

VERSIÓN PRELIMINAR

Python en Ámbitos Científicos

Facundo Batista & Manuel Carlevaro

7 de noviembre de 2024

VERSIÓN PRELIMINAR

Título: Python en Ámbitos Científicos
Autores: Facundo Batista & Manuel Carlevaro
ISBN-13 (versión electrónica): ???-?-???-???-?
© Facundo Batista & Manuel Carlevaro
Primera Edición (versión preliminar)
Escrito con X₃TEX.

Licencia: [Creative Commons Atribución-NoComercial-CompartirIgual 4.0 Internacional \(CC BY-NC-SA 4.0\)](https://creativecommons.org/licenses/by-nc-sa/4.0/)
Lugar: Olivos y La Plata, Buenos Aires, Argentina
Año: 2024
Web: <http://pyciencia.taniquetil.com.ar/>

10 9 8 7 6 5 4 3 2 1

Prefacio

Cuando la Comunidad Científica notó que las computadoras podían facilitar su trabajo, sus integrantes se convirtieron posiblemente en los adoptantes tempranos más entusiastas de la tecnología. Aunque el uso y la potencia de las computadoras crece continuamente en el ámbito científico, poco se ha avanzado en la formación y desarrollo de habilidades en Desarrollo de Software.

El cálculo científico requiere realizar combinaciones de múltiples tareas de diversa clase. Por ejemplo, es necesario registrar automáticamente datos de un experimento y visualizarlos, realizar cálculos numéricos o simbólicos, ordenar, clasificar, simular, etc. Muchas veces podemos utilizar paquetes de software que realizan esas tareas por nosotros, pero muchas otras ocurre que nadie ha implementado un determinado cómputo en la forma que necesitamos, o simplemente queremos probar ideas nuevas. Cualquiera sea la naturaleza de nuestra actividad en la ciencia o en la tecnología, no resulta infrecuente la necesidad de interactuar con computadoras a través de programas propios.

El uso de Python en aplicaciones científicas ha aumentado sostenidamente en los últimos años, sin embargo es difícil encontrar libros o manuales en castellano de Python que no estén pensados para programadores.

Este libro nace con la idea primaria de acercar Python al mundo científico, en un libro pensado para científicos, a partir de nuestra experiencia en el dictado del curso "Herramientas Computacionales para Científicos" que ofrecemos en la Universidad Nacional de La Plata y la Universidad Tecnológica Nacional, desde 2007. De la misma manera, la elección del castellano como idioma de escritura es un factor crítico, porque aunque sabemos que el inglés es una herramienta fundamental tanto para programar como para hacer ciencia, estamos convencidos que no debería ser una barrera de entrada.

Más allá de las secciones básicas de un libro (índice, bibliografía, etc.), el libro tiene tres grandes partes. La primera habla de Python, tanto de forma introductoria como también sobre otros temas que son fundamentales y algunas bibliotecas importantes. La segunda trata algunas herramientas fundamentales que son base para el trabajo en el resto del libro. Finalmente la tercera parte muestra cómo abordar temas científicos básicos utilizando Python, de forma teórica y práctica.

En todos los casos este libro esquivó la pretensión de ser una referencia absoluta, sino que tiene el propósito de allanar el camino de los científicos para dar los primeros pasos en el lenguaje y solucionar los problemas básicos (pero no por eso menos importantes) de la ciencia y la tecnología.

Tanto los textos como el código fuente, ejemplos e imágenes son Copyright de Facundo Batista y Manuel Carlevaro y están compartidos bajo la licencia Creative Commons Atribución-NoComercial-CompartirIgual 4.0 Internacional (CC BY-NC-SA 4.0) [1], salvo que se especifique puntualmente lo contrario.

Olivos y La Plata, Buenos Aires, Argentina,

Facundo Batista & Manuel Carlevaro

Índice general

Prefacio	2
Índice general	3
I Python	4
1. Tipos de datos	5
1.1. Números	5
1.2. Cadenas	10
1.3. Listas	18
1.4. Tuplas	22
1.5. Pensando como un pythonista	26
1.6. Conjuntos	33
1.7. Diccionarios	37
1.8. Iteradores	40
II Herramientas fundamentales	44
III Temas específicos	45
IV Apéndices	46
A. Zen de Python	47
Bibliografía	48

Parte I

Python

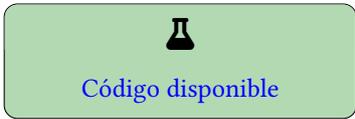
Esta primera parte comprende varios capítulos orientados a proveer la información necesaria de Python para poder entender el resto del libro.

Se sugiere a los lectores que no tienen experiencia previa en el uso de Python, un recorrido secuencial a través de los capítulos que componen esta Parte.

1 | Tipos de datos

En este capítulo mostraremos los tipos de datos más utilizados en Python.

No incluiremos *todos* los tipos de datos integrados en el lenguaje (ni mucho menos todos los presentes en la Biblioteca Estándar), sólo haremos foco en aquellos más utilizados y que les permitirá comenzar y realizar gran parte de todo lo que harán con Python.



1.1. Números

Comenzamos con el tipo de datos más básico en todos los lenguajes: los números enteros, cuyo tipo de datos es `int`.

Las operaciones básicas son las mismas que en cualquier otro lenguaje de programación y no tenemos ninguna sorpresa. Los únicos detalles a destacar son que la división de números enteros da como resultado un `float` (el punto flotante binario, que veremos a continuación), y que el operador para la exponenciación es `**`.

```
CELL 01
3 + 5
-----
8
```

```
CELL 02
2 - 8
-----
-6
```

```
CELL 03
4 * -8
-----
-32
```

CELL 04
<code>32 / 5</code>

<code>6.4</code>

CELL 05
<code>5 ** 2</code>

<code>25</code>

Complementando la división, Python incluye un operador para la división con el resultado truncado (celda 6), el módulo (celda 7), e incluso una función integrada que da ambos resultados al mismo tiempo (celda 8):

CELL 06
<code>32 // 5</code>

<code>6</code>

CELL 07
<code>32 % 5</code>

<code>2</code>

CELL 08
<code>divmod(32, 5)</code>

<code>(6, 2)</code>

Más allá de estas operaciones particulares, el rasgo más relevante a destacar en los enteros de Python es que no tienen límite (porque no están atados a ninguna representación en particular), entonces no hace falta que nos preocupemos por ese detalle:

CELL 09
<code>2529 ** 15 * 1834</code>

<code>2030634551567076694888541641414492972028354488232975466</code>

Los números enteros en Python también funcionan como secuencias de bits, y más allá que usemos la representación decimal para expresarlos, o en binario, o en hexadecimal, simplemente son números enteros. Escribirlos en otras bases es sencillo, y para verlos en esas representaciones tenemos funciones integradas:

CELL 10
<code>0b1001010</code>

<code>74</code>

CELL 11
<code>0x3f</code>

<code>63</code>

CELL 12
<code>hex(63)</code>

<code>'0x3f'</code>

CELL 13
<code>bin(74)</code>

<code>'0b1001010'</code>

Python tiene operadores para trabajar con los enteros como secuencias de bits: el “and” (&), el “or” (|), el “xor” (^), y los “shift” a izquierda y derecha (<< y >>).

CELL 14
<code>bin(0b1001 & 0b0001)</code>

<code>'0b1'</code>

CELL 15
<code>bin(0b1001 0b0001)</code>

<code>'0b1001'</code>

CELL 16
<code>bin(0b1001 ^ 0b0001)</code>

<code>'0b1000'</code>

CELL 17
<code>bin(0b1100 << 2)</code>

<code>'0b110000'</code>

CELL 18
<code>bin(0b1100 >> 2)</code>

<code>'0b11'</code>

El punto flotante binario (`float`) de Python, por otro lado, es exactamente el punto flotante ejecutado en el procesador, y tiene el mismo comportamiento que en los otros lenguajes:

CELL 19
2.35 * 28

65.8

CELL 20
15 + 2.3

17.3

CELL 21
95 / 17

5.588235294117647

CELL 22
2.3 ** 15

266635.23546439095

Así y todo siendo uno de los tipos de datos más utilizados en todos los lenguajes, hay ciertas trampas del punto flotante binario que es imperioso conocer, y a los que les dedicamos un capítulo separado en ??.

Uno de esos detalles es que no debemos utilizar `float` para manejar dinero. De esa limitación nació la implementación del punto flotante decimal, que podemos encontrar en el módulo `decimal` de la Biblioteca Estándar.

CELL 23
<code>from decimal import Decimal</code>
<code>Decimal(23) * 53</code>

<code>Decimal('1219')</code>

CELL 24
<code>Decimal(15.4) + 87</code>

<code>Decimal('102.400000000000003552713679')</code>

CELL 25
<code>Decimal('15.4') + 87</code>

<code>Decimal('102.4')</code>



La forma `from ... import ...` nos permite importar el módulo y usar sus objetos directamente, sin tener que luego repetir el nombre del módulo todas las veces. Más detalles en ??.

Como vemos en el ejemplo, no es lo mismo construir el Decimal a partir de la representación textual de un número “con coma” que de un `float`; volvemos a hacer referencia al capítulo ?? donde nos adentramos en estos detalles.

Relacionado, tenemos la posibilidad de directamente utilizar fracciones si así fuese necesario, a través del módulo `fractions`:

CELL 26

```
from fractions import Fraction

Fraction(1, 3) * 6

Fraction(2, 1)
```

CELL 27

```
Fraction(27, 5) ** 2

Fraction(729, 25)
```

CELL 28

```
Fraction(2, 3) + Fraction(6, 9)

Fraction(4, 3)
```

Finalmente, debemos mencionar el tipo de datos `complex`, integrado en el lenguaje, para manejar números complejos. En verdad tenemos soporte en la sintaxis para números imaginarios (agregándoles una `j` al final), y con ello armamos los complejos directamente:

CELL 29

```
3j

3j
```

CELL 30

```
3j + 2

(2+3j)
```

CELL 31

```
(3j + 2) * (7 + 2.5j)
(6.5+26j)
```

CELL 32

```
1j ** 2
(-1+0j)
```

CELL 33

```
import math

x = 1.5
math.cos(x) + 1j * math.sin(x) == math.e ** (x * 1j)

True
```

Las clases de cada tipo de dato funcionan como conversores entre un tipo de dato y otro, con un resultado específico en cada caso si las llamamos sin pasarles un valor.

CELL 34

```
int()
0
```

CELL 35

```
int(9.2)
9
```

CELL 36

```
float(-7)
-7.0
```

1.2. Cadenas

Las cadenas constituyen otro de los tipos de datos básicos que están presentes en infinidad de lenguajes. En Python no tienen nada de especial a primera vista, pero vamos a ir explorando algunas particularidades más adelante.

Por lo pronto hagamos la definición formal: las cadenas en Python son secuencias de caracteres Unicode delimitadas por comillas dobles o simples indistintamente.

En el siguiente ejemplo vemos casos de uno y otro delimitador, especialmente para el caso donde necesitamos usar uno de ellos como carácter dentro de la cadena:

CELL 01
<code>"Hola mundo"</code>
<code>'Hola mundo'</code>

CELL 02
<code>'Python :).'</code>
<code>'Python :).'</code>

CELL 03
<code>"Let's rock!"</code>
<code>"Let's rock!"</code>

También como en tantos otros lenguajes, la barra invertida tiene el propósito de escapar ciertos caracteres especiales (y es un último recurso para usar los delimitadores dentro de la cadena). Hay que tener en cuenta siempre que la barra invertida sirve para escaparse a sí misma, y que los caracteres especiales suelen mostrarse distinto en el modo `str` (representación más humana) que en el modo `repr` (representación más exacta):

CELL 04
<code>"Hola\nmundo"</code>
<code>'Hola\nmundo'</code>

CELL 05
<code>print("Hola\nmundo")</code>
Hola mundo

CELL 06
<code>"foo \" bar"</code>
<code>'foo " bar'</code>

CELL 07
<code>"foo \" bar ' baz"</code>
<code>'foo " bar \' baz'</code>

CELL 08
<code>'foo \\" bar'</code>
<code>'foo \\" bar'</code>

CELL 09

```
print('foo \' bar')
```

```
foo \' bar
```

Python tiene otro delimitador para cadenas: la triple comilla (ya sea doble o simple), que nos permite escribir una cadena a través de múltiples líneas:

CELL 10

```
mensaje = """
Hola
    mundo
    :)
"""
print(mensaje)
```

```
Hola
    mundo
    :)
```

CELL 11

```
mensaje
```

```
'\nHola\n    mundo\n    :)\n'
```

Las operaciones básicas a realizar sobre las cadenas están integradas en el lenguaje: saber el largo, concatenar y repetir:

CELL 12

```
len("Hola")
```

```
4
```

CELL 13

```
"Ho" + 'la'
```

```
'Hola'
```

CELL 14

```
'Na' * 5 + " Batman!"
```

```
'NaNaNaNaNa Batman!'
```

Las cadenas tienen también muchos métodos para trabajar con las mismas, veamos algunos de ellos (¡sólo ejemplos! les recomendamos revisar todos los métodos en la sección correspondiente de la Referencia de la Biblioteca [2], ya que son muy útiles en el día a día):

CELL 15

```
# transformamos la cadena a mayúsculas
"Moño".upper()

'MOÑO'
```

CELL 16

```
# hacemos que la cadena sea un "título"
"python is a great language".title()

'Python Is A Great Language'
```

CELL 17

```
# buscamos la posición de "great" en la cadena
"python is a great language".index("great")

12
```

CELL 18

```
# preguntamos si la cadena está formada exclusivamente por dígitos
"12345".isdigit()

True
```

CELL 19

```
# sacamos los "blancos" (espacios, tabs, enters) del principio y final
"\t\t corrido \n".strip()

'corrido'
```

El método `join` merece una mención especial. Arma una cadena a partir de los elementos que encuentra en otro objeto. Todavía no vimos el concepto de “iterable”, pero mencionemos que cuando recorremos una cadena vamos encontrando cada uno de sus caracteres. Entonces, el `join` construirá una nueva cadena separando los caracteres recibidos con la cadena dada.

CELL 20

```
"-".join("hola")

'h-o-l-a'
```

CELL 21

```
".|.".join("CIENCIA")

'C.|.I.|.E.|.N.|.C.|.I.|.A'
```

El `join` acepta otros iterables, por supuesto, y podemos pasarle listas, tuplas, y otros tipos de datos que veremos más adelante, siempre y cuando sus elementos sean cadenas.

Un método en particular de gran utilidad es el `format`, que permite armar cadenas reemplazando valores, lo cual es mucho más legible y ofrece más control que andar concatenando cadenas individuales. Más allá de estos ejemplos puntuales que mencionamos a continuación, el sistema de formateo en cuestión es muy poderoso, y vale la pena al menos sobrevolar la documentación [3].

CELL 22
<code>"Tiempo estimado: {:.2fs}".format(1.34844)</code>

<code>'Tiempo estimado: 1.35s'</code>

CELL 23
<code>"Valor para muestra {:05d}: {:>13s} ".format(7, "failure")</code>

<code>'Valor para muestra 00007: failure '</code>

Notablemente, no mencionamos ningún método para acceder a un carácter de la cadena, o a una subcadena de la misma. Esto se debe a que dicha funcionalidad está integrada en el lenguaje y es la misma para todas las secuencias.

