Python 3

> type()

Todo es un objeto

"One of my goals for Python was to make it so that all objects were “first class.” By this, I meant that I wanted all objects that could be named in the language (e.g., integers, strings, functions, classes, modules, methods, etc.) to have equal status. That is, they can be assigned to variables, placed in lists, stored in dictionaries, passed as arguments, and so forth."

\- Guido van Rossum

Built-in types

- Numéricos: int, float, complex
- Secuencias: list, tuple, range
- Secuencias de texto: str
- Secuencias binarias: bytes, bytearray, memoryview
- Conjuntos: set, frozenset
- Mapeadores: dict
- Booleanos: bool
- Tipo nulo: None

id() & type()

La función id() devuelve la identidad de un objeto. Esto es un entero que es único durante su tiempo de vida. En CPython este entero corresponde a su dirección en memoria.

```
>>> id(1)
9079008
>>> a = 2
>>> id(a)
9079040
>>> id(None)
8537744
```

La función type() devuelve el tipo de un objeto.

```
>>> type(1)
<class 'int'>
>>> type(int)
<class 'type'>
>>> type(len)
<class 'builtin function or method'>
```

No se recomienda utilizar esta función para testear el tipo de un objeto. En estos casos es mejor utilizar isinstance() que tiene en cuenta subclases.

Mutabilidad (I)

Los objetos **inmutables** no pueden ser modificados luego de su creación. Algunos tipos inmutables son: `int`, `float`, `string`, `tuple`, `frozenset` y `bytes`.

```
>>> b = 'id'
>>> a = 'hi'
>>> a[0] = 'v'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> b = a
>>> b += ' friend'
>>> b
'hi friend'
>>> a
'hi'
```

Los objetos **mutables** si pueden ser modificados luego de su creación. Algunos ejemplos son: `list`, `dict`, `set` y `bytearray`.

```
>>> a = [1]
>>> a[0] = 0
>>> a
[0]
>>> b = a
>>> b.append(1)
>>> b
[0, 1]
>>> a
[0, 1]
```

Mutabilidad (II)

- Los objetos inmutables son más rápidos de acceder.
- Es computacionalmente costoso cambiar “cambiar” objetos inmutables.
- Los objetos mutables son mejores cuando se desean cambiar valores del objeto (extender su tamaño, cambiar valores, etc.).
- Los objetos inmutables tienen la garantía de que no pueden ser cambiados.

Mutabilidad (III)

Cuando se “cambia” un valor **inmutable** se genera una nueva copia del objeto con la nueva información.

```
>>> a = 'Hello'
>>> b = a
>>> a += ' World!'
>>> b
'Hello'
>>> l1 = [0, 1, 2]
>>> l2 = l1
>>> l1[0] = ':D'
>>> l1
[':D', 1, 2]
>>> l2  # lists are mutable
[':D', 1, 2]
```

Hay que tener cuidado cuando se opera con contenedores como tuple que tienen **mutables** dentro.

```
>>> a = [0, 1, 2]
>>> t1 = ('a', a)
>>> t1
('a', [0, 1, 2])
>>> t2 = (t1, t1)
>>> t2
(('a', [0, 1, 2]), ('a', [0, 1, 2]))
>>> t1 = ('b', [])
>>> t1
('b', [])
>>> t2
(('a', [0, 1, 2]), ('a', [0, 1, 2]))
>>> a[0] = 99
>>> t2
(('a', [99, 1, 2]), ('a', [99, 1, 2]))
```

Objetos hasheables (I)

Un objeto es **hasheable** si tiene un hash que nunca cambia durante su tiempo de vida (implementando `__hash__()`) y puede ser comparado con otros objetos (implementando `__eq__()`). Los objetos que se comparen como iguales deben tener el mismo hash.

La mayoría de los **inmutables** built-in son **hasheables**. Los contenedores **mutables** (como `list` y `dict`) no lo son. Los contenedores **inmutables** (como `tuple`) son **hasheables** mientras sus componentes lo sean.

