**4. Algoritmos sobre listas y comprensión de listas**

En esta clase introducimos el debugger y les proponemos una serie de ejercicios donde tienen que escribir algoritmos que operen sobre listas. Luego introducimos la *comprensión de listas*, un concepto muy hermoso y pythonesco. Cerramos la clase con una discusión sobre el concepto de objeto que subyace al lenguaje y algunos ejercicios para repasar estos conceptos usando el dataset de la clase pasada.

**Recordá:** aunque no termines todos los ejercicios, completá el formulario del final de la clase y subí los que hayas podido hacer **ANTES** del día miércoles siguiente a las 14hs. Esto es parte de las condiciones de aprobación de la materia: todes podemos tener una semana complicada, pero en general esperamos que entregues la mayoría de los ejercicios.

* [4.1 Debuggear programas](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/04_Listas_y_Listas/01_Debugger.md)
* [4.2 Listas y búsqueda lineal](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/04_Listas_y_Listas/02_IteradoresLista.md)
* [4.3 Comprensión de listas](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/04_Listas_y_Listas/03_Comprension_Listas.md)
* [4.4 Objetos](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/04_Listas_y_Listas/04_Objetos.md)
* [4.5 Arbolado porteño y comprensión de listas](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/04_Listas_y_Listas/05_Arboles2_LC.md)
* [4.6 Cierre de la clase](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/04_Listas_y_Listas/06_Cierre.md)

[Contenidos](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/Contenidos.md) | [Anterior (3 Traba](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/03_Datos/00_Resumen.md)

**4.1 Debuggear programas**

Python tiene un debugger poderoso que te permite probar porciones de código. Esto es sencillo y está integrado en IDEs como Spyder.

Vimos en la [Sección 3.1](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/03_Datos/01_Bugs.md#tres-tipos-de-errores) diferentes ejemplos de problemas que pueden aparecer y tuviste que arremangarte e ingeniártelas para resolverlos a mano. En esta sección vamos a introducir la herramientas *pdb* (Python debugger) que ofrece el lenguaje para resolver este tipo de problemas.

**Testear es genial, debuggear es horrible.**

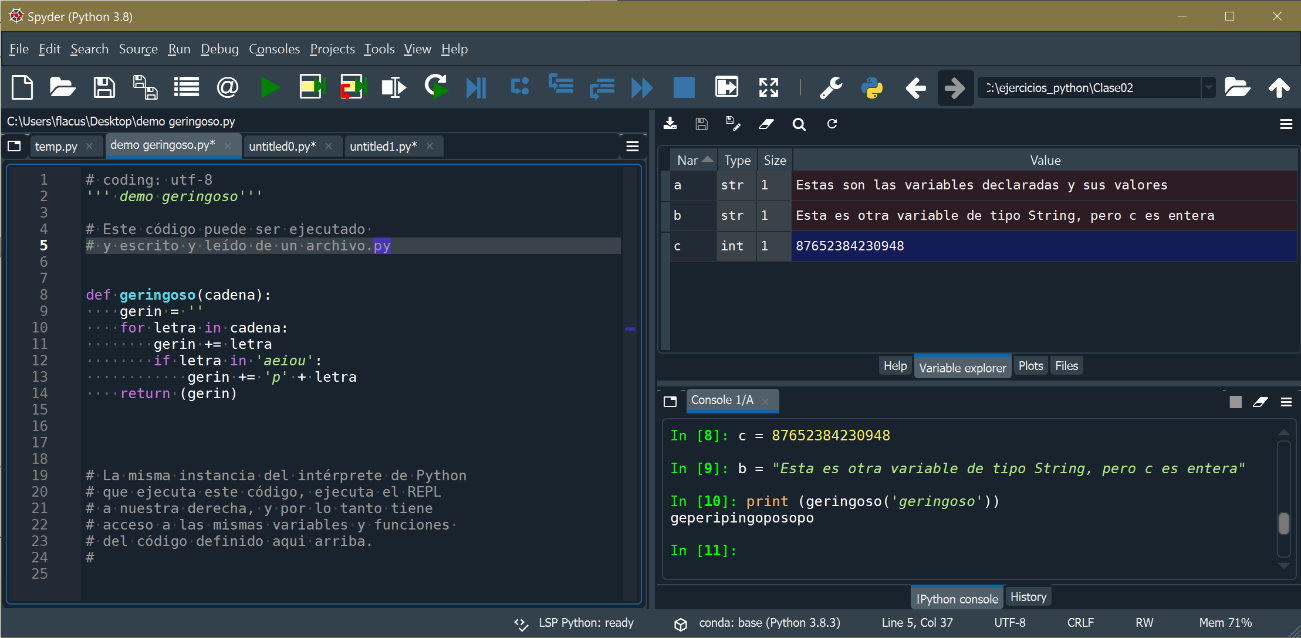
Se dice que hay un *bug* (un error) cuando un programa no se comporta como el programador espera o hace algo inesperado. Es muy frecuente que los programas tengan bugs. Después de escribir un fragmento de código por primera vez, es conveniente correrlo algunas veces usando tests que permitan poner en evidencia esos bugs.

Diseñar un conjunto de *tests* adecuado no es una tarea sencilla y es frecuente que queden casos especiales que causen errores inesperados.

Python es un lenguaje interpretado, con tipos de datos dinámicos (una misma variable puede cambiar de tipo, de int a float, por ejemplo). No tiene un compilador que te alerte sobre inconsistencias de tipos antes de ejecutar el programa. Es bueno usar *buenas prácticas* que minimicen estos potenciales errores pero igual es posible que algunos errores se filtren.

Testear consiste en ejecutar un programa o porción de código en condiciones controladas, con entradas conocidas y salidas predichas de forma de poder verificar si lo que da el algoritmos es lo que esperabas.

La ejecución de un algoritmo puede pensarse como un árbol (el árbol de ejecución del algoritmo, cada condición booleana da lugar a una ramificación del árbol). Según la entrada que le des, el programa se va a ejecutar siguiendo una rama u otra. Lo ideal es testear todas las ramas posibles de ejecución y que los casos de prueba (*test cases*) incluyan todos los casos *especiales* (casos como listas vacías, índices apuntando al primer o al último elemento, claves ausentes, etc.) comprobando en cada caso que el programa se comporte según lo esperado.

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/04_Listas_y_Listas/spyder-partes.png)

Los entornos de desarrollo integrado (como el Spyder) dan la posiblidad de combinar el uso de un intérprete de Python con un editor de código y suelen integrar también el uso del debugger. Aún con herramientas como el Spyder, hacer debugging es lento y tedioso. Antes de entrar en los detalles de cómo hacerlo, comentaremos algunos métodos que tratan de reducir su necesidad. Profundizaremos sobre estos métodos más adelante.

**Aseveraciones (assert)**

El comando assert se usa para un control interno del programa. Si la expresión que queremos verificar es False, se levanta una excepción de tipo AssertionError. La sintaxis de assert es la siguiente.

assert <expresion> [, 'Mensaje']

Por ejemplo

assert isinstance(10, int), 'Necesito un entero (int)'

La idea *no es* usarlo para comprobar la validez de lo ingresado por el usuario. El propósito de usar assert es verificar que ciertas condiciones se cumplan. En general se lo usa mientras el programa está en desarrollo, y luego se los quita o desactiva cuando el programa funciona.

**Programación por contratos**

Se llama programación por contratos a una forma de programar en la que le programadore define, para cada parte del programa, el tipo y formato de datos con que llamarla y el tipo de datos que devolverá.

Para asegurarse que los tipos de datos sean los esperados, el uso irrestricto de verificaciones puede ayudar en el diseño de software, y detecta tempranamente un error en los datos pasados a una función evitando que se propague.

Por ejemplo: podrías poner verificaciones para cada parámetro de una función.

def add(x, y):

assert isinstance(x, int), 'Necesito un entero (int)'

assert isinstance(y, int), 'Necesito un entero (int)'

return x + y

De este modo, una funcion puede verificar que todos sus argumentos sean válidos.

>>> add(2, 3)

5

>>> add('2', '3')

Traceback (most recent call last):

...

