líneas de código están recorridas hacia la derecha.
- Este espacio en blanco debe ser al menos de uno, pero pueden ser más.
- Por omisión, en JupyterLab (y algunos otros editores, se usan 4 espacios en blanco para cada línea de código dentro del bloque.
- El número de espacios en blanco se debe mantener durante todo el bloque de código.
- Cuando termina el sangrado, es decir las líneas de código ya no tienen ningún espacio en blanco al inicio, se cierra el bloque de código, en este caso el `while`.
- El uso de una sangría para organizar los bloques de código lo hace Python para que el código sea más entendible.
- **Ejemplos válidos**: