3. Trabajando con datos

En esta clase discutimos los tipos de errores que pueden aparecer en un programa (bugs) e introducimos algunas técnicas primitivas para buscarlos.

Para escribir programas útiles, necesitamos aprender a trabajar con datos. En esta clase vas a escribir un programa que lee un archivo de datos en formato csv y realiza un cálculo simple. Vas a aprender a estructurar tu código creando funciones y estructuras de datos un poco más complejas como tuplas, conjuntos y diccionarios.

Esta clase comenzamos a utilizar un entorno de desarrollo integrado (IDE). Vamos a trabajar con datos reales sobre los árboles de la ciudad de Buenos Aires en espacios públicos.

Recordá que, aunque no termines todos, completar el formulario del final de cada clase y entregar los ejercicios es parte de las condiciones de aprobación de la materia (todes nos podemos atrasar alguna vez, pero en general esperamos que entregues). Esto debe hacerse ANTES de la clase siguiente.

3.1 Errores

3.2 Secuencias

3.3 Contadores del módulo collections

3.4 Entorno de desarrollo integrado

3.5 Impresión con formato

3.6 Arbolado porteño

3.7 Cierre de la clase

3.1 Errores

Tres tipos de errores:

Programando nos podemos encontrar con tres tipos de errores.

Los errores sintácticos son los que se dan cuando escribimos incorrectamente. Por ejemplo si queremos escribir x = (a + b) \* c pero en vez de eso escribimos x = (a + b] \* c, el programa no va a correr.

Un segundo tipo de error lo forman los errores en tiempo de ejecución, que se dan cuando el programa empieza a ejecutarse pero se produce un error durante su ejecución. Por ejemplo si le pedimos al usuarie que ingrese su edad esperando un número entero e ingresa "veintiséis años", es probable que el programa dé un error. Si leemos un archivo CSV y una fila tiene datos faltantes, el programa puede dar un error. Este tipo de errores en Python generan excepciones que, como veremos más adelante, pueden administrarse adecuadamente.

El tercer tipo de error es el más difícil de encontrar y de entender. Son los errores semánticos, que se dan cuando el programa no hace lo que está diseñado para hacer. Tienen que ver con el sentido de las instrucciones. En estos casos el programa se ejecuta pero da un resultado incorrecto o inesperado. En general, la mejor forma de encontrar estos errores es correr paso a paso el código que genera un resultado inesperado, tratando de entender dónde está la falla, usando el debugger. Veremos cómo usar el debugger la clase que viene, por ahora trabajaremos de forma un poco más primitiva.

Debuggear a mano

Los errores (o bugs) son difíciles de rastrear y resolver. Especialmente errores que sólo aparecen bajo cierta combinación particular de condiciones que resulta en que el programa no pueda continuar o dé un resultado inesperado. Si tu programa corre, pero no da el resultado que esperás, o se cuelga y no entendés porqué, tenés algunas herramientas concretas que te ayudan a buscar el origen del problema. A continuación veremos algunas metodologías específicas (aunque un poco primitivas) que permiten rastrear el origen del problema.

¿Qué dice un traceback?

Si te da un error, lo primero que podés hacer es intentar entender la causa del error usando como punto de partida el "traceback":

python3 blah.py

Traceback (most recent call last):

File "blah.py", line 13, in ?

foo()

File "blah.py", line 10, in foo

bar()

File "blah.py", line 7, in bar

spam()

File "blah.py", line 4, in spam

x.append(3)

AttributeError: 'int' object has no attribute 'append'

La última línea dice algo como que "el objeto int no tiene un atributo append "- lo cual es obvio, pero ¿cómo llegamos ahí?

La última línea es el motivo concreto del error.

Las líneas anteriores te dicen el camino que siguió el programa hasta llegar al error. En este caso: el error ocurrió en x.append(3) en la línea 4, dentro de la función spam del módulo "blah.py", que fue llamado por la función bar en la línea 7 del mismo archivo, que fue llamada por... y así siguiendo.

Sin embargo a veces esto no proporciona suficiente información (por ejemplo, no sabemos el valor de cada parámetro usado en las llamadas).

Una posibilidad que a veces da resultado es copiar el traceback en Google. Si estás usando una biblioteca de funciones que mucha gente usa (como numpy ó math) es muy probable que alguien se haya encontrado antes con el mismo problema que vos, y alguien más le haya explicado qué lo causa, o cómo evitarlo.

Usá el modo REPL de Python

Si usás Python desde la línea de comandos, podés usarlo pasándole un -i como parámetro antes del script a ejecutar. Cuando el intérprete de Python termine de ejecutar el script se va a quedar en modo interactivo (en lugar de volver al sistema operativo). Podés averiguar en qué estado quedó el sistema.

python3 -i blah.py

Traceback (most recent call last):

File "blah.py", line 13, in ?

foo()

File "blah.py", line 10, in foo

bar()

File "blah.py", line 7, in bar

spam()

File "blah.py", line 4, in spam

x.append(3)

AttributeError: 'int' object has no attribute 'append'

>>> print( repr(x) )

Este parámetro (el -i, que ya usamos antes) preserva el estado del intérprete al finalizar el script y te permite interrogarlo sobre el estado de las variables y obtener información que de otro modo perderías. En el ejemplo de recién interesa saber qué es x y cómo llegó a ese estado. Si estás usando un IDE esta posibilidad de interacción suele ocurrir naturalmente.

Debuggear con print

print() es una forma rápida y sencilla de permitir que el programa se ejecute (casi) normalmente mientras te da información del estado de las variables. Si elegís bien las variables que mostrar, es probable que digas "¡¡Ajá!!".

Sugerencia: es conveniente usar repr() para imprimir las variables

def spam(x):

print('DEBUG:', repr(x))

...

repr() te muestra una representación técnicamente más precisa del valor de una variable, y no la representación bonita que solemos ver.

>>> from decimal import Decimal

>>> x = Decimal('3.4')

# SIN `repr`

>>> print(x)

3.4

# CON `repr`

>>> print(repr(x))

Decimal('3.4')

>>>

Debuggear con lápiz y papel

Muchas veces uno asume que el intérprete está haciendo algo. Si agarrás un lápiz y un papel y hacés de intérprete anotando el estado de cada variable y siguiendo las instrucciones del programa paso a paso, es posible que entiendas que las cosas no son como creías.

Estas alternativas son útiles pero un poco primitivas. La mejor forma de debuggear un programa en Python es usar el debugger.

Ejercicios:

En los siguientes ejercicios te proponemos que uses las técnicas que mencionamos arriba para resolver los problemas que aparecen a continuación. Determiná los errores de los siguientes códigos y corregilos en un archivo solucion\_de\_errores.py comentando brevemente los errores. ¿Qué tipo de errores tiene cada uno?

En el archivo solucion\_de\_errores.py separá las correcciones de los distintos códigos con una línea que contenga solamente los símbolos #%% seguido de una o varias líneas comentadas indicando el ejercicio y el problema que tenía. Al terminar, debería verse así tu archivo:

#solucion\_de\_errores.py

#Ejercicios de errores en el código

#%%

#Ejercicio 3.1. Función tiene\_a()

#Comentario: El error era de TAL tipo y estaba ubicado en TAL lugar.

# Lo corregí cambiando esto por aquello.

# A continuación va el código corregido

...

...

#%%

#Ejercicio 3.2. Función tiene\_a(), nuevamente

#Comentario: El error era de TAL tipo y estaba ubicado en TAL lugar.

...

...

#%%

#Ejercicio 3.3. Función tiene\_uno()

#Comentario: El error era de TAL tipo y estaba ubicado en TAL lugar.

...

...

...

Ejercicio 3.1: Semántica

¿Anda bien en todos los casos de prueba?

def tiene\_a(expresion):

n = len(expresion)

i = 0

while i<n:

if expresion[i] == 'a':

return True

else:

return False

i += 1

tiene\_a('UNSAM 2020')

tiene\_a('abracadabra')

tiene\_a('La novela 1984 de George Orwell')

Ejercicio 3.2: Sintaxis

¿Anda bien en todos los casos de prueba?

def tiene\_a(expresion)

n = len(expresion)

i = 0

while i<n

if expresion[i] = 'a'

return True

i += 1

return Falso

tiene\_a('UNSAM 2020')

tiene\_a('La novela 1984 de George Orwell')

Ejercicio 3.3: Tipos

¿Anda bien en todos los casos de prueba?

def tiene\_uno(expresion):

n = len(expresion)

i = 0

tiene = False

while (i<n) and not tiene:

if expresion[i] == '1':

tiene = True

i += 1

return tiene

tiene\_uno('UNSAM 2020')

tiene\_uno('La novela 1984 de George Orwell')

tiene\_uno(1984)

Ejercicio 3.4: Alcances

La siguiente suma no da lo que debería:

def suma(a,b):

c = a + b

a = 2

b = 3

c = suma(a,b)

print(f"La suma da {a} + {b} = {c}")

Ejercicio 3.5: Pisando memoria

El siguiente ejemplo usa el dataset de la clase anterior, pero no lo imprime como corresponde, ¿podés determinar por qué y explicarlo brevemente en la versión corregida?

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)

Ayuda: Primero tratá de pensarlo, pero si este último se te hace muy difícil, podés mirar un poco de la teoría relacionada con esto un par de secciones más adelante (Sección 4.4).

3.2 Secuencias

Tipo de secuencias

Python tiene tres tipos de datos que son secuencias.

String: 'Hello'. Una cadena es una secuencia de caracteres.

Lista: [1, 4, 5].

Tupla: ('Pera', 100, 490.1).

Todas las secuencias tienen un orden, indexado por enteros, y tienen una longitud.

a = 'Hello' # String o cadena

b = [1, 4, 5] # Lista

c = ('Pera', 100, 490.1) # Tupla

# Orden indexado

a[0] # 'H'

b[-1] # 5

c[1] # 100

# Longitud de secuencias

len(a) # 5

len(b) # 3

len(c) # 3

Las secuencias pueden ser replicadas: s \* n.

>>> a = 'Hello'

>>> a \* 3

'HelloHelloHello'

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

>>> b \* 2

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

>>>

Las secuencias del mismo tipo también pueden ser concatenadas: s + t.

>>> a = (1, 2, 3)

>>> b = (4, 5)

>>> a + b

(1, 2, 3, 4, 5)

>>>

>>> c = [1, 5]

>>> a + c

Traceback (most recent call last):

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

TypeError: can only concatenate tuple (not "list") to tuple

Rebanadas (slicing)

Sacar una rebanada es tomar una subsecuencia de una secuencia. La sintaxis es s[comienzo:fin], donde comienzo y fin son los índices de la subsecuencia que querés.

a = [0,1,2,3,4,5,6,7,8]

a[2:5] # [2,3,4]

a[-5:] # [4,5,6,7,8]

a[:3] # [0,1,2]

Los índices comienzo y fin deben ser enteros.

Las rebanadas no incluyen el valor final. Es como los intervalos semi-abiertos en matemática.

Si los índices son omitidos toman sus valores por defecto: el principio o el final de la lista.

Reasigación de rebanadas

En listas, una rebanada puede ser reasignada o eliminada.

# Reasignación

a = [0,1,2,3,4,5,6,7,8]

a[2:4] = [10,11,12] # [0,1,10,11,12,4,5,6,7,8]

Observación: La rebanada reasignada no tiene que tener necesariamente la misma longitud.

# Eliminación

a = [0,1,2,3,4,5,6,7,8]

del a[2:4] # [0,1,4,5,6,7,8]

Reducciones de secuencias

Hay algunas operaciones usuales que reducen una secuencia a un solo valor.

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

>>> sum(s)

10

>>> min(s)

1

>>> max(s)

4

>>> t = ['Hello', 'World']

>>> max(t)

'World'

>>>

Iterar sobre una secuencia

Los ciclos for iteran sobre los elementos de una secuencia.

>>> s = [1, 4, 9, 16]

>>> for i in s:

... print(i)

...

1

4

9

16

>>>

En cada iteración del ciclo obtenés un nuevo elemento para trabajar. La variable iteradora va a tomar este nuevo valor. En el siguiente ejemplo la variable iteradora es x:

for x in s: # `x` es una variable iteradora

...instrucciones

En cada iteración, el valor previo de la variable (si hubo alguno) es sobreescrito. Luego de terminar el ciclo, la variable retiene su último valor.

El comando break

Podés usar el comando break para romper un ciclo antes de tiempo.

for name in namelist:

if name == 'Juana':

break

...

...

instrucciones

Cuando el comando break se ejecuta, sale del ciclo y se mueve a las siguientes instrucciones. El comando break sólo se aplica al ciclo más interno. Si un ciclo está anidado en otro ciclo, el comando no va a romper el ciclo externo.

El comando continue

Para saltear un elemento y moverse al siguiente, usá el comando continue.

for line in lines:

if line == '\n': # Salteo las instrucciones que procesan líneas

continue

# Instrucciones que procesan líneas

...

Éste es útil cuando el elemento encontrado no es de interés o es necesario ignorarlo en el procesamiento.

Ciclos sobre enteros

Para iterar sobre un rango de números enteros, usá range().

for i in range(100):

# i = 0,1,...,99

La sintaxis es range([comienzo,] fin [,paso]) (lo que figura entre corchetes es opcional).

for i in range(100):

# i = 0,1,...,99

...codigo

for j in range(10,20):

# j = 10,11,..., 19

...codigo

for k in range(10,50,2):

# k = 10,12,...,48

# Observá que va de a dos.

...codigo

El valor final nunca es incluido. Es como con las rebanadas.

comienzo es opcional. Por defecto es 0.

paso es opcional. Por defecto es 1.

range() calcula los valores a medida que los necesita. No guarda realmente en memoria el rango completo de números.

La función enumerate()

La función enumerate agrega un contador extra a una iteración.

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

for i, nombre in enumerate(nombres):

# i = 0, nombre = 'Edmundo'

# i = 1, nombre = 'Juana'

# i = 2, nombre = 'Rosita'

La forma general es enumerate(secuencia [, start = 0]). start es opcional. Un buen ejemplo de cuándo usar enumerate() es para llevar la cuenta del número de línea mientras estás leyendo un archivo:

with open(nombre\_archivo) as f:

for nlinea, line in enumerate(f, start=1):

...

Al fin de cuentas, enumerate es sólo una forma abreviada y simpática de escribir:

i = 0

for x in s:

instrucciones

i += 1

Al usar enumerate tenemos que tipear menos y el programa funciona un poco más rápido.

Tuplas y ciclos for

Podés iterar con múltiples variables de iteración.

points = [

(1, 4),(10, 40),(23, 14),(5, 6),(7, 8)

]

for x, y in points:

# x = 1, y = 4

# x = 10, y = 40

# x = 23, y = 14

# ...

Cuando usás múltiples variables, cada tupla es desempaquetada en un conjunto de variables de iteración. El número de variables debe coincidir con la cantidad de elementos de cada tupla.

La función zip()

La función zip toma múltiples secuencias y las combina en un iterador.

columnas = ['nombre', 'cajones', 'precio']

valores = ['Pera', 100, 490.1 ]

pares = zip(columnas, valores)

# ('nombre','Pera'), ('cajones',100), ('precio',490.1)

Para obtener el resultado debés iterar. Podés usar múltiples variables para desempaquetar las tuplas como mostramos antes.

for columna, valor in pares:

...

Un uso frecuente de zip es para crear pares clave/valor y construir diccionarios.

d = dict(zip(columnas, valores))

Ejercicios

Ejercicio 3.6: Contar

Probá algunos ejemplos elementales de conteo:

>>> for n in range(10): # Contar 0 ... 9

print(n, end=' ')

0 1 2 3 4 5 6 7 8 9

>>> for n in range(10,0,-1): # Contar 10 ... 1

print(n, end=' ')

10 9 8 7 6 5 4 3 2 1

>>> for n in range(0,10,2): # Contar 0, 2, ... 8

print(n, end=' ')

0 2 4 6 8

>>>

Ejercicio 3.7: Más operaciones con secuencias

Interactivamente experimentá con algunas operaciones de reducción de secuencias.

>>> data = [4, 9, 1, 25, 16, 100, 49]

>>> min(data)

1

>>> max(data)

100

>>> sum(data)

204

>>>

Probá iterar sobre los datos.

>>> for x in data:

print(x)

4

9

...

>>> for n, x in enumerate(data):

print(n, x)

0 4

1 9

2 1

...

>>>

A veces los comandos for, len(), y range() son combinados para recorrer listas:

>>> for n in range(len(data)):

print(data[n])

4

9

1

...

>>>

Sin embargo, Python tiene mejores alternativas para esto. Te recomendamos familiarizarte con ellas y usarlas: por su simpleza producen código más legible y reducen la posibilidad de un bug en el código. Simplemente usá un ciclo for normal si querés iterar sobre los elementos de la variable data. Y usá enumerate() si necesitás tener el índice por algún motivo.

Ejercicio 3.8: Un ejemplo práctico de enumerate()

Recordá que el archivo Data/missing.csv contiene datos sobre los cajones de un camión, pero tiene algunas filas que faltan. Usando enumerate(), modificá tu programa costo\_camion.py de forma que imprima un aviso (warning) cada vez que encuentre una fila incorrecta.

>>> cost = costo\_camion('../Data/missing.csv')

Fila 4: No pude interpretar: ['Mandarina', '', '51.23']

Fila 7: No pude interpretar: ['Naranja', '', '70.44']

>>>

Para hacer esto, vas a tener que cambiar algunas partes de tu código.

...

for n\_fila, fila in enumerate(filas, start=1):

try:

...

except ValueError:

print(f'Fila {n\_fila}: No pude interpretar: {fila}')

Ejercicio 3.9: La función zip()

En el archivo Data/camion.csv, la primera línea tiene los encabezados de las columnas. En los códigos anteriores la descartamos.

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

>>> filas = csv.reader(f)

>>> encabezados = next(filas)

>>> encabezados

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

>>>

Pero, ¿no puede ser útil conocer los encabezados? Es acá donde la función zip() entra en acción. Primero tratá de aparear los encabezados con una fila de datos:

>>> fila = next(filas)

>>> fila

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

>>> list(zip(encabezados, fila))

[ ('nombre', 'Lima'), ('cajones', '100'), ('precio', '32.20') ]

>>>

Fijate cómo zip() apareó los encabezados de las columnas con los valores de la columna. Usamos list() arriba para devolver el resultado en una lista de forma que lo puedas ver. Normalmente, zip() crea un iterador que debe ser consumido en un ciclo for.

Este apareamiento es un paso intermedio para construir un diccionario. Probá lo siguiente:

>>> record = dict(zip(encabezados, fila))

>>> record

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

>>>

Esta transformación es un truco sumamente útil cuando tenés que procesar muchos archivos de datos. Por ejemplo, suponé que querés hacer que el programa costo\_camion.py trabaje con diferentes archivos de entrada, pero que no le importe la posición exacta de la columna que tiene la cantidad de cajones o el precio. Es decir, que entienda que la columna tiene el precio por su encabezado y no por su posición dentro del archivo.

Modificá la función costo\_camion() en el archivo costo\_camion.py para que se vea así:

# costo\_camion.py

def costo\_camion(nombre\_archivo):

...

for n\_fila, fila in enumerate(filas, start=1):

record = dict(zip(encabezados, fila))

try:

ncajones = int(record['cajones'])

precio = float(record['precio'])

costo\_total += ncajones \* precio

# Esto atrapa errores en los int() y float() de arriba.

except ValueError:

print(f'Fila {n\_fila}: No pude interpretar: {fila}')

...

Ahora, probá tu función con un archivo completamente diferente Data/fecha\_camion.csv que se ve así:

nombre,fecha,hora,cajones,precio

"Lima","6/11/2007","9:50am",100,32.20

"Naranja","5/13/2007","4:20pm",50,91.10

"Caqui","9/23/2006","1:30pm",150,83.44

"Mandarina","5/17/2007","10:30am",200,51.23

"Durazno","2/1/2006","10:45am",95,40.37

"Mandarina","10/31/2006","12:05pm",50,65.10

"Naranja","7/9/2006","3:15pm",100,70.44

>>> costo\_camion('../Data/fecha\_camion.csv')

47671.15

>>>

Si lo hiciste bien, vas a descubrir que tu programa aún funciona a pesar de que le pasaste un archivo con un formato de columnas completamente diferente al de antes. ¡Y eso está muy bueno!