La forma de acceder a un carácter de la secuencia es utilizando los corchetes, escribiendo directamente la posición entre ellos:

CELL 24
<code>cad = "Hola mundo"</code> <code>cad[0]</code>

<code>'H'</code>

CELL 25
<code>cad[2]</code>

<code>'l'</code>

CELL 26
<code>cad[-2]</code>

<code>'d'</code>

En el ejemplo vemos que la primer posición de la secuencia es la número 0, y que si utilizamos números negativos se empieza a contar desde el final (lo cual es muy práctico, porque no hay que calcular la posición que queremos restándole algo al largo de la cadena).

Para obtener subcadenas también podemos usar corchetes, pero entre ellos vamos a escribir dos valores, “desde” y “hasta”. Si no incorporamos alguno de esos números se usará un valor por defecto (“desde el principio” o “hasta el final” respectivamente), y también pueden ser valores negativos:

CELL 27
<pre>cad = "Hola mundo" cad[2:6]</pre>

'la m'

CELL 28
<pre>cad[:6]</pre>

'Hola m'

CELL 29
<pre>cad[2:]</pre>

'la mundo'

CELL 30
<pre>cad[-3:]</pre>

'ndo'

Esta operación de tomar la subcadena de una cadena en inglés se llama *slicing*, y en general usamos ese término también en castellano (porque “rebanar” cadenas nunca terminó prendiendo).

Entender qué carácter se accede por posición es sencillo (sólo tenemos que acordarnos contar desde 0), pero en el caso de los slices es más difícil, hasta que nos acostumbramos. Una buena regla mnemotécnica es pensar que también arranca desde 0, pero lo que se cuentan son las “separaciones entre las letras”. Más allá de cómo nos acordemos, la regla que tiene Python para numerar en los slices tiene dos propiedades muy útiles: por un lado si hacemos `cad[x:y]` con valores positivos el largo de la subcadena va a ser $y - x$, y por el otro es sencillo separar una cadena en dos, porque se usa el mismo número, o sea que `cad[:x] + cad[x:]` nos termina dando la misma `cad`.

También como todas las secuencias, si queremos tomar elementos de la misma pero con un determinado paso, podemos utilizar un tercer valor entre los corchetes:

CELL 31
<pre>cad = "Hola mundo, chau mundo\n" cad[:-1:2]</pre>

'Hl ud,ca ud'

En el ejemplo usamos varias propiedades al mismo tiempo. Por un lado el “desde” lo dejamos en blanco, para que tome el inicio por default, luego el “hasta” lo usamos negativo porque nos interesaba obviar el *newline* del extremo derecho, y finalmente el tercer valor, el paso, en dos, para ir agarrando cada dos caracteres.

Como con los números, podemos usar el constructor `str` de la cadena para convertir desde otros tipos de datos (así como podemos también usar los constructores de números para pasar

de cadenas a enteros o flotantes):

	CELL 32
<code>str(2.3)</code>	

<code>'2.3'</code>	

	CELL 33
<code>float("1.44")</code>	

1.44	

	CELL 34
<code>int("23")</code>	

23	

	CELL 35
<code>int("3F", base=16)</code>	

63	

Cuando mencionamos que los delimitadores de las cadenas eran la comilla doble, o simple, o sus variantes “triple”, evitamos mencionar que en todos estos casos se puede poner un prefijo para cambiar el tipo de cadena definida. Por default si no especificamos nada (o si usamos la letra “u”, por compatibilidad con versiones viejas de Python), la cadena será de tipo Unicode, lo que implica que será una secuencia de caracteres Unicode, con lo cual podemos escribir caracteres con acento, o en otros idiomas:

	CELL 36
<code>"El camión llegará mañana"</code>	

<code>'El camión llegará mañana'</code>	

	CELL 37
<code>"El valor de π es aproximadamente 3.14"</code>	

<code>'El valor de π es aproximadamente 3.14'</code>	

	CELL 38
<code>"¡Hola! привет"</code>	

<code>'¡Hola! привет'</code>	

Tengamos en cuenta que considerar a los caracteres como Unicode, más allá de su posible representación como bytes, nos permite trabajar con herramientas de transformación de texto sin mayor inconveniente:

CELL 39

```
m = "moño"
len(m)
```

4

CELL 40

```
m.upper()
```

'MOÑO'

Si necesitamos trabajar con bytes directamente, podemos tener cadenas de bytes (donde no es más una secuencia de caracteres, sino una secuencia de bytes) si prefijamos la cadena con la letra “b”. En estos casos es muy útil la notación `\x` para ingresar el valor del byte en hexadecimal, como mostramos en el siguiente ejemplo:

CELL 41

```
val = b"ab\x00\xffcd"
val
```

b'ab\x00\xffcd'

CELL 42

```
len(val)
```

6

CELL 43

```
val[3]
```

255

Es importante entender esta distinción no sólo porque las cadenas de bytes eran el default en versiones viejas de Python (y nos podemos cruzar con algún código así) sino también porque eventualmente necesitaremos convertir las cadenas Unicode a cadenas de bytes, ya que es la única manera de mandar cadenas a través de la red o grabarlas en disco.

Estas conversiones las hacemos con los métodos `encode` y `decode`. Aunque en el siguiente ejemplo parece simple, el tema de convertir de un lado para el otro le trae muchos dolores de cabeza a la mayoría de los programadores (más allá del lenguaje en que programen); les recomendamos que si quieren adentrarse en el tema vean esta charla por uno de los autores del libro [4].

CELL 44

```
mu = "moño"
mu
```

'moño'

CELL 45

```
mb = mu.encode("utf8")
mb
```

```
b'mo\xc3\xb1o'
```

CELL 46

```
mb.decode("utf8")
```

```
'moño'
```

Otro tipo de cadena usada frecuentemente es la de tipo *raw* (término que usamos en inglés, porque decir que son “sin procesar” o “crudas” es raro), donde la diferencia con las cadenas comunes es que la barra invertida NO funciona como carácter de escape (lo cual es especialmente útil al escribir expresiones regulares, tema que por respeto a los lectores NO tocaremos en el libro), sino que es tratada como un carácter normal:

CELL 47

```
r"\n"
```

```
'\n'
```

CELL 48

```
len(r"\n")
```

```
2
```

Finalmente, en las versiones más modernas de Python tenemos un tipo de cadena que se formatea automáticamente con las variables del entorno, sin tener que llamar explícitamente a `format`, permitiendo incluso expresiones y algunos detalles más:

CELL 49

```
t = 1.34844
f"Tiempo estimado: {t:.2f}s"
```

```
'Tiempo estimado: 1.35s'
```

CELL 50

```
vals = [1, 4, 2, 1, 7]
f"Proceso completo! init={vals[0]} promedio={sum(vals)/len(vals)}"
```

```
'Proceso completo! init=1 promedio=3.0'
```

1.3. Listas

Las listas son secuencias de objetos, se delimitan con corchetes, y cada objeto (sus elementos internos, que pueden ser cualquier cosa) se separan con comas.

Se acceden como cualquier secuencia, exactamente como vimos antes con las cadenas [1.2](#), e incluso tienen la misma forma de repetir las o concatenarlas:

```
CELL 01
lista = [1, "a", 5.6, "foo"]
lista
-----
[1, 'a', 5.6, 'foo']
```

```
CELL 02
len(lista)
-----
4
```

```
CELL 03
lista[1]
-----
'a'
```

```
CELL 04
lista[2:]
-----
[5.6, 'foo']
```

```
CELL 05
lista * 2
-----
[1, 'a', 5.6, 'foo', 1, 'a', 5.6, 'foo']
```

```
CELL 06
lista + [1, 2]
-----
[1, 'a', 5.6, 'foo', 1, 2]
```

La gran diferencia con las cadenas a nivel de comportamiento (más allá de que unas son secuencias de caracteres y las otras son secuencias de cualquier objeto Python) es que mientras las cadenas son “inmutables”, las listas (como muchos otros tipos de datos) son “mutables”, o sea que pueden cambiar.

El detalle fino de esto se explica un poco más adelante [1.5](#), pero veamos como para las listas efectivamente tenemos métodos que permiten modificarlas:

```
CELL 07
lista = [1, 2, 3]
lista
-----
[1, 2, 3]
```

CELL 08

```
lista.append(4)
lista
```

```
[1, 2, 3, 4]
```

CELL 09

```
otra = [5, 6]
lista.extend(otra)
lista
```

```
[1, 2, 3, 4, 5, 6]
```

¡Y no sólo agregarle elementos! También podemos reemplazar algún elemento o toda una parte:

CELL 10

```
lista
```

```
[1, 2, 3, 4, 5, 6]
```

CELL 11

```
lista[1] = 99
lista
```

```
[1, 99, 3, 4, 5, 6]
```

CELL 12

```
lista[2:] = [0, 0]
lista
```

```
[1, 99, 0, 0]
```

También podemos borrar elementos, no sólo con el método que nos permite especificar cual elemento en sí queremos borrar, sino también con la sentencia **del**.

CELL 13

```
lista
```

```
[1, 99, 0, 0]
```

CELL 14

```
lista.remove(99)
lista
```

```
[1, 0, 0]
```

CELL 15

```
del lista[1]
lista
```

```
[1, 0]
```



La sentencia `del` es bastante especial en Python, ya que *borra* elementos, lo cual nunca es trivial en un sistema que administra memoria automáticamente. En parte por eso no es una función (como `len`), sino una sentencia, porque está muy integrada con el funcionamiento base del lenguaje.

Como en los otros casos, el constructor de las listas nos es bastante útil para crear listas a partir de otros objetos. `list` construye una lista a partir de los elementos del iterable que reciba:

CELL 13

```
lista
```

```
[1, 99, 0, 0]
```

CELL 14

```
lista.remove(99)
lista
```

```
[1, 0, 0]
```

CELL 15

```
del lista[1]
lista
```

```
[1, 0]
```

CELL 16

```
list("abcd")
```

```
['a', 'b', 'c', 'd']
```



Un “iterable” es cualquier objeto capaz de devolver sus elementos internos uno por vez, lo que nos permite iterarlos en un bucle `for` o para construir una lista, por ejemplo; una cadena es un iterable que devuelve caracteres, una lista es un iterable que devuelve sus objetos internos, etc.

1.4. Tuplas

Las tuplas también son secuencias de objetos, y se definen separando estos elementos con coma (como en la definición de las listas), y opcionalmente rodeándolas con paréntesis, por claridad.

CELL 01
<code>(1, 'a', 2.3)</code>
<code>(1, 'a', 2.3)</code>
CELL 02
<code>1, 5, 'xx'</code>
<code>(1, 5, 'xx')</code>
CELL 03
<code>t = (5,)</code>
<code>t</code>
<code>(5,)</code>
CELL 04
<code>len(t)</code>
<code>1</code>
CELL 05
<code>t[0]</code>
<code>5</code>

Una diferencia fundamental con las listas es que las tuplas son inmutables, y en general una buena recomendación para decidir su uso en un caso u otro es entender si para los objetos internos es importante su cantidad y posición. Por ejemplo, la lista es mejor para guardar los archivos que hay en un directorio (más allá del orden en sí, el archivo no es más o menos archivo si está antes o después, y en el directorio podemos tener 0, 3, 27, 1000 archivos), en cambio la tupla es mejor para guardar coordenadas (si es en el plano vamos a tener siempre una tupla de dos elementos, ni uno ni tres, y es importante si un número está primero o segundo, ya que son ejes distintos).

En realidad la diferencia es aún más profunda: las tuplas forman parte del núcleo más central del lenguaje, ya que se usa en infinidad de detalles internos. Por ejemplo, si recordamos la función interna `divmod` que explicamos cuando vimos números, veremos que esa función devuelve dos números... en verdad devuelve una tupla, que por otra característica de Python (“desempaquetado de tuplas”, que también tendemos a denominar en inglés como *tuple unpacking*) podemos usar directamente como si fueran dos objetos:

CELL 06

```
divmod(60, 7)
```

```
(8, 4)
```

CELL 07

```
a, b = divmod(60, 7)
print("a={}, b={}".format(a, b))
```

```
a=8, b=4
```

Obviamente necesitamos la misma cantidad de elementos en la izquierda y la derecha del igual, para que el “desempaquetado” funcione, aunque tenemos la posibilidad de usar un “expansor” para que tome múltiples argumentos:

CELL 08

```
a, b, c = 1, 2
```

```
ValueError                                Traceback (most recent call last)
Input In [8], in <cell line: 1>()
----> 1 a, b, c = 1, 2
```

```
ValueError: not enough values to unpack (expected 3, got 2)
```

CELL 09

```
a, b, c = 1, 2, 3, 4, 5, 6
```

```
ValueError                                Traceback (most recent call last)
Input In [9], in <cell line: 1>()
----> 1 a, b, c = 1, 2, 3, 4, 5, 6
```

```
ValueError: too many values to unpack (expected 3)
```

CELL 10

```
a, b, *c = 1, 2, 3, 4, 5, 6
print("a={}, b={}, c={}".format(a, b, c))
```

```
a=1, b=2, c=[3, 4, 5, 6]
```

CELL 11

```
a, *b, c = 1, 2, 3, 4, 5, 6
print("a={}, b={}, c={}".format(a, b, c))
```

```
a=1, b=[2, 3, 4, 5], c=6
```

Una vez más, el constructor es la herramienta principal para convertir entre tipos:

CELL 12

```
tuple("foobar")

('f', 'o', 'o', 'b', 'a', 'r')
```

CELL 13

```
tuple([2, None, 0.1])

(2, None, 0.1)
```

Mencionábamos antes que en las tuplas cada elemento no es uno más sino que su posición es de importancia semántica. El siguiente paso en ese sentido es poder asignarle un nombre a cada posición, y para ello tenemos el tipo `namedtuple` (en el módulo `collections` de la biblioteca estándar).

Para usarlo necesitamos declarar la estructura, indicando los campos que vamos a tener y un nombre para esa estructura. Como ejemplo armamos una tupla que contiene un ítem de un subtítulo: desde y hasta cuando debería mostrarse en la película, y el texto.

CELL 14

```
from collections import namedtuple

SubItem = namedtuple("SubItem", ["time_from", "time_to", "text"])
```

Por simplicidad, podemos directamente nombrar los campos en una única cadena (incluso separándolos por coma), sin que cambie su significado:

CELL 15

```
# 100% equivalentes a la definición anterior
SubItem = namedtuple("SubItem", "time_from, time_to, text")
SubItem = namedtuple("SubItem", "time_from time_to text")
```

La podemos usar igual que una tupla normal, reemplaza a una tupla en todos sus aspectos, pero ya incluso al ver el objeto notamos que nos provee mucha más información que sólo los tres valores entre paréntesis:

CELL 16

```
item1 = SubItem(2.3, 3.7, "Hola mundo")
item1

SubItem(time_from=2.3, time_to=3.7, text='Hola mundo')
```

Incluso la creación del objeto puede ser usando los nombres (obviamente sin importar el orden), con lo cual ya es imposible confundirse cuál valor corresponde a cuál posición:

CELL 17

```

item2 = SubItem(text="Chau mundo", time_from=4.5, time_to=12.3)
item2

-----

SubItem(time_from=4.5, time_to=12.3, text='Chau mundo')

```

A los elementos los podemos acceder por posición (ya dijimos que se comporta como una tupla normal) con lo cual a priori no tenemos que modificar el código que usa este elemento si reemplazamos una tupla normal por una `namedtuple`, pero la gran ventaja aquí es que podemos usar el nombre del atributo:

CELL 18

```

print(item1[2], item1.text)
print(item2[0], item2.time_from)

-----

Hola mundo Hola mundo
4.5 4.5

```

Noten en el segundo ítem como al acceder por posición se utiliza la definición de la estructura y no cómo se haya creado el elemento en sí.

Como cualquier nombre puede ser un atributo definido por el usuario, las `namedtuple` tienen sus propios métodos y atributos prefijados con un guión bajo, aunque no sean privados como a priori indicaría la convención.

Los más usados son para convertir la tupla en diccionario (usando los nombres como claves), listar los campos, e incluso generar una nueva tupla reemplazando algunos valores (ya que no podemos cambiar la `namedtuple` en sí, es tan inmutable como una tupla normal).

CELL 19

```

item1._asdict()

-----

{'time_from': 2.3, 'time_to': 3.7, 'text': 'Hola mundo'}

```

CELL 20

```

item1._fields

-----

('time_from', 'time_to', 'text')

```

CELL 21

```

item1._replace(time_to=4.1)

-----

SubItem(time_from=2.3, time_to=4.1, text='Hola mundo')

```

A los campos les podemos asignar valores por defecto, de manera que si los omitimos al crear el objeto tomarán esos valores:

```

CELL 22

SubItem = namedtuple("SubItem", "time_from time_to text", defaults=[])

item1 = SubItem(2, 3.5, "Un texto")
print(item1)

item2 = SubItem(2, 3.5)
print(item2)

-----

SubItem(time_from=2, time_to=3.5, text='Un texto')
SubItem(time_from=2, time_to=3.5, text='')

```

Los valores por defecto incluidos en esa lista corresponden a los últimos nombres definidos (ya que no pueden quedar argumentos sin valor por defecto luego de los que sí tienen, igual a lo que sucede con una función).

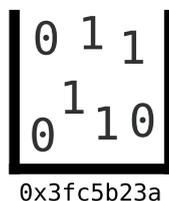
1.5. Pensando como un pythonista

Hay una diferencia fundamental en cómo Python maneja los objetos internamente en la ejecución de un programa con respecto a otros lenguajes, y es imperativo entenderla porque nos explica mucho del funcionamiento del lenguaje.

Python no tiene “variables”, y en consecuencia esas variables no tienen “valores”. Estos son términos y conceptos que provienen de otros lenguajes de más bajo nivel (es decir, más cercanos al procesador), y no se aplican en Python.

Python tiene “objetos”, y usamos “nombres” para hacer referencia a esos objetos. Es verdad que a veces por facilismo o sobresimplificación usamos las palabras “variable” y “valor” también cuando hablamos de un programa en Python o el funcionamiento del lenguaje, pero eso no nos tiene que confundir sobre el funcionamiento real.

Veamos la diferencia. Otros lenguajes, como C por ejemplo, sí poseen el concepto de “variable”, como lugar donde se guarda un valor. Ese lugar es en memoria, y el valor que se guarda son los bits en esa posición de memoria.



En estos lenguajes es necesario declarar cómo se interpretan esos bits en esa posición de memoria. Por ejemplo, varios bytes seguidos pueden ser varias letras, o un número entero, e incluso podemos decirle al lenguaje que nos muestre esos bits de una manera u otra:

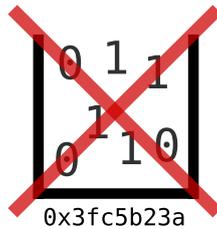
```

int a = 7303014;
printf("%i %s", a, (char *)&a);

```

```
--> 7303014 foo
```

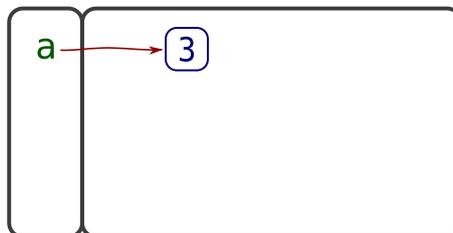
En cambio, en Python no seguimos esa filosofía.



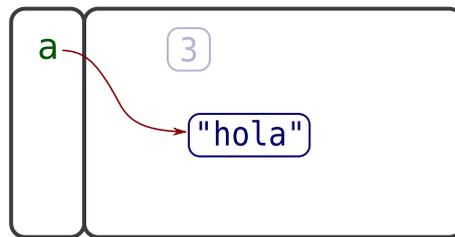
En Python manejamos directamente objetos. Estos objetos son de un tipo específico, y no pueden cambiar de tipo (“tipado fuerte”). Para referenciar esos objetos en memoria usamos nombres.



Estos nombres pueden apuntar a uno u otro objeto, y no están asociados al tipo del objeto que vinculan. Entonces, no hace falta decir que el nombre a va a ser un entero o una cadena, por ejemplo, porque el tipo pertenece al objeto que es nombrado. Cuando decimos que `a = 3` sólo estamos creando un objeto en memoria de tipo entero (el 3) y a ese objeto lo estamos referenciando usando el nombre a.



Si luego escribimos que `a = "ho1a"`, estamos creando otro objeto en memoria, este de tipo cadena, con el valor "ho1a", y estamos usando el mismo nombre a para referenciarlo. No es que a haya cambiado de tipo, sólo apunta a otro objeto.



Obviamente, como `a` apunta al segundo objeto, ya no apunta al primero. Si ese objeto no es accesible desde ningún otro lado, para todo propósito nosotros podemos considerar que ya no existe (eventualmente Python lo eliminará y liberará memoria, pero esto es algo que podemos ignorar tranquilamente ya que Python administra la memoria por nosotros).

En los diagramas mostrados arriba vemos dos áreas diferentes. El grande es el espacio de objetos (a grandes rasgos lo que llamamos “la memoria”, sin entrar en detalle cual sección de memoria en particular o cómo es manejada por Python). La columna de la izquierda, donde por ahora tenemos el nombre `a`, es llamada “espacio de nombres”, una sección en particular (de la memoria, obviamente, porque de última todo está en memoria, pero que no se nos mezcle con la otra parte genérica) donde guardamos los nombres, que son simplemente cadenas. En el espacio de nombres no podemos almacenar otra cosa que nombres, y estos nombres siempre apuntan a objetos “en la memoria” (no pueden apuntar a otros nombres en el espacio de nombres).

Python tiene un espacio de nombres “global”, que está siempre disponible durante la ejecución del programa y accesible de cualquier lado, y va creando otros espacios de nombres en diferentes momentos y con una accesibilidad limitada. Por ejemplo, cuando se ejecuta una función esta tiene su propio “espacio de nombres local”, diferente al global y diferente a los espacios de nombres de otras funciones, que es lo que permite que cada función pueda usar un mismo nombre apuntando a diferentes objetos y no entren en conflicto.

En los diagramas, además de las dos secciones, y el nombre en una y el objeto en otra, vemos una flecha que va del nombre al objeto. Esta flecha representa justamente que ese nombre referencia a dicho objeto. Por eso cuando en Python escribimos `a = "¡Hola!"` estamos realmente *vinculando* el nombre `a` con el “objeto cadena con valor `"¡Hola!"`” (en inglés el verbo es *bind*).

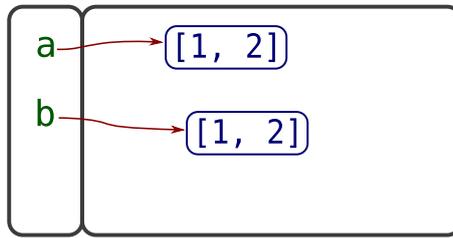
Retomemos el uso de estos diagramas mostrando los nombres y su vinculación a los objetos, y la diferencia fundamental entre objetos mutables e inmutables, que en conjunto son claves para entender cómo funciona Python internamente (para que eventualmente logremos “pensar como un Pythonista”).

Vayamos entonces con un ejemplo más complejo.

Ahora en los siguientes dos pasos vinculamos primero el nombre `a` a una lista con dos números, y luego `b` a otra lista con los mismos dos números.

```

CELL 01
a = [1, 2]
b = [1, 2]
```



(El diagrama tiene una simplificación, en pos de la legibilidad: en verdad la lista no tiene a los números “adentro”, sino que los números son otros objetos en la memoria, y desde cada posición de la lista se los referencia; vamos a ver esto mismo en los próximos diagramas cuando la complejidad lo amerite, pero en este caso solamente para los números no vale la pena.)

Vemos que en memoria tenemos dos objetos lista con el mismo contenido. Podemos adivinar que si le preguntamos a Python si ambos nombres apuntan a objetos iguales, nos dirá que sí, pero si le preguntamos si los dos nombres apuntan *al mismo objeto*, nos dirá que no.

CELL 02	
<code>a == b</code>	
True	

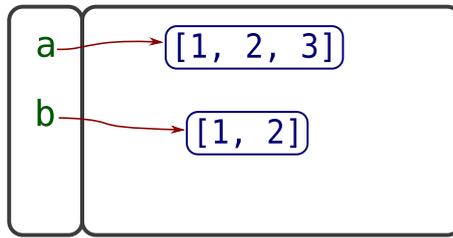
CELL 03	
<code>a is b</code>	
False	

El primer comparador es el de “igualdad”, mientras que el segundo es de “identidad”. Vamos a ver más comparadores luego cuando veamos el `if` en ??.

Modifiquemos ahora una de las listas.

CELL 04	
<code>a.append(3)</code> <code>a</code>	
[1, 2, 3]	

CELL 05	
<code>b</code>	
[1, 2]	



Vemos que la lista a ahora tiene un elemento más, mientras que la b sigue con su estado anterior, como podíamos esperar.

Intentemos algo nuevo, ahora usemos el = entre dos nombres. Acá es importante que recordemos que no podemos vincular un nombre a otro nombre (apuntar con la flechita de un nombre a otro nombre), sino que lo que termina sucediendo es que ese nombre nuevo queda vinculado al objeto que apuntaba el otro nombre.

```

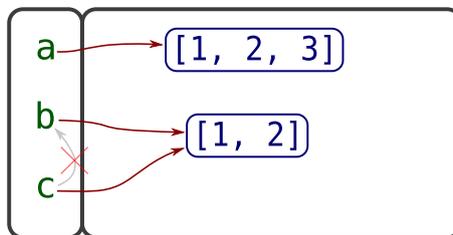
CELL 06
c = b
c
-----
[1, 2]
    
```

```

CELL 07
c == b
-----
True
    
```

```

CELL 08
c is b
-----
True
    
```



Es por eso que si ahora comparamos la igualdad y la identidad entre c y b Python contestará verdadero en ambos casos, porque ambos nombres apuntan al mismo objeto.

Si ahora modificamos la lista c, vemos que también se modifica b. Habiendo escrito eso, hagamos el ejercicio de entender que esa frase es inexacta, y casi tramposa. Porque c o b no son

listas, son nombres que apuntan a una lista, la misma, entonces tampoco tiene sentido decir que “también se modifica”. Entonces, escribiendo esa frase correctamente, podríamos decir que “podemos modificar la lista y ver ese cambio, usando tanto un nombre como el otro”, que es lo que vemos en el siguiente paso.

```

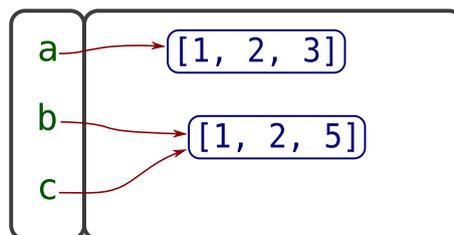
CELL 09
c.append(5)
c
-----
[1, 2, 5]

```

```

CELL 10
b
-----
[1, 2, 5]

```



Arranquemos otro ejemplo.

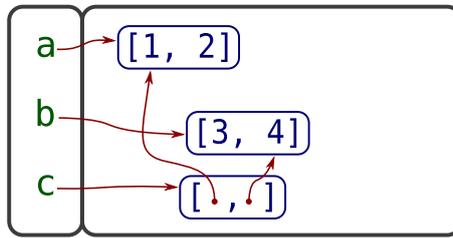
Volvemos a tener dos listas con números, y ahora también armamos una tercer lista que incluye las primeras dos. En verdad las dos primeras listas no están “adentro” de la primera, sino que son referenciadas en cada posición. Esto es exactamente lo mismo que mencionábamos arriba acerca de que dibujábamos a los números adentro de la lista pero en verdad eran objetos también en la zona de memoria, referenciados desde la lista.

En este caso en el dibujo seguimos con la simplificación de dibujar a los números adentro de la lista, pero somos más precisos con la tercer lista apuntando a las primeras dos.

```

CELL 11
a = [1, 2]
b = [3, 4]
c = [a, b]
c
-----
[[1, 2], [3, 4]]

```



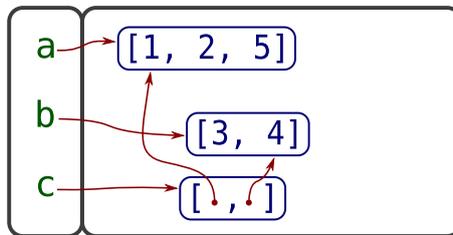
Si ahora modificamos la lista que llamamos a, obviamente vamos a ver que esa lista está cambiada, pero también vamos a ver reflejado ese cambio si miramos c, ya que estamos hablando en definitiva de la misma lista vista directamente o a través de una posición de la otra lista.

```

CELL 12
a.append(5)
a
-----
[1, 2, 5]
    
```

```

CELL 13
c
-----
[[1, 2, 5], [3, 4]]
    
```



Por otro lado, si hacemos `b = "foo"` tenemos que entender que no estamos modificando el objeto referenciado por b sino que realmente estamos revinculando el nombre b a otro objeto, y por lo tanto el objeto anterior (el que dejamos de referenciar cuando apuntamos b a otro lado) queda intacto.

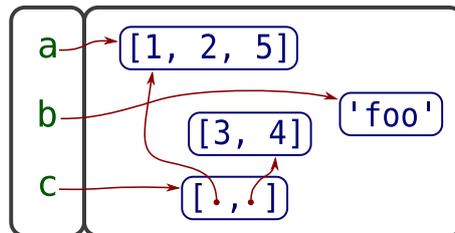
```

CELL 14
b = "foo"
b
-----
'foo'
    
```

```

CELL 15
c
-----
[[1, 2, 5], [3, 4]]

```



A diferencia del ejemplo que teníamos al principio de la sección, donde el objeto “viejo” quedaba sin uso y decíamos que Python eventualmente lo iba a eliminar para liberar memoria, en este caso el objeto que antes era `b` no queda totalmente desreferenciado, sigue siendo apuntado por `c`, con lo cual permanecerá en memoria sin problemas.

E incluso podríamos volver a asignarle otro nombre, y seguir interactuando con el objeto directamente a través del nuevo nombre:

```

CELL 16
d = c[1]
d
-----
[3, 4]

```

```

CELL 17
d.append(7)
d
-----
[3, 4, 7]

```

```

CELL 18
c
-----
[[1, 2, 5], [3, 4, 7]]

```

Como nota final, cabe destacar que el comparador de identidad `is`, con la excepción de cuando preguntamos `is None` o `is not None` (que queda bien en inglés), se usa en contadísimas ocasiones, en general necesitamos comparar por igualdad con el `==`.

1.6. Conjuntos

Los conjuntos son contenedores de objetos, se delimitan con llaves, y cada objeto (sus elementos internos) se separan con comas.

A diferencia de otros contenedores que vimos previamente, los conjuntos no son secuencias, ya que los elementos dentro del conjunto no tienen un orden en particular. Es hasta inexacto decir que “están desordenados”, porque directamente este tipo de dato no tiene el concepto de orden.

Las propiedades más interesantes de los conjuntos son las matemáticas: cada elemento puede estar solamente una vez, y además podemos realizar intersecciones, uniones y diferencias entre ellos.