Objetos hasheables (II)

```
>>> class A(object):
...     pass
...
>>> a1 = A()
>>> a2 = A()
>>> a1 == a2
False
>>> hash(a1) == hash(a2)
False
```

Por default las instancias de las clases definidas por el usuario son **hasheables**, todas las instancias se comparan como desiguales entre sí y su hash es derivado de su `id()`.

== vs is

```
>>> 'Example' == 'Example'
True
>>> 'Example' is 'Example'
True
>>> [0, 1, 2] == [0, 1, 2]
True
>>> [0, 1, 2] is [0, 1, 2]
False
>>> a = 'Example'
>>> b = a
>>> b is a
True
>>> a += 's'
>>> b is a
False
>>> l1 = [0, 1, 2]
>>> l2 = l1
>>> l2 is l1
True
>>> l1[0] = ':D'
>>> l2 is l1
True
>>> l1
[':D', 1, 2]
>>> l2
[':D', 1, 2]
```

El operador `==` compara dos objetos y devuelve `True` si son considerados iguales o `False` si no lo son.

En cambio el operador `is` compara sus dos operandos y devuelve `True` si son el mismo objeto o `False` si no lo son.

Common Data Types

bool

El tipo bool solo tiene como instancias dos constantes: True y False.

Los operadores <, <=, ==, !=, >, >=, not, is e is not siempre devuelven un resultado del tipo bool.

Los operadores or y and evalúan sus operandos de forma booleana pero pueden devolver un resultado de otro tipo.

x or y devuelve y si x es falso, sino x.
x and y devuelve x si x es falso, sino y

int & float

Los valores int no tienen límite de tamaño.

Los valores float son representados internamente como double precision floating point numbers. Los detalles de esta implementación varía con la arquitectura de la máquina y el intérprete de Python.

Los valores float son almacenados utilizando fracciones binarias y por lo tanto muchas fracciones decimales no pueden ser representadas de manera exacta con este tipo.

```
>>> 1 / 2  # always float
0.5
>>> 1 // 2  # always int
0
>>> -1 // 2  # be careful with negative values
-1
>>> int('-1')  # str to int
-1
>>> float('1.54')  # str to float
1.54
>>> .1 + .1 + .1 == .3  # what the f*ck
False
>>> threshold = .00001  # let's fix that last comparison
>>> abs((.1 + .1 + .1) - .3) < threshold  # abs() returns the absolute value of a number
True
```

Algunos ejemplos con datos numéricos

Tuplas (I)

El tipo tuple es un tipo **inmutable** que representa secuencias de elementos (normalmente heterogeneos).

```
>>> type((0, 'a'))  # a simple tuple
<class 'tuple'>
>>> type(())  # use () to create an empty tuple
<class 'tuple'>
>>> type((1))  # this is not a tuple
<class 'int'>
>>> type((1,))  # but this is
<class 'tuple'>
>>> tuple(range(10))  # building a tuple from an iterable
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> a = (0, 'a')  # individual elements can be accessed but not changed
>>> a[1]
'a'
>>> a[1] = 'b'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Tuplas (II)

Python tiene una funcionalidad que permite asignarle a una tupla de variables los valores de una tupla.

```
>>> point = (-5, 3, 2)  # a tuple that represents a point in a 3D space
>>> (x, y, z) = point  # unpack the point into three variables
>>> x
-5
>>> y
3
>>> z
2
>>> (y, x) = (x, y)  # tuples can be used to swap the values of two variables
>>> x
3
>>> y
-5
```

Listas

El tipo `list` es un tipo **mutable** que representa secuencias de elementos (normalmente homogeneos).