AssertionError: Necesito un entero (int)

>>>

**El debugger de Python (pdb)**

Es posible usar el debugger de Python directamente en el intérprete (sin interfaz gráfica) para seguir el funcionamiento de un programa. No vamos a entrar en esos detalles acá. Solo mencionamos que la función breakpoint () inicia el debugger:

def mi\_funcion():

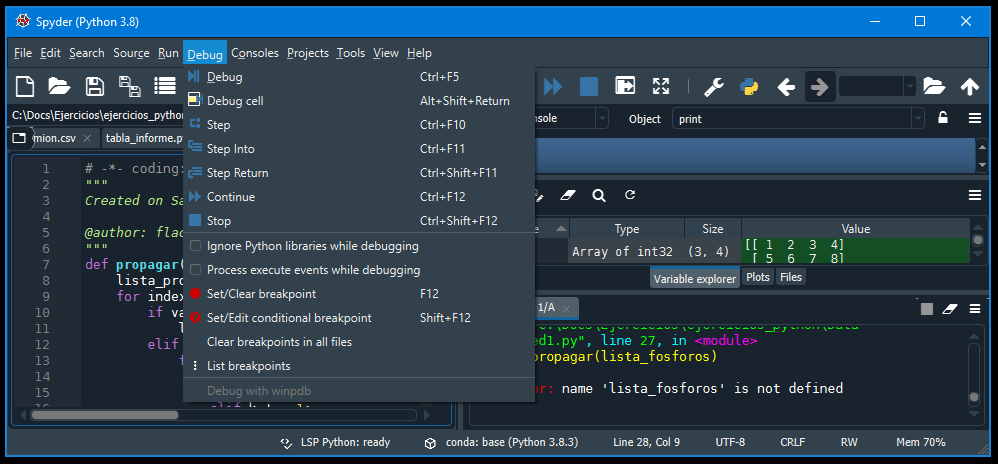
...

breakpoint() # Iniciar el debugger (Python 3.7+)

...

Podés encontrar instrucciones detalladas [acá](https://docs.python.org/3/library/pdb.html) sobre como usarlo.

Nos resulta más cómodo usar un IDE como Spyder para hacer debugging y ése es el método que describiremos aquí. Este es el menú desplegable del debugger:

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/04_Listas_y_Listas/debug_menu.png)

Fijate los nombres de cada ícono:

| **Nombre** | **Acción** |
| --- | --- |
| Debug | inicia el modo debug |
| Step | da un paso en el programa |
| Step Into | entra en la función referida |
| Step Return | ejecuta hasta salir de la función |
| Continue | retoma la ejecución normal |
| Stop | detiene el programa |

Vamos a volver a analizar el siguiente código, similar al del [Ejercicio 3.1](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/03_Datos/01_Bugs.md#ejercicio-31-sem%C3%A1ntica) para que veas la utilidad del debugger:

def tiene\_a(expresion):

n = len(expresion)

i = 0

while i<n:

if expresion[i] == 'a':

return True

else:

return False

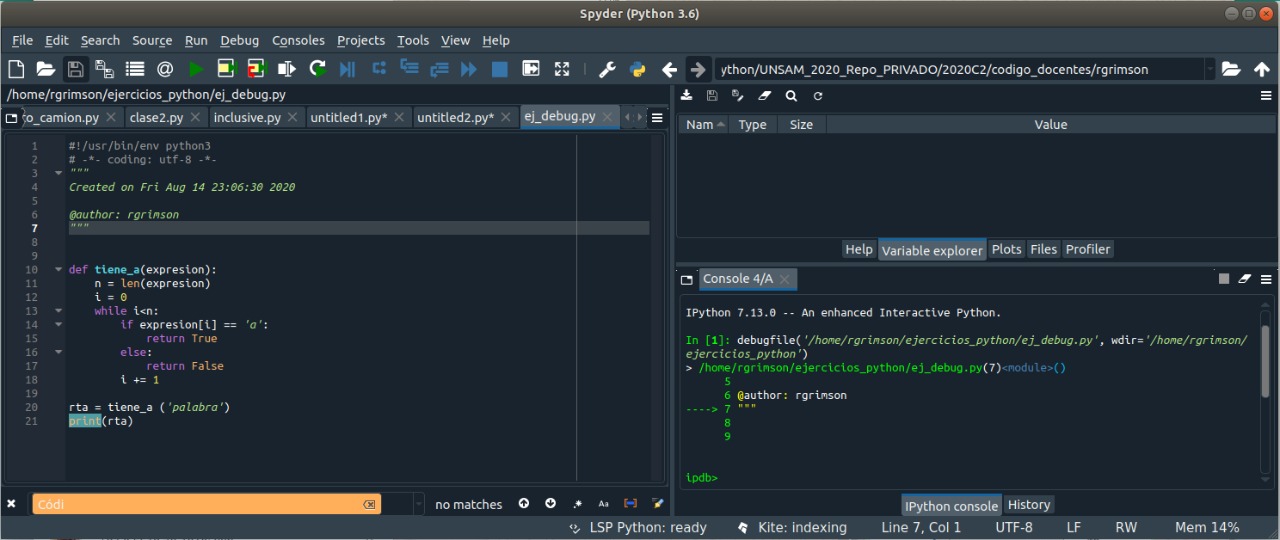
i += 1

rta = tiene\_a ('palabra')

print(rta)

Una vez que tengas el código copiado en el Spyder, vamos a ejecutarlo en *modo debug*:

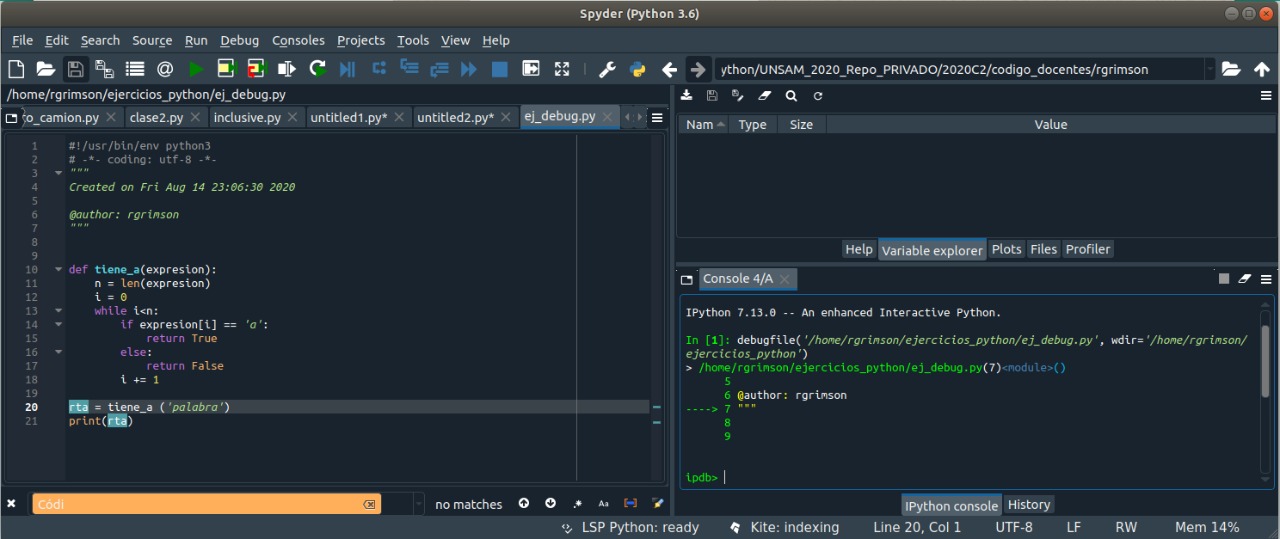
Primero entramos al *modo debug* (Ctrl+F5): El programa queda pausado antes de comenzar. Notá los cambios en la ventana interactiva.

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/04_Listas_y_Listas/debug2.jpg)

Si damos un paso en el programa: ¿qué va a ocurrir? Debemos tratar de responder esta pregunta antes de avanzar cada paso. *Es nuestra predicción, contrastada con lo que realmente sucede, lo que delata el error*.

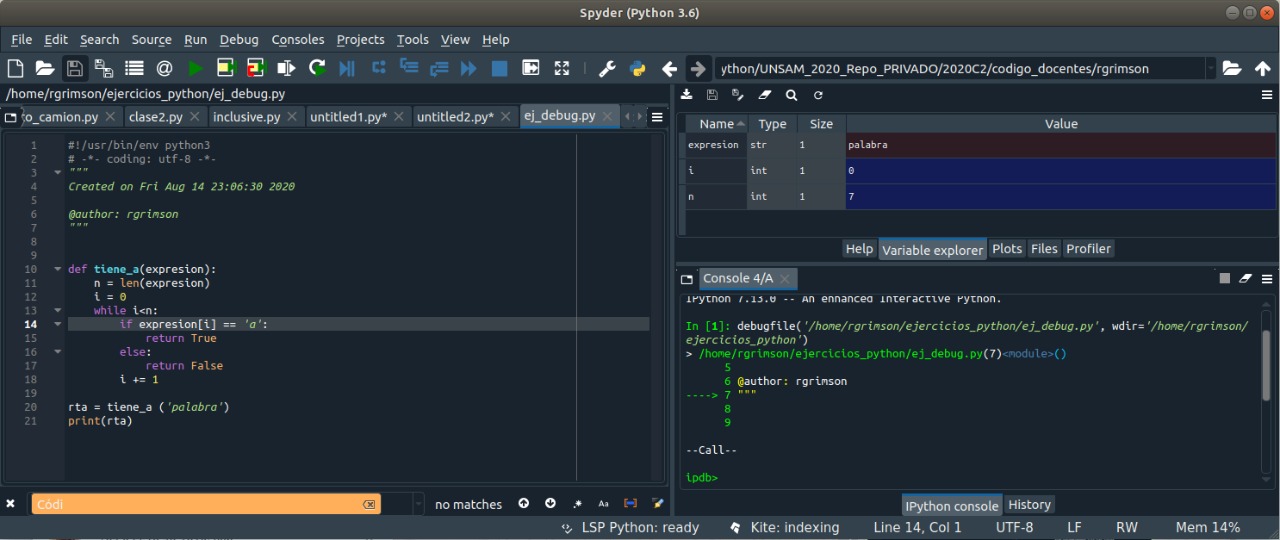
Queremos ver la evolución de las variables en la solapa *Variable Explorer* (solapa del centro en el panel superior de la derecha). El programa está en ejecución pero pausado. Sabemos que estamos en *modo debug* por el prompt ipdb> abajo.

Damos algunos pasos (con Step, Ctrl + F10) hasta llegar a la llamada a la función tiene\_a() que queremos analizar.

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/04_Listas_y_Listas/debug3.jpg)

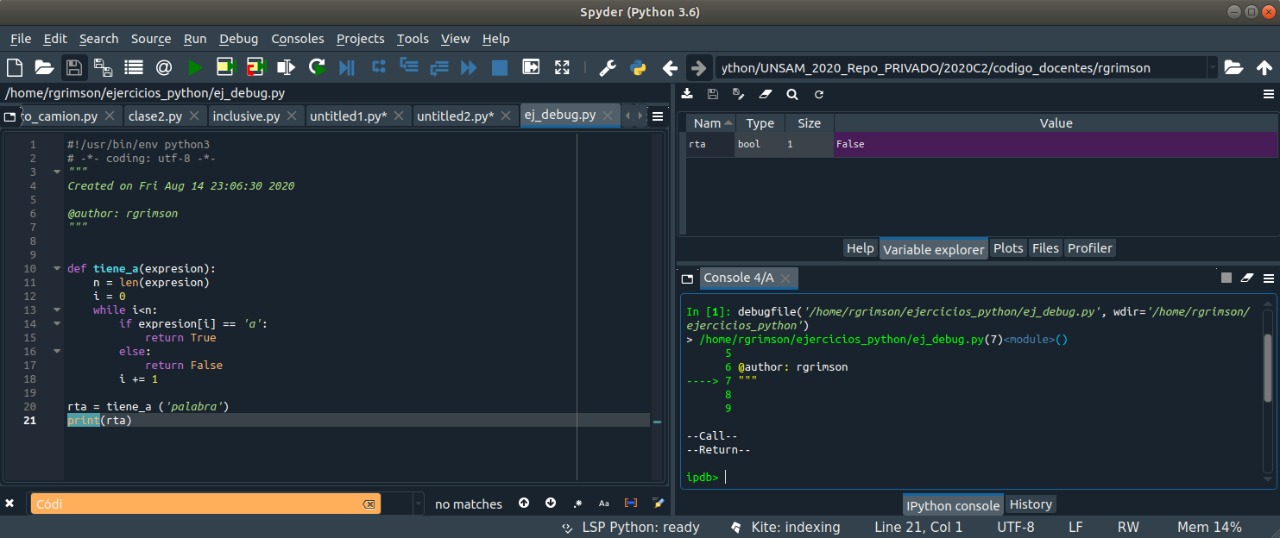
Fijate que el debugger pasó por la línea de definición de la función (y ahora sabe dónde ir a buscarla) pero nunca entró al cuerpo de la función aún. Eso va a ocurrir recién al llamarla.

A esta altura, no queremos simplemente dar un paso (eso ejecutaría la función entera, de una) sino entrar en los detalles de esta función. Para eso usamos Step Into (Ctrl + F11) de forma de entrar en la ejecución de la función tiene\_a(). Una vez dentro, seguimos dando pasos (con Step, Ctrl + F10), siempre pensando qué esperamos que haga la función y observando la evolución de las variables en el explorador de variables. Sigamos así hasta llegar al condicional if. Vemos en el *Variable Explorer* que todas las variables internas de la función están definidas y con sus valores asignados.

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/04_Listas_y_Listas/debug4.jpg)

Como i = 0 sabemos que es la primera iteración. Corroboramos que n=7 (“palabra” tiene 7 letras). En este punto se evalúa if palabra[i] == 'a':, y saltaremos a alguna de las dos ramas de ejecución según la evaluación resulte True o False.

La expresión resulta False ya que la primera letra de 'palabra' es la 'p' y no una 'a'. Pero entonces la siguiente instrucción será return False con lo que saldremos de la función habiendo sólo evaluado la primera letra de la palabra pasada como parámetro. ¿Esto es lo que queríamos?

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/04_Listas_y_Listas/debug5.jpg)

Acabamos de volver de la función. Las variables internas a la función ya no están visibles (salimos de su alcance o *scope*). El programa sigue en ejecución, en *modo debug*.

Si seguimos dando pasos con Step (Ctrl + F10) vamos a pasar por el print() y terminar la ejecución del programa, saliendo del *modo debug*.

Si, en cambio, al llegar a la línea del print() en lugar de Step (Ctrl + F10) avanzáramos con un Step Into (Ctrl + F11), entraríamos en los detalles de la definición de esta función y la cosa se pondría un toque técnica. Cuando esto ocurre es útil usar el Step Return (Ctrl + Shift + F11) para salir de tanto nivel de detalle.

En todo caso, lo que observamos en esta ejecución de tiene\_a() es que salimos de la función después de haber analizado sólo la primera letra de la palabra. ¿Es correcto esto? ¿Donde está el error? ¿Cómo lo podemos resolver?

**Comentario.** Recorrer la ejecución de un programa como un simple espectador no nos muestra claramente un error en el código. Es la incongruencia entre lo esperado y lo que realmente sucede lo que lo marca. Esto exige mucha atención para, antes de ejecutar cada paso, preguntarse: ¿qué espero que ocurra? Luego, al avanzar un paso en la ejecución, puede ocurrir que lo que esperamos que pase no sea lo que realmente pasa. Entonces estamos en un **paso clave** de la ejecución, que nos marca que estamos frente a una de dos: ó frente a un error en el código ó frente a la oportunidad de mejorar nuestra comprensión del mismo.

**Ejercicios**

**Ejercicio 4.1: Debugger**

Ingresá y corré el siguiente código en tu IDE:

def invertir\_lista(lista):

'''Recibe una lista L y la develve invertida.'''

invertida = []

i = len(lista)

while i > 0: # tomo el último elemento

i = i-1

invertida.append (lista.pop(i)) #

return invertida

l = [1, 2, 3, 4, 5]

m = invertir\_lista(l)

print(f'Entrada {l}, Salida: {m}')

Deberías observar que la función modifica el valor de la lista de entrada. Eso no debería ocurrir: una función nunca debería modificar los parámetros salvo que sea lo esperado. Usá el debugger y el explorador de variables para determinar cuál es el primer **paso clave** en el que se modifica el valor de esta variable.

**Ejercicio 4.2: Más debugger**

Siguiendo con los ejemplos del [Ejercicio 3.1](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/03_Datos/01_Bugs.md#ejercicio-31-sem%C3%A1ntica), usá el debugger para analizar el siguiente código:

import csv

from pprint import pprint

def leer\_camion(nombre\_archivo):

camion = []

registro = {}

with open(nombre\_archivo,"rt") as f:

filas = csv.reader(f)

encabezado = next(filas)

for fila in filas:

registro[encabezado[0]] = fila[0]

registro[encabezado[1]] = int(fila[1])

registro[encabezado[2]] = float(fila[2])

camion.append(registro)

return camion

camion = leer\_camion('../Data/camion.csv')

pprint(camion)

Observá en particular lo que ocurre al leer la segunda fila de datos del archivo y guardarlos en la variable registro con los datos ya guardados en la lista camion.

**Más ejercitacion con el debugger?**

**4.2 Listas y búsqueda lineal**

En esta sección seguiremos usando Python, pero nos concentraremos en la parte algorítmica. Vas a escribir funciones sencillas (y no tanto) que realicen operaciones de bajo nivel sobre listas.

Éste es un curso de Python y de algoritmos. Python es un lenguaje de alto nivel. Esto significa que con pocas instrucciones permite realizar operaciones muy complejas. Los lenguajes de bajo nivel están más cerca del lenguaje del procesador y programar en ellos por ejemplo, un análisis de datos, es mucho más tedioso.

Sin embargo, entre las cosas que trae resueltas Python hay algunos algoritmos que nos interesa que vuelvas a escribir vos, por motivos didácticos. En lo que sigue te vamos a pedir en algunas ocasiones que no uses toda la potencia y simpleza de Python sino que te arremangues y escribas algunas funciones desde los primeros rudimentos.

Queremos mostrarte en ejemplos concretos cómo distintas maneras de resolver un mismo problema pueden dar lugar a algoritmos con eficiencias muy diferentes. A veces una es mejor para un uso y la otra para otro uso. En concreto, vamos a profundizar en el problema de la búsqueda y en el problema del ordenamiento, que son dos problemas elementales que ilustran conceptos centrales del desarrollo de algoritmos.

El uso adecuado de estos conceptos puede hacer la diferencia entre un algoritmo que termina el procesamiento en unos pocos minutos o uno que hay que dejar corriendo dos días (y rezar para que no se corte la electricidad mientras corre).

**Búsqueda lineal**

**El problema de la búsqueda**

Presentamos ahora uno de los problemas más clásicos de la computación: **el problema de la búsqueda**. El mismo se puede enunciar de la siguiente manera:

**Problema:** Dada una lista lista y un elemento e devolver el índice de e en lista si e está en lista, y devolver -1 si e no está en lista.

Este problema tiene una solución muy sencilla en Python: se puede usar el método index() de las listas.

Probá esta solución:

>>> [1, 3, 5, 7].index(5)

2

>>> [1, 3, 5, 7].index(20)

Traceback (most recent call last):

File "<ipython-input-177-1bcce50c5c91>", line 1, in <module>

[1, 3, 5, 7].index(20)

ValueError: 20 is not in list

Vemos que usar la función index() resuelve nuestro problema si el valor buscado está en la lista, pero si el valor no está no sólo no devuelve un -1, sino que se produce un error.

El problema es que para poder aplicar la función index() debemos estar seguros de que el valor está en la lista, y para averiguar eso Python nos provee del operador in:

>>> 5 in [1, 3, 5, 7]

True

>>> 20 in [1, 3, 5, 7]

False

Si llamamos a la función index() sólo cuando el resultado de in es verdadero, y devolvemos -1 cuando el resultado de in es falso, estaremos resolviendo el problema planteado usando sólo funciones provistas por Python:

def busqueda\_con\_index(lista, e):

'''Busca un elemento e en la lista.

Si e está en lista devuelve el índice,

de lo contrario devuelve -1.

'''

if e in lista:

pos = lista.index(e)

else:

pos = -1

return pos

Probemos la función busqueda\_con\_index():

>>> busqueda\_con\_index([1, 4, 54, 3, 0, -1], 1)

0

>>> busqueda\_con\_index([1, 4, 54, 3, 0, -1], -1)

5

>>> busqueda\_con\_index([1, 4, 54, 3, 0, -1], 3)

3

>>> busqueda\_con\_index([1, 4, 54, 3, 0, -1], 44)

-1

>>> busqueda\_con\_index([], 0)

-1

**¿Cuántas comparaciones hace este programa?**

Es decir, ¿cuánto esfuerzo computacional requiere este programa? ¿Cuántas veces compara el valor que buscamos con los datos de la lista? No lo sabemos porque no sabemos cómo están implementadas las operaciones in e index(). La pregunta queda planteada por ahora pero daremos un método para averiguarlo más adelante.

**Búsqueda lineal**

Nos interesa estudiar formas alternativas de programar la búsqueda usando operaciones más elementales, y no las primitivas in e index() de nuestro lenguaje de alto nivel. Aceptemos entonces que no vamos a usar ni in ni index(). En cambio, podemos iterar sobre los índices y elementos de una lista para hacer comparaciones elementales.

Consideremos la siguiente solución: iterar sobre los índices y elementos de una lista de manera de comparar el elemento e buscado con cada uno de los elementos de la lista y devolver la posición donde lo encontremos, en caso de encontrarlo. Si llegamos al final de la lista sin haber salido antes de la función es porque el valor de e no está en la lista, y en ese caso devolvemos -1.

En esta solución lo ideal es usar enumerate (ver la [Sección 3.2](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/03_Datos/02_Secuencias.md#la-funci%C3%B3n-enumerate)) ya que dentro de la iteración necesitamos tener acceso tanto al valor del elemento (para ver si es igual al buscado) como a su índice (es el valor que tenemos que devolver).

Primero hagámoslo sin usarlo y luego lo agregamos para entender su ventaja. En ambos casos necesitamos una variable i que cuente en cada momento en qué posición de la lista estamos. Si no usamos enumerate, debemos inicializar i en 0 antes de entrar en el ciclo e incrementarla en 1 en cada paso.

El programa nos queda así:

def busqueda\_lineal(lista, e):

'''Si e está en la lista devuelve su posición, de lo

contrario devuelve -1.

'''

pos = -1 # comenzamos suponiendo que e no está

i = 0

for z in lista: # recorremos los elementos de la lista

if z == e: # si encontramos a e

pos = i # guardamos su posición

break # y salimos del ciclo

i += 1

return pos

La versión con enumerate es mucho más elegante:

def busqueda\_lineal(lista, e):

'''Si e está en la lista devuelve su posición, de lo

contrario devuelve -1.

'''

pos = -1 # comenzamos suponiendo que e no está

for i, z in enumerate(lista): # recorremos la lista

if z == e: # si encontramos a e

pos = i # guardamos su posición

break # y salimos del ciclo

return pos

Y ahora lo probamos:

>>> busqueda\_lineal([1, 4, 54, 3, 0, -1], 44)

-1

>>> busqueda\_lineal([1, 4, 54, 3, 0, -1], 3)

3

>>> busqueda\_lineal([1, 4, 54, 3, 0, -1], 0)

4

>>> busqueda\_lineal([], 42)

-1

**¿Cuántas comparaciones hace este programa?**

Volvemos a preguntarnos lo mismo que en la sección anterior pero con el nuevo programa: ¿cuánto esfuerzo computacional requiere este programa?, ¿cuántas veces compara el valor que buscamos con los datos de la lista? Ahora podemos analizar el código de busqueda\_lineal:

El ciclo recorre uno a uno los elementos de la lista, y en el cuerpo de ese ciclo, se compara cada elemento con el valor buscado. En el caso de encontrarlo se devuelve la posición. Si el valor no está en la lista, se recorrerá la lista entera, haciendo una comparación por cada elemento.

O sea que si el valor está en la posición *p* de la lista se hacen *p* comparaciones. En el *peor caso*, si el valor no está, se hacen tantas comparaciones como elementos tenga la lista.

En resumen: Si la lista crece, la cantidad de comparaciones para encontrar un valor arbitrario crecerá en forma proporcional al tamaño de la lista. Es decir que:

**El algoritmo de búsqueda lineal tiene un comportamiento *proporcional a la longitud de la lista involucrada*, o que es un algoritmo *lineal*.**

**Ejercicios**

**Ejercicio 4.3: Búsquedas de un elemento**

Creá el archivo busqueda\_en\_listas.py para guardar tu código de este ejercicio y el siguiente.

En este primer ejercicio tenés que escribir una función buscar\_u\_elemento() que reciba una lista y un elemento y devuelva la posición de la última aparición de ese elemento en la lista (o -1 si el elemento no pertenece a la lista).

Probá tu función con algunos ejemplos:

>>> buscar\_u\_elemento([1,2,3,2,3,4],1)

0

>>> buscar\_u\_elemento([1,2,3,2,3,4],2)

3

>>> buscar\_u\_elemento([1,2,3,2,3,4],3)

4

>>> buscar\_u\_elemento([1,2,3,2,3,4],5)

-1

Agregale a tu programa busqueda\_en\_listas.py una función buscar\_n\_elemento() que reciba una lista y un elemento y devuelva la cantidad de veces que aparece el elemento en la lista. Probá también esta función con algunos ejemplos.

**Ejercicio 4.4: Búsqueda de máximo y mínimo**

Agregale a tu archivo busqueda\_en\_listas.py una función maximo() que busque el valor máximo de una lista de números positivos. Python tiene el comando max que ya hace esto, pero como práctica te proponemos que completes el siguiente código:

def maximo(lista):

'''Devuelve el máximo de una lista,

la lista debe ser no vacía y de números positivos.

'''

# m guarda el máximo de los elementos a medida que recorro la lista.

m = 0 # Lo inicializo en 0

for e in lista: # Recorro la lista y voy guardando el mayor

...

return m

Probá tu función con estos ejemplos:

>>> maximo([1,2,7,2,3,4])

7

>>> maximo([1,2,3,4])

4

>>> maximo([-5,4])

4

>>> maximo([-5,-4])

0

¿Por qué falla en el último caso? ¿Por qué anda en el caso anterior? ¿Cómo se puede inicializar m para que la función ande también con números negativos? Corregilo y guarda la versión mejorada en el archivo busqueda\_en\_listas.py.

Si te dan ganas, agregá una función minimo() al archivo.

**Ejercitación con iteradores y listas**

**Ejercicio 4.5: Invertir una lista**

Escribí una función invertir\_lista(lista) que dada una lista devuelva otra que tenga los mismos elementos pero en el orden inverso. Es decir, el que era el primer elemento de la lista de entrada deberá ser el último de la lista de salida y análogamente con los demás elementos.

def invertir\_lista(lista):

invertida = []

for e in lista: # Recorro la lista

... #agrego el elemento e al principio de la lista invertida

return invertida

Guardá la función en el archivo invlista.py y probala con las siguientes listas:

1. [1, 2, 3, 4, 5]
2. ['Bogotá', 'Rosario', 'Santiago', 'San Fernando', 'San Miguel']

**Ejercicio 4.6: Propagación**

Imaginate una fila con varios fósforos uno al lado del otro. Los fósforos pueden estar en tres estados: nuevos, prendidos fuego o ya gastados (carbonizados). Representaremos esta situación con una lista *L* con un elemento por fósforo, que en cada posición tiene un 0 (nuevo), un 1 (encendido) o un -1 (carbonizado). El fuego se propaga inmediatamente de un fósforo encendido a cualquier fósforo nuevo que tenga a su lado. Los fósforos carbonizados no se encienden nuevamente.

Escribí una función llamada propagar que reciba un vector con 0's, 1's y -1's y devuelva un vector en el que los 1's se propagaron a sus vecinos con 0. Guardalo en un archivo propaga.py.

Por ejemplo:

>>> propagar([ 0, 0, 0,-1, 1, 0, 0, 0,-1, 0, 1, 0, 0])

[ 0, 0, 0,-1, 1, 1, 1, 1,-1, 1, 1, 1, 1]

>>> propagar([ 0, 0, 0, 1, 0, 0])

[ 1, 1, 1, 1, 1, 1]

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/04_Listas_y_Listas/fosforos.jpg) Propagación análoga a la del [Ejercicio 4.6](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/04_Listas_y_Listas/02_IteradoresLista.md#ejercicio-46-propagaci%C3%B3n)

**4.3 Comprensión de listas**

Una tarea que realizamos una y otra vez es procesar los elementos de una lista. En esta sección introducimos la definición de listas por comprensión que es una herramienta potente para hacer exactamente eso.

**Crear listas nuevas**

La comprensión de listas crea un una nueva lista aplicando una operación a cada elemento de una secuencia.

>>> a = [1, 2, 3, 4, 5]

>>> b = [2\*x for x in a]

>>> b

[2, 4, 6, 8, 10]

>>>

Otro ejemplo:

>>> nombres = ['Edmundo', 'Juana']

>>> a = [nombre.lower() for nombre in nombres]

>>> a

['edmundo', 'juana']

>>>

La sintaxis general es : [<expresión> for <variable> in <secuencia>].

**Filtros**

La comprensión de listas se puede usar para filtrar.

>>> a = [1, -5, 4, 2, -2, 10]

>>> b = [2\*x for x in a if x > 0]

>>> b

[2, 8, 4, 20]

>>>

**Casos de uso**

La comprensión de listas es enormemente útil. Por ejemplo, podés recolectar los valores de un campo específico de un diccionario:

frutas = [s['nombre'] for s in camion]

O podés hacer consultas (*queries*) como si las secuencias fueran bases de datos.

a = [s for s in camion if s['precio'] > 100 and s['cajones'] > 50]

También podés combinar la comprensión de listas con reducciones de secuencias:

costo = sum([s['cajones']\*s['precio'] for s in camion])

**Sintaxis general**

[<expresión> for <variable> in <secuencia> if <condición>]

Lo que significa

resultado = []

for variable in secuencia:

if condición:

resultado.append(expresión)

**Digresión histórica**

La comprensión de listas viene de la matemática (definición de conjuntos por comprensión).

a = [x \* x for x in s if x > 0] # Python

a = {x^2 | x ∈ s, x > 0} # Matemática

La mayoría de los programadores no suelen pensar en el costado matemático de esta herramienta. Podemos verla simplemente como una abreviación copada para definir listas.

**Ejercicios**

Corré tu programa informe.py de forma de tener los datos sobre cajones cargados en tu intérprete en modo interactivo.

Luego, tratá de escribir los comandos adecuados para realizar las operaciones descriptas abajo. Estas operaciones son reducciones, transformaciones y consultas sobre la carga del camión.

**Ejercicio 4.7: Comprensión de listas**

Probá un par de comprensión de listas para familiarizarte con la sintaxis.

>>> nums = [1,2,3,4]

>>> cuadrados = [x \* x for x in nums]

>>> cuadrados

[1, 4, 9, 16]

>>> dobles = [2 \* x for x in nums if x > 2]

>>> dobles

[6, 8]

>>>

Observá que estás creando nuevas listas con los datos adecuadamente transformados o filtrados.

**Ejercicio 4.8: Reducción de secuencias**

Calculá el costo total de la carga del camión en un solo comando.

>>> camion = leer\_camion('../Data/camion.csv')

>>> costo = sum([s['cajones'] \* s['precio'] for s in camion])

>>> costo

47671.15

>>>

Luego, leyendo la variable precios, calculá también el valor en el mercado de la carga del camión usando una sola línea de código.

>>> precios = leer\_precios('../Data/precios.csv')

>>> valor = sum([s['cajones'] \* precios[s['nombre']] for s in camion])

>>> valor

62986.1

>>>

Ambos son ejemplos de aplicación-reducción. La comprensión de listas está aplicando una operación a lo largo de la lista.

>>> [s['cajones'] \* s['precio'] for s in camion]

[3220.0000000000005, 4555.0, 15516.0, 10246.0, 3835.1499999999996, 3254.9999999999995, 7044.0]

>>>

La función sum() luego realiza una reducción del resultado

>>> sum(\_)

47671.15

>>>

Con este conocimiento algunos ya empiezan su startup de big-data.

**Ejercicio 4.9: Consultas de datos**

Probá los siguientes ejemplos de consultas (queries) de datos.

Primero, generá una lista con la info de todas las frutas que tienen más de 100 cajones en el camión.

>>> mas100 = [s for s in camion if s['cajones'] > 100]

>>> mas100

[{'cajones': 150, 'nombre': 'Caqui', 'precio': 103.44},

{'cajones': 200, 'nombre': 'Mandarina', 'precio': 51.23}]

>>>

Ahora, una con la info sobre cajones de Mandarina y Naranja.

>>> myn = [s for s in camion if s['nombre'] in {'Mandarina','Naranja'}]

>>> myn

[{'cajones': 50, 'nombre': 'Naranja', 'precio': 91.1},

{'cajones': 200, 'nombre': 'Mandarina', 'precio': 51.23},

{'cajones': 50, 'nombre': 'Mandarina', 'precio': 65.1},

{'cajones': 100, 'nombre': 'Naranja', 'precio': 70.44}]

>>>

O una con la info de las frutas que costaron más de $10000.

>>> costo10k = [s for s in camion if s['cajones'] \* s['precio'] > 10000]

>>> costo10k

[{'cajones': 150, 'nombre': 'Caqui', 'precio': 103.44},

{'cajones': 200, 'nombre': 'Mandarina', 'precio': 51.23}]

>>>

Esta forma de escribir resulta análoga a las consultas a una base de datos con SQL.

**Ejercicio 4.10: Extracción de datos**

Usando un comprensión de listas, construí una lista de tuplas (nombre, cajones) que indiquen la cantidad de cajones de cada fruta tomando los datos de camion.

>>> nombre\_cajones =[(s['nombre'], s['cajones']) for s in camion]

>>> nombre\_cajones

[('Lima', 100), ('Naranja', 50), ('Caqui', 150), ('Mandarina', 200), ('Durazno', 95), ('Mandarina', 50), ('Naranja', 100)]

>>>

Si cambiás los corchetes ([,]) por llaves ({, }), obtenés algo que se conoce como comprensión de conjuntos. Vas a obtener valores únicos.

Por ejemplo, si quisieras un listado de las frutas en el camión pordías usar:

>>> nombres = {s['nombre'] for s in camion}

>>> nombres

{'Caqui', 'Durazno', 'Lima', 'Mandarina', 'Naranja'}

>>>

Si especificás pares clave:valor, podés construir un diccionario. Por ejemplo, si queremos un diccionario con el total de cada fruta en el camión podemos comenzar con

>>> stock = {nombre: 0 for nombre in nombres}

>>> stock

{'Caqui': 0, 'Durazno': 0, 'Lima': 0, 'Mandarina': 0, 'Naranja': 0}

>>>

que es una comprensión de diccionario. Y seguir sumando los cajones:

>>> for s in camion:

stock[s['nombre']] += s['cajones']

>>> stock

{'Caqui': 150, 'Durazno': 95, 'Lima': 100, 'Mandarina': 250, 'Naranja': 150}

>>>

Otro ejemplo útil podría ser generar un diccionario de precios de venta de aquellos productos que están efectivamente cargados en el camión:

>>> camion\_precios = {nombre: precios[nombre] for nombre in nombres}

>>> camion\_precios

{'Caqui': 105.46, 'Durazno': 73.48, 'Lima': 40.22, 'Mandarina': 80.89, 'Naranja': 106.28}

>>>

**Ejercicio 4.11: Extraer datos de un archivo CSV**

Saber usar combinaciones de comprensión de listas, diccionarios y conjuntos resulta útil para procesar datos en diferentes contextos. Aunque puede volverse medio críptico si no estás habituade. Acá te mostramos un ejemplo de cómo extraer columnas seleccionadas de un archivo CSV que tiene esas características. No es dificil cuando lo entendés, pero está muy concentrado todo.

Primero, leamos el encabezado (header) del archivo CSV:

>>> import csv

>>> f = open('../Data/fecha\_camion.csv')

>>> rows = csv.reader(f)

>>> headers = next(rows)

>>> headers

['nombre', 'fecha', 'hora', 'cajones', 'precio']

>>>

Luego, definamos una lista que tenga las columnas que nos importan:

>>> select = ['nombre', 'cajones', 'precio']

>>>

Ubiquemos los índices de esas columnas en el CSV:

>>> indices = [headers.index(ncolumna) for ncolumna in select]

>>> indices

[0, 3, 4]

>>>

Y finalmente leamos los datos y armemos un diccionario usando comprensión de diccionarios:

>>> row = next(rows)

>>> record = {ncolumna: row[index] for ncolumna, index in zip(select, indices)} # comprensión de diccionario

>>> record

{'precio': '32.20', 'nombre': 'Lima', 'cajones': '100'}

>>>

No es trivial este comando. El comando es sintácticamente muy compacto, pero es conceptualmente (un poco) complejo. Cuando te sientas cómode con esta lectura de una línea del archivo (si no pasa, tranca, podemos seguir sin esto), leé el resto:

>>> camion = [{ ncolumna: row[index] for ncolumna, index in zip(select, indices)} for row in rows]

>>> camion

[{'cajones': '50', 'nombre': 'Naranja', 'precio': '91.1'},

{'cajones': '150', 'nombre': 'Caqui', 'precio': '103.44'},

{'cajones': '200', 'nombre': 'Mandarina', 'precio': '51.23'},

{'cajones': '95', 'nombre': 'Durazno', 'precio': '40.37'},

{'cajones': '50', 'nombre': 'Mandarina', 'precio': '65.1'},

{'cajones': '100', 'nombre': 'Naranja', 'precio': '70.44'}]

>>>

¡Por las barbas de mi abuelo! Acabamos de reducir casi toda la función leer\_camion() a un solo comando.

**Comentario**

La comprensión de listas se usa frecuentemente en Python. Es una forma eficiente de transformar, filtrar o juntar datos. Tiene una sintaxis potente pero tratá de no pasarte con su uso: mantené cada comando tan simple como sea posible. Está perfecto descomponer un solo comando complejo en muchos pasos. Concretamente: compartir el último ejemplo con personas desprevenidas puede no ser lo ideal.

Dicho esto, saber manipular datos rápidamente es una habilidad increíblemente útil. Hay numerosas situaciones donde puede que tengas que resolver algún tipo de problema excepcional (en el sentido de raro o único) para importar, extraer o exportar datos. La comprensión de listas te puede ahorrar muchísimo tiempo en esas tareas.

**4.4 Objetos**

En esta sección introducimos algunos conceptos sobre el modelo interno de objeto en Python y discutimos algunos temas relacionados con el manejo de memoria, copias de variable y verificación de tipos.

**Asignaciones**

Muchas operaciones en Python están relacionadas con *asignar* o *guardar* valores.

a = valor # Asignación a una variable

s[n] = valor # Asignación a una lista

s.append(valor) # Agregar a una lista

d['key'] = valor # Agregar a una diccionario

*Ojo: las operaciones de asignación****nunca hacen una copia****del valor asignado.* Las asignaciones son simplemente copias de las referencias (o copias del puntero, si preferís).

**Ejemplo de asignación**

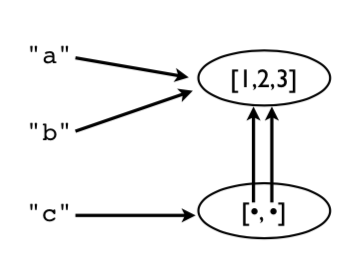
Considerá este fragmento de código.

a = [1,2,3]

b = a

c = [a,b]

A continuación te mostramos en un gráfico las operaciones de memoria suyacentes. En este ejemplo, hay solo un objeto lista [1,2,3], pero hay cuatro referencias a él.

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/04_Listas_y_Listas/referencias.png)

Esto significa que al modificar un valor modificamos *todas* las referencias.

>>> a.append(999)

>>> a

[1,2,3,999]

>>> b

[1,2,3,999]

>>> c

[[1,2,3,999], [1,2,3,999]]

>>>

Observá cómo un cambio en la lista original desencadena cambios en todas las demás variables (ouch!). Esto es porque no se hizo ninguna copia. Todos son punteros a la misma cosa.

Esto es lo mismo que pasaba en el [Ejercicio 3.5](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/03_Datos/01_Bugs.md#ejercicio-35-pisando-memoria).

**Reasignar valores**

La reasignación de valores *nunca* sobreescribe la memoria ocupada por un valor anterior.

a = [1,2,3]

b = a

a = [4,5,6]

print(a) # [4, 5, 6]

print(b) # [1, 2, 3] Mantiene el valor original

Acordate: **Las variables son nombres, no ubicaciones en la memoria.**

**Peligros**

Si no te explican esto, tarde o temprano te trae problemas. Un típico ejemplo es cuando cambiás un dato pensando que es una copia privada y, sin querer, esto corrompe los datos en otra parte del programa.

*Comentario: Esta es una de las razones por las que los tipos de datos primitivos (int, float, string) son immutables (de sólo lectura).*

**Identidad y referencias**

Podés usar el operador is (es) para verificar si dos valores corresponden al mismo objeto.

>>> a = [1,2,3]

>>> b = a

>>> a is b

True

>>>

is compara la identidad del objeto (que está representada por un número entero). Esta identidad también la podés obtener usando id().

>>> id(a)

3588944

>>> id(b)

3588944

>>>

Observación: Para ver si dos valores son iguales, es mejor usar el ==. El comportamiento de is puede dar resultados inesperados:

>>> a = [1,2,3]

>>> b = a

>>> c = [1,2,3]

>>> a is b

True

>>> a is c

False

>>> a == c

True

>>>

**Copias superficiales**

Las listas y diccionarios tienen métodos para hacer copias (no meras referencias, sino duplicados):

>>> a = [2,3,[100,101],4]

>>> b = list(a) # Hacer una copia

>>> a is b

False

Ahora b es una nueva lista.

>>> a.append(5)

>>> a

[2, 3, [100, 101], 4, 5]

>>> b

[2, 3, [100, 101], 4]

A pesar de esto, los elementos de a y de b siguen siendo compartidos.

>>> a[2].append(102)

>>> b[2]

[100,101,102]

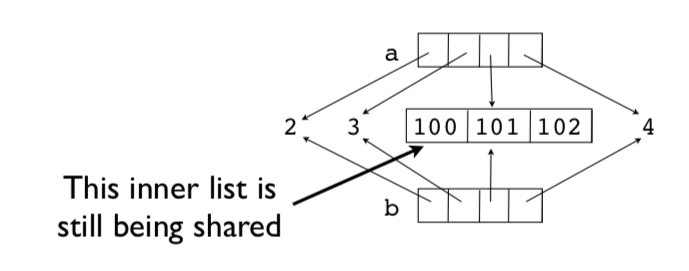
>>>

>>> a[2] is b[2]

True

>>>

En este ejemplo, la lista interna [100, 101, 102] es compartida por ambas variables. La copia que hicimos con el comando b = list(a) es un *copia superficial* (superficial en el sentido de *poco profunda*, en inglés se dice *shallow copy*). Mirá este gráfico.

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/04_Listas_y_Listas/shallow.png)

La lista interna sigue siendo compartida.

**Copias profundas**

A veces vas a necesitar hacer una copia de un objeto así como de todos los objetos que contenga. Llamamos a esto una *copia pofunda* (*deep copy*). Podés usar la función deepcopy del módulo copy para esto:

>>> a = [2,3,[100,101],4]

>>> import copy

>>> b = copy.deepcopy(a)

>>> a[2].append(102)

>>> b[2]

[100,101]

>>> a[2] is b[2]

False

>>>

**Nombre, valores y tipos**

Los nombres de variables no tienen un tipo asociado. Sólo son nombres. Pero los valores sí tienen un tipo subyacente.

>>> a = 42

>>> b = 'Hello World'

>>> type(a)

<type 'int'>

>>> type(b)

<type 'str'>

type() te dice el tipo del valor.

**Verificación de tipos**

Podés verificar si un objeto es una instancia de cierto tipo.

if isinstance(a, list):

print('a es una lista')

O incluso si su tipo está entre varios tipos.

if isinstance(a, (list,tuple)):

print('a una lista o una tupla')

*Cuidado: Demasiadas verificaciones de tipos pueden resultar en un código excesivamente complejo. Típicamente lo usás para evitar errores comunes cometidos por otres usuaries de tu código.*

**Todo es un objeto**

Números, cadenas, listas, funciones, excepciones, clases, instancias, etc. son todos objetos. Esto significa que pueden ser nombrados, pueden ser pasados como datos, ubicados en contenedores, etc. sin restricciones. No hay objetos especiales en Python. Todos los objetos viajan en primera clase.

Un ejemplo simple:

>>> import math

>>> items = [abs, math, ValueError ]

>>> items

[<built-in function abs>,

<module 'math' (builtin)>,

<type 'exceptions.ValueError'>]

>>> items[0](-45)

45

>>> items[1].sqrt(2)

1.4142135623730951

>>> try:

x = int('not a number')

except items[2]:

print('Failed!')

Failed!

>>>

Acá, items es una lista que tiene una función, un módulo y una excepción. Sí, éste es un ejemplo raro. Pero es un ejemplo al fin. Podés usar los elementos de la lista en lugar de los nombres originales:

items[0](-45) # abs

items[1].sqrt(2) # math

except items[2]: # ValueError

Con un gran poder viene siempre una gran responsabilidad. Que puedas no significa que debas hacer este tipo de cosas.

**Ejercicios**

En estos ejercicios mostramos algo de la potencia que tiene el hecho de que todos los objetos sean de la misma jerarquía.

**Ejercicio 4.12: Datos de primera clase**

En el archivo Data/camion.csv, leímos datos organizados en columnas que se ven así:

nombre,cajones,precio

"Lima",100,32.20

"Naranja",50,91.10

...

En las clases anteriores, usamos el módulo csv para leer el archivo, pero tuvimos que hacer conversiones de tipo. Por ejemplo:

for row in rows:

nombre = row[0]

cajones = int(row[1])

precio = float(row[2])

Este tipo de conversiones puede hacerse de una manera más inteligente usando algunas operaciones de listas.

Hagamos una lista de Python con los nombres de las funciones de conversión que necesitamos para convertir cada columna al tipo apropiado:

>>> types = [str, int, float]

>>>

Podés crear esta lista porque en Python todos los objetos son de la misma clase (de primera clase, digamos). Por lo tanto, si querés tener funciones en una lista, no pasa nada. Los elementos de la lista que creaste son funciones que convierten un valor x a un tipo dado (str(x), int(x), float(x)).

Ahora, leé una fila de datos del archivo anterior:

>>> import csv

>>> f = open('../Data/camion.csv')

>>> rows = csv.reader(f)

>>> headers = next(rows)

>>> row = next(rows)

>>> row

['Lima', '100', '32.20']

>>>

Como ya dijimos, con esta fila no podemos hacer operaciones porque los tipos son incorrectos. Por ejemplo:

>>> row[1] \* row[2]

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: can't multiply sequence by non-int of type 'str'

>>>

Sin embargo, los datos pueden aparearse con los tipos especificados en types. Por ejemplo:

>>> types[1]

<type 'int'>

>>> row[1]

'100'

>>>

Probá convertir uno de los valores:

>>> types[1](row[1]) # Es equivalente a int(row[1])

100

>>>

Probá con otro:

>>> types[2](row[2]) # Equivalente a float(row[2])

32.2

>>>

Probá calcular usando los tipos convertidos:

>>> types[1](row[1])\*types[2](row[2])

3220.0000000000005

>>>

Hagamos un Zip de la lista de tipos con la de datos y veamos el resultado:

>>> r = list(zip(types, row))

>>> r

[(<type 'str'>, 'Lima'), (<type 'int'>, '100'), (<type 'float'>,'32.20')]

>>>

Se puede ver que esto aparea una función de conversión de tipos con un valor. Por ejemplo, int está en un par con el valor '100'.

Esta lista zipeada es útil si querés realizar conversiones de todos los valores. Por ejemplo:

>>> converted = []

>>> for func, val in zip(types, row):

converted.append(func(val))

...

>>> converted

['Lima', 100, 32.2]

>>> converted[1] \* converted[2]

3220.0000000000005

>>>

Asegurate de entender lo que está pasando en el código de arriba. En el ciclo la variable func va tomando los valores de las funciones de conversión de tipos (str, int, float) y la variable val va tomando los valores de los datos en la fila: 'Lima', '100', '32.2'. La expresión func(val) convierte los tipos de cada dato.

El código de arriba puede comprimirse en una sola instrucción usando comprensión de listas.

>>> converted = [func(val) for func, val in zip(types, row)]

>>> converted

['Lima', 100, 32.2]

>>>

**Ejercicio 4.13: Diccionarios**

¿Te acordás que la función dict() te permite hacer fácilmente un diccionario si tenés una secuencia de tuplas con claves y valores? Hagamos un diccionario usando el encabezado de las columnas:

>>> headers

['nombre', 'cajones', 'precio']

>>> converted

['Lima', 100, 32.2]

>>> dict(zip(headers, converted))

{'precio': 32.2, 'nombre': 'Lima', 'cajones': 100}

>>>

Si estás en sintonía con la comprensión de listas podés escribir una sola línea usando comprensión de diccionarios:

>>> { name: func(val) for name, func, val in zip(headers, types, row) }

{'precio': 32.2, 'name': 'Lima', 'cajones': 100}

>>>

**Ejercicio 4.14: Fijando ideas**

Usando las técnicas de este ejercicio, vas a poder escribir instrucciones que conviertan fácilmente campos como los de nuestro archivo en un diccionario de Python.

Para ilustrar esto, supongamos que leés un archivo de datos de la siguiente forma:

>>> f = open('../Data/dowstocks.csv')

>>> rows = csv.reader(f)

>>> headers = next(rows)

>>> row = next(rows)

>>> headers

['name', 'price', 'date', 'time', 'change', 'open', 'high', 'low', 'volume']

>>> row

['AA', '39.48', '6/11/2007', '9:36am', '-0.18', '39.67', '39.69', '39.45', '181800']

>>>

Convirtamos estos datos usando un truco similar:

>>> types = [str, float, str, str, float, float, float, float, int]

>>> converted = [func(val) for func, val in zip(types, row)]

>>> record = dict(zip(headers, converted))

>>> record

{'volume': 181800, 'name': 'AA', 'price': 39.48, 'high': 39.69,

'low': 39.45, 'time': '9:36am', 'date': '6/11/2007', 'open': 39.67,

'change': -0.18}

>>> record['name']

'AA'

>>> record['price']

39.48

>>>

Bonus: ¿Cómo modificarías este ejemplo para transformar la fecha (date) en una tupla como (6, 11, 2007)?

Es importante que entiendas lo que hicimos en este ejercicio. Volveremos sobre esto más adelante.

**4.5 Arbolado porteño y comprensión de listas**

Seguimos aquí con el arbolado porteño. Vamos a plantear algunos ejercicios para hacer con la técnica de comprensión de listas introducida recientemente.

**Ejercicios**

Seguimos trabajando con el archivo CSV de "[Arbolado en espacios verdes](https://data.buenosaires.gob.ar/dataset/arbolado-espacios-verdes)" que ya está en tu carpeta Data. Vamos a estudiar esta base de datos y responder algunas preguntas. Guardá los ejercicios de esta sección en el archivo arboles.py.

**Ejercicio 4.15: Lectura de todos los árboles**

Basándote en la función leer\_parque(nombre\_archivo, parque) del [Ejercicio 3.18](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/03_Datos/06_Arboles1.md#ejercicio-318-lectura-de-los-%C3%A1rboles-de-un-parque), escribí otra leer\_arboles(nombre\_archivo) que lea el archivo indicado y devuelva una **lista de diccionarios** con la información de todos los árboles en el archivo. La función debe devolver una lista conteniendo un diccionario por cada árbol con todos los datos.

Vamos a llamar arboleda a esta lista.

**Ejercicio 4.16: Lista de altos de Jacarandá**

Usando comprensión de listas y la variable arboleda podés por ejemplo armar la lista de la altura de los árboles.

H=[float(arbol['altura\_tot']) for arbol in arboleda]

Usá los filtros (recordá la [Sección 4.3](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/04_Listas_y_Listas/03_Comprension_Listas.md#filtros)) para armar la lista de alturas de los Jacarandás solamente.

**Ejercicio 4.17: Lista de altos y diámetros de Jacarandá**

En el ejercicio anterior usaste una sola linea para seleccionar las alturas de los Jacarandás en parques porteños. Ahora te proponemos que armes una nueva lista que tenga pares (tuplas de longitud 2) conteniendo no solo el alto sino también el diámetro de cada Jacarandá en la lista.

Esperamos que obtengas una lista similar a esta:

[(5.0, 10.0),

(5.0, 10.0),

...

(12.0, 25.0),

...

(7.0, 97.0),

(8.0, 28.0),

(2.0, 30.0),

(3.0, 10.0),

(17.0, 40.0)]

**Ejercicio 4.18: Diccionario con medidas**

En este ejercicio vamos a considerar algunas especies de árboles. Por ejemplo:

especies = ['Eucalipto', 'Palo borracho rosado', 'Jacarandá']

Te pedimos que armes un diccionario en el que estas especies sean las claves y los valores asociados sean los datos que generaste en el ejercicio anterior. Más aún, organizá tu código dentro de una función medidas\_de\_especies(especies,arboleda) que recibe una lista de nombres de especies y una lista como la del [Ejercicio 4.15](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/04_Listas_y_Listas/05_Arboles2_LC.md#ejercicio-415-lectura-de-todos-los-%C3%A1rboles) y devuelve un diccionario cuyas claves son estas especies y sus valores asociados sean las medidas generadas en el ejercicio anterior.

Vamos a usar esta función la semana próxima. A modo de control, si llamás a la función con las tres especies del ejemplo como parámetro (['Eucalipto', 'Palo borracho rosado', 'Jacarandá']) y la arboleda entera, deberías recibir un diccionario con tres entradas (una por especie), cada una con una lista asociada conteniendo 4112, 3150 y 3255 pares de números (altos y diámetros), respectivamente.

Acordate de guardar los ejercicios de esta sección en el archivo arboles.py.

*Extra: casi todes usan un for para crear este diccionario. ¿Lo podés hacer usando una****comprensión de diccionarios****? Te recordamos la sintaxis: diccionario = { clave: valor for clave in claves }*

**4.6 Cierre de la clase**

En esta clase vimos algo más sobre el debugger, el manejo de listas y la creación de listas por comprensión. Para cerrar esta clase te pedimos dos cosas:

* Que recopiles las soluciones de los siguientes ejercicios:

1. El archivo busqueda\_en\_listas.py del [Ejercicio 4.3](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/04_Listas_y_Listas/02_IteradoresLista.md#ejercicio-43-b%C3%BAsquedas-de-un-elemento) y el [Ejercicio 4.4](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/04_Listas_y_Listas/02_IteradoresLista.md#ejercicio-44-b%C3%BAsqueda-de-m%C3%A1ximo-y-m%C3%ADnimo).
2. El archivo invlista.py del [Ejercicio 4.5](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/04_Listas_y_Listas/02_IteradoresLista.md#ejercicio-45-invertir-una-lista).
3. El archivo propaga.py del [Ejercicio 4.6](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/04_Listas_y_Listas/02_IteradoresLista.md#ejercicio-46-propagaci%C3%B3n).
4. El archivo arboles.py sobre arbolado porteño y comprensión de listas incluyendo el [Ejercicio 4.16](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/04_Listas_y_Listas/05_Arboles2_LC.md#ejercicio-416-lista-de-altos-de-jacarand%C3%A1) y el [Ejercicio 4.17](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/04_Listas_y_Listas/05_Arboles2_LC.md#ejercicio-417-lista-de-altos-y-di%C3%A1metros-de-jacarand%C3%A1).

* Que completes [este formulario](https://docs.google.com/forms/d/1I3B43uVprrHh7ZOFk3d5tO29GOW_lsRM1AMXS2CbNSY) usando como identificación tu dirección de mail. Al terminar vas a obtener un link para enviarnos tus ejercicios y podrás participar de la revisión de pares.

¡Gracias! Nos vemos en la próxima clase.