CELL 01
<pre>frutas = {'manzana', 'naranja', 'sandía', 'limón'} frutas</pre> <hr style="border-top: 1px dashed #000;"/> <pre>{'limón', 'manzana', 'naranja', 'sandía'}</pre>
CELL 02
<pre>vitamina_c = {'limón', 'perejil', 'fresa', 'naranja'}</pre>
CELL 03
<pre>vitamina_c</pre> <hr style="border-top: 1px dashed #000;"/> <pre>{'fresa', 'limón', 'naranja', 'perejil'}</pre>
CELL 04
<pre>frutas & vitamina_c</pre> <hr style="border-top: 1px dashed #000;"/> <pre>{'limón', 'naranja'}</pre>
CELL 05
<pre>frutas - vitamina_c</pre> <hr style="border-top: 1px dashed #000;"/> <pre>{'manzana', 'sandía'}</pre>
CELL 06
<pre>frutas vitamina_c</pre> <hr style="border-top: 1px dashed #000;"/> <pre>{'fresa', 'limón', 'manzana', 'naranja', 'perejil', 'sandía'}</pre>

El conjunto es el tipo de dato dejado de lado más injustamente de todo el lenguaje, sospechamos que esto es así porque en otros lenguajes no se encuentra, entonces los programadores no se acostumbran a usarlo. Sin embargo, es muy poderoso y una vez que aprendamos a expresar nuestros algoritmos utilizando las capacidades de los conjuntos, evitaremos hacer un montón de bucles y comparaciones innecesarias en nuestros programas.

Sus elementos internos pueden ser solamente objetos inmutables (en verdad, podemos tener objetos que hagamos nosotros que siendo mutables, se les pueda calcular el `hash` en función de propiedades que no cambien, pero por simplicidad pensemos en objetos inmutables). Esto es porque para decidir si un elemento pertenece a un conjunto se usa su hash (de forma similar a los

diccionarios que veremos luego). Una implicancia de esta propiedad es que es extremadamente rápido verificar si un elemento pertenece a un determinado conjunto, aunque este sea muy grande (porque se calcula su hash, y listo), en contraposición con lo que sucede con una lista o una tupla (porque en estos casos hay que comparar todos los objetos, algo potencialmente prohibitivo dependiendo del tamaño del contenedor).



En este contexto, un hash es un número entero calculado a partir de los atributos del objeto. Aquellos objetos iguales tendrán el mismo hash. Los conjuntos y diccionarios usan este hash como referencia en su funcionamiento, de ahí la restricción de que los objetos no cambien (para que sigan teniendo siempre el mismo hash).

Los conjuntos son objetos mutables, y como tales tienen métodos para agregar y eliminar elementos internos.

En el siguiente ejemplo vemos como podemos agregar objetos de a uno, o muchos simultáneamente (usando un iterable que los contiene). En ambos casos, tenemos que recordar la propiedad de los conjuntos de que no pueden tener objetos repetidos.

CELL 07

```
música = {'metal', 'tango', 'rock'}
música
-----
{'metal', 'rock', 'tango'}
```

CELL 08

```
música.add('rock')
música
-----
{'metal', 'rock', 'tango'}
```

CELL 09

```
música.add('jazz')
música
-----
{'jazz', 'metal', 'rock', 'tango'}
```

CELL 10

```
música.update(['tango', 'cumbia'])
música
-----
{'cumbia', 'jazz', 'metal', 'rock', 'tango'}
```

CELL 11

```
'rock' in música
```

```
True
```

Tenemos varias formas de remover elementos de los conjuntos, una que saca el elemento indicado pero falla si no está, otra que saca el elemento indicado (sin fallar si no está), y otra que saca un elemento de forma arbitraria (fallando si el conjunto estaba ya vacío):

CELL 12

```
música.remove('jazz')
música
```

```
{'cumbia', 'metal', 'rock', 'tango'}
```

CELL 13

```
música.remove('jazz')
```

```

KeyError                                Traceback (most recent call last)
<ipython-input-13-533e3682df9b> in <module>
----> 1 música.remove('jazz')

```

```
KeyError: 'jazz'
```

CELL 14

```
música.discard('metal')
música
```

```
{'cumbia', 'rock', 'tango'}
```

CELL 15

```
música.discard('metal')
música
```

```
{'cumbia', 'rock', 'tango'}
```

CELL 16

```
música.pop()
```

```
'tango'
```

CELL 17

```
música
```

```
{'cumbia', 'rock'}
```

Como con el resto de los tipos, podemos usar el constructor para convertir entre ellos; en este caso el constructor toma cualquier iterable y se queda con los elementos que recibe:

CELL 18

```
set(['foo', 'bar'])
-----
{'bar', 'foo'}
```

CELL 19

```
set("son ocho los orozco")
-----
{' ', 'c', 'h', 'l', 'n', 'o', 'r', 's', 'z'}
```

1.7. Diccionarios

Los diccionarios también se delimitan con llaves, como los conjuntos, pero en este caso lo que se separan por comas son pares de objetos, cada par siendo una clave y un valor (separados entre sí por dos puntos).

Es que los diccionarios, a diferencia de los otros contenedores que vimos hasta ahora (listas, conjuntos, etc) guardan objetos identificados cada uno por una determinada clave. Los valores guardados pueden ser cualquier objeto de Python, mientras que las claves pueden ser objetos que se les puede calcular el hash.

La forma más directa de acceder a los valores guardados es a través de sus respectivas claves, pero también podemos listar todas las claves, todos los valores, e incluso todos los ítems (pares clave/valor).

CELL 01

```
d = {'foo': 23, 'bar': 90, 'baz': 133}
d
-----
{'foo': 23, 'bar': 90, 'baz': 133}
```

CELL 02

```
d['foo']
-----
23
```

CELL 03

```
d.keys()
-----
dict_keys(['foo', 'bar', 'baz'])
```

CELL 04

```
d.values()
-----
dict_values([23, 90, 133])
```

CELL 05

```
d.items()
-----
dict_items([('foo', 23), ('bar', 90), ('baz', 133)])
```

Cada clave puede estar una sola vez en el diccionario (porque se accede a través del hash de las claves, de forma similar a lo que veíamos con los conjuntos), entonces si asignamos un nuevo valor a una clave ya presente, estaremos pisando el valor anterior. Si la clave no estaba, estaremos creando un nuevo ítem en el diccionario.

CELL 06

```
d
-----
{'foo': 23, 'bar': 90, 'baz': 133}
```

CELL 07

```
d['foo'] = 1288
d
-----
{'foo': 1288, 'bar': 90, 'baz': 133}
```

CELL 08

```
d['otra'] = 7
d
-----
{'foo': 1288, 'bar': 90, 'baz': 133, 'otra': 7}
```

Tengamos en cuenta que desde Python 3.6 los diccionarios recuerdan el orden de inserción de sus claves (antes no tenían un orden en particular).

Si accedemos a una clave que no está presente en el diccionario, se genera una excepción de tipo **KeyError**. A veces es útil usar el método `get` que accede al diccionario y devuelve el valor correspondiente a la clave si es que la clave existe, pero en caso de que la clave no exista devolverá un valor que podemos especificar (o **None** si no indicamos nada).

CELL 09

```
d
-----
{'foo': 1288, 'bar': 90, 'baz': 133, 'otra': 7}
```

CELL 10

```
d['foo']
-----
1288
```

CELL 11

```
d['fux']
```

```

KeyError                                Traceback (most recent call last)
<ipython-input-11-15405d883549> in <module>
----> 1 d['fux']

KeyError: 'fux'

```

CELL 12

```
d.get('foo')
```

```
1288
```

CELL 13

```
d.get('fux', "nada")
```

```
'nada'
```

CELL 14

```
d.get('fux')
```

Podemos borrar elementos del diccionario con la declaración **del**, pero también a veces es útil extraer el valor correspondiente a la clave que estamos borrando, y para eso tenemos el método `pop`, al que si le pasamos un valor por default soportará no encontrar a la clave indicada.