```
>>> []  # an empty list
[]
>>> a = [1, 2, 3]  # define a list using extension
>>> b = [x ** 2 for x in range(10) if x % 2]  # define a list using comprehension
>>> b
[1, 9, 25, 49, 81]
>>> b[1]
9
>>> b[1] = ':D'
>>> b
[1, ':D', 25, 49, 81]
>>> list(range(10))  # build a list from an iterable
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Strings

El tipo string es un tipo de dato **inmutable**. Es una secuencia de caracteres encodeados utilizando Unicode.

```
>>> 'a "laser"'
'a "laser"'
>>> "a 'laser'"
"a 'laser'"
>>> '''Triple single or double quotes
...  for strings that span multiple lines'''
'Triple single or double quotes\n for strings that span multiple lines'
>>> ('hello' ' world')  # strings separated by whitespace will be joined into one
'hello world'
```

bytes & bytearray

Los tipos bytes y bytearray son secuencias de bytes. Sus literales y representaciones están basadas en texto ASCII pero operan como secuencias de enteros (con valores entre 0 y 255). bytes es **inmutable** y bytearray **mutable**.

```
>>> b'abcd'  # a bytes literal
b'abcd'
>>> b'abcd\xE1'  # escape sequences are needed for values over 127
b'abcd\xe1'
>>> bytes(10)  # a 0 filled bytes object of length 10
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
>>> bytes(range(10))  # build a bytes object from an iterable
b'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t'
>>> bytearray()  # create an empty bytearray instance
bytearray(b'')
>>> bytearray(10)  # a 0 filled bytearray object of length 10
bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00')
>>> bytearray(range(10))  # build a bytearray object from an iterable
bytearray(b'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t')
```

set & frozenset

Los sets son colecciones no ordenadas de elementos **hasheables** únicos. Comúnmente se utilizan para testear pertenencia, remover duplicados y realizar operaciones entre conjuntos. set es **mutable** y frozenset es **inmutable**.

```
>>> {1, 2, 1}  # create a set (duplicate will be removed)
{1, 2}
>>> a = set(range(9))  # create a set from an iterable
>>> a
{0, 1, 2, 3, 4, 5, 6, 7, 8}
>>> a[0]  # sets are collections, not sequences
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object is not subscriptable
>>> a.add(9)
>>> a
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> frozenset(['a', 'b', 'c'])  # create a frozenset instance
frozenset({'b', 'c', 'a'})
```

Diccionarios

El tipo dict mapea objetos **hasheables** a cualquier tipo de objeto. Es un tipo **mutable.**

```
>>> {}  # create an empty dict
{}
>>> a = {'a': 1, 'b': 2}  # create a dictionary that contains two elements
>>> a['a'] = 0  # dicts are mutable
>>> a['c'] = 3
>>> a
{'a': 0, 'b': 2, 'c': 3}
>>> a['d']  # a KeyError is raised if we try to access a key that is not in the map
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'd'
>>> list(a)  # dicts iterate over their keys
['a', 'b', 'c']
```

code()

Ejercicios 2 (I)

- Escribir un programa que dados dos números m y n imprima:

  ```
  <m> + <n> = <result>
  <m> - <n> = <result>
  <m> * <n> = <result>
  <m> / <n> = <result>
  ```

  En caso de que n sea 0 indicar con un mensaje que la división no se puede realizar.
- Escribir un programa que dadas las coordenadas (x, y) de 3 puntos determine si estos están alineados.

Ejercicios 2 (II)

- Escribir un programa que dados dos números m y n genere una lista de los números naturales hasta m * n que no son divisibles por 3.
- En una biblioteca se quiere almacenar información sobre sus libros. Cada libro está identificado con un número de ID y tiene la siguiente información: título, autor, año y número de copias. Escribir un programa que permita cargar la información de estos libros y permita consultarla a través de su ID. Se asume que el ID de consulta es válido siempre.

Bibliografía

- Python docs
- How to Think Like a Computer Scientist: Learning with Python
- Mutable vs Immutable Objects in Python