CELL 15

```
d
```

```
{'foo': 1288, 'bar': 90, 'baz': 133, 'otra': 7}
```

CELL 16

```
del d['bar']
```

```
d
```

```
{'foo': 1288, 'baz': 133, 'otra': 7}
```

CELL 17

```
d.pop('foo')
```

```
1288
```

CELL 18

```
d
```

```
{'baz': 133, 'otra': 7}
```

CELL 19

```
d.pop('foo')
```

```

KeyError                                Traceback (most recent call last)
<ipython-input-19-9d43697015c6> in <module>
----> 1 d.pop('foo')

KeyError: 'foo'

```

CELL 20

```
d.pop('foo', "no estaba")
```

```
'no estaba'
```

Podemos crear diccionarios a partir de otras estructuras, cualquier iterable en verdad, pero tendrán que ser de pares de valores, donde cada par será clave/valor. También podemos crear diccionarios tomando como fuente las claves, usando el método `fromkeys`, pero en este caso las claves tendrán siempre el mismo valor inicial. Y como detalle especial con este tipo de datos podemos usar una forma de pasar parámetros a las funciones que en este caso es particularmente útil (aunque sólo para el caso donde las claves son cadenas).

CELL 21

```
dict([('a', [1, 2]), ('b', [3, 4])])
```

```
{'a': [1, 2], 'b': [3, 4]}
```

CELL 22

```
dict.fromkeys(['foo', 'bar'])
```

```
{'foo': None, 'bar': None}
```

CELL 23

```
dict.fromkeys(['foo', 'bar'], 0)
```

```
{'foo': 0, 'bar': 0}
```

CELL 24

```
dict(a=[1, 2], b=[3, 4])
```

```
{'a': [1, 2], 'b': [3, 4]}
```

1.8. Iteradores

Antes de irnos de la Sección de Tipos de datos, mencionemos un concepto importante en Python: los iteradores.

A nivel de definición general, un iterador se refiere al objeto que permite al programador

recorrer un contenedor. Prestemos atención a la diferencia entre el contenedor en sí (como colección de elementos) del iterador (que nos permite recorrer esos elementos).

Esta característica se expresa en muchos rincones del lenguaje. Tenemos la declaración `for`, que veremos más adelante en la Sección ??, que nos permite construir un bucle alrededor de iterar un objeto, pero también podemos hacerlo a mano, aunque de esta manera tenemos que pedirle al objeto iterable que nos de un iterador:

```

CELL 01
números = [1, 2, 3]
type(números)

list

```

```

CELL 02
iterador = iter(números)
type(iterador)

list_iterator

```

```

CELL 03
next(iterador)

1

```

```

CELL 04
next(iterador)

2

```

```

CELL 05
next(iterador)

3

```

```

CELL 06
next(iterador)

StopIteration                                Traceback (most recent call last)
<ipython-input-6-2389250a88e0> in <module>
----> 1 next(iterador)

StopIteration:

```

Esa excepción que vemos ahí es perfectamente normal, es el mecanismo que tienen los iteradores para indicar que no hay más elementos para entregar.

Una forma útil en el intérprete interactivo de iterar un objeto y ver el resultado es a través del constructor `list`. Usémoslo y veamos como en Python muchos tipos integrados en el lenguaje soportan el protocolo de iteración (decimos que son “iterables”):

CELL 07

```
list([1, 2, 3]) # una lista
```

```
[1, 2, 3]
```

CELL 08

```
list((1, 2, 3)) # una tupla
```

```
[1, 2, 3]
```

CELL 09

```
list("abcd") # una cadena
```

```
['a', 'b', 'c', 'd']
```

CELL 10

```
list({1, 2, 3}) # un conjunto
```

```
[1, 2, 3]
```

CELL 11

```
list({'a': 1, 'b': 2, 'c': 3}) # un diccionario
```

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

No todos los tipos soportan este protocolo, claro. Por ejemplo, no se puede iterar un número entero, ya que no es un contenedor, no tiene elementos para entregar.

También tengamos en cuenta que normalmente cada vez que pidamos un iterador vamos a tener uno “fresco”, que arrancará desde el principio (aunque nosotros podríamos cambiar este comportamiento si hacemos nuestros propios tipos de datos). Pero si trabajamos sobre el iterador puntualmente, podemos pedirle elementos de diversas maneras sin tener que volver a comenzar.

CELL 12

```
lista = [1, 2, 3, 4, 5, 6, 7]
```

```
list(lista)
```

```
[1, 2, 3, 4, 5, 6, 7]
```

CELL 13

```
iterador = iter(lista)
```

```
iterador
```

```
<list_iterator at 0x7f4a0cd2b700>
```

CELL 14

```
next(iterador)
```

1

CELL 15

```
next(iterador)
```

2

CELL 16

```
list(iterador)
```

```
[3, 4, 5, 6, 7]
```

Finalmente, prestemos atención cuando estamos iterando un objeto, ya que el mismo puede potencialmente entregar una cantidad muy grande de elementos, lo cual no tiene sentido tener en memoria simultáneamente. Mostremos un ejemplo de esto con la función integrada `range`, que entrega números en un determinado rango, ahí vemos como para un rango chico podemos iterarlo completamente con una lista, pero para uno muy grande, aunque podamos iterarlo a mano, realmente no tiene sentido hacerlo hasta el final.

CELL 17

```
nros = range(3)
list(nros)
```

```
[0, 1, 2]
```

CELL 18

```
muchos_nros = range(1000000000000000)
iter_muchos_nros = iter(muchos_nros)
next(iter_muchos_nros)
```

0

CELL 19

```
next(iter_muchos_nros)
```

1

Así y todo `range` en algún momento va a terminar. Pero podemos tener generadores que sean infinitos, ya veremos como construirlos cuando hablemos de Funciones ??.

Parte II

Herramientas fundamentales

Los capítulos siguientes desarrollan un conjunto de conocimientos sobre herramientas fundamentales que serán aprovechados en la Parte III al abordar temas de aplicaciones específicas.

Parte III

Temas específicos

En esta Parte desarrollaremos capítulos que abordan temas específicos de aplicación de herramientas de Python. Cada capítulo está autocontenido, de forma que el lector o lectora puede acceder directamente al tema de interés sin realizar un recorrido secuencial de los capítulos, aunque serán utilizados conceptos y herramientas tratados en la Parte [II](#).

Parte IV
Apéndices

A | Zen de Python

Incluimos aquí las frases traducidas correspondientes al Zen de Python [5].

- Bello es mejor que feo.
- Explícito es mejor que implícito.
- Simple es mejor que complejo.
- Complejo es mejor que complicado.
- Plano es mejor que anidado.
- Espaciado es mejor que denso.
- La legibilidad es importante.
- Los casos especiales no son lo suficientemente especiales como para romper las reglas.
- Sin embargo la practicidad le gana a la pureza.
- Los errores nunca deberían pasar silenciosamente.
- A menos que se silencien explícitamente.
- Frente a la ambigüedad, evitar la tentación de adivinar.
- Debería haber una, y preferiblemente solo una, manera obvia de hacerlo.
- A pesar de que esa manera no sea obvia a menos que seas Holandés.
- Ahora es mejor que nunca.
- A pesar de que nunca es muchas veces mejor que *ahora* mismo.
- Si la implementación es difícil de explicar, es una mala idea.
- Si la implementación es fácil de explicar, puede que sea una buena idea.
- Los espacios de nombres son una gran idea, ¡tenemos más de esos!

Bibliografía

- [1] URL: <https://creativecommons.org/licenses/by-nc-sa/4.0/deed.es>.
- [2] URL: <https://docs.python.org/dev/library/stdtypes.html#string-methods>.
- [3] URL: <https://docs.python.org/dev/library/string.html#format-string-syntax>.
- [4] URL: <https://www.youtube.com/watch?v=jAZ-NyAwpsg>.
- [5] Tim Peters. *The Zen of Python*. 19 de ago. de 2004. URL: <https://www.python.org/dev/peps/pep-0020/>.