El cambio que hicimos acá es sutil, pero importante. En lugar de tener hardcodeado un formato fijo, la nueva versión de la función costo\_camion() puede sacar la información de interés de cualquier archivo CSV. En la medida en que el archivo tenga las columnas requeridas, el código va a funcionar.

Modificá el programa informe.py que escribiste antes (ver Ejercicio 2.18) para que use esta técnica para elegir las columnas a partir de sus encabezados.

Probá correr el programa informe.py sobre el archivo Data/fecha\_camion.csv y fijate si da la misma salida que antes.

Ejercicio 3.10: Invertir un diccionario

Un diccionario es una función que mapea claves en valores. Por ejemplo, un diccionario de precios de cajones de frutas.

>>> precios = {

'Pera' : 490.1,

'Lima' : 23.45,

'Naranja' : 91.1,

'Mandarina' : 34.23

}

>>>

Si usás el método items(), obtenés pares (clave,valor):

>>> precios.items()

dict\_items([('Pera', 490.1), ('Lima', 23.45), ('Naranja', 91.1), ('Mandarina', 34.23)])

>>>

Sin embargo, si lo que querés son pares (valor, clave), ¿cómo lo hacés? Ayuda: usá zip().

>>> lista\_precios = list(zip(precios.values(),precios.keys()))

>>> lista\_precios

[(490.1, 'Pera'), (23.45, 'Lima'), (91.1, 'Naranja'), (34.23, 'Mandarina')]

>>>

¿Por qué haría algo así? Por un lado porque te permite realizar cierto tipo de procesamiento de datos sobre la información del diccionario.

>>> min(lista\_precios)

(23.45, 'Lima')

>>> max(lista\_precios)

(490.1, 'Pera')

>>> sorted(lista\_precios)

[(23.45, 'Lima'), (34.23, 'Mandarina'), (91.1, 'Naranja'), (490.1, 'Pera')]

>>>

Esto también ilustra un atributo importante de las tuplas. Cuando son usadas en una comparación, las tuplas son comparadas elemento-a-elemento comenzando con el primero. Es similar a la lógica subyacente al orden lexicográfico o alfabético en las cadenas.

La función zip() se usa frecuentemente en este tipo de situaciones donde necesitás aparear datos provenientes de diferentes lugares. Por ejemplo, para aparear los nombres de las columnas con los valores para hacer un diccionario de valores con nombres.

Observá que zip() no está limitada a pares. Podés usarla con cualquier número de listas de entrada:

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

>>> b = ['w', 'x', 'y', 'z']

>>> c = [0.2, 0.4, 0.6, 0.8]

>>> list(zip(a, b, c))

[(1, 'w', 0.2), (2, 'x', 0.4), (3, 'y', 0.6), (4, 'z', 0.8))]

>>>

También, tené en cuenta que zip() se detiene cuando la más corta de las entradas se agota.

>>> a = [1, 2, 3, 4, 5, 6]

>>> b = ['x', 'y', 'z']

>>> list(zip(a,b))

[(1, 'x'), (2, 'y'), (3, 'z')]

>>>

3.3 Contadores del módulo collections

El módulo collections ofrece objetos útiles para manejar datos. En esta sección introducimos brevemente los contadores, que son solo una de las clases incluidas en este módulo.

Ejemplo: Contar cosas

Digamos que querés hacer una tabla con el total de cajones de cada fruta.

camion = [

('Pera', 100, 490.1),

('Naranja', 50, 91.1),

('Caqui', 150, 83.44),

('Naranja', 100, 45.23),

('Pera', 75, 572.45),

('Lima', 50, 23.15)

]

Hay dos entradas de Naranja y dos de Pera en esta lista. Estos cajones deben ser combinados juntos de alguna forma.

Contadores

Solución: Usá un Counter (contador).

from collections import Counter

total\_cajones = Counter()

for nombre, n\_cajones, precio in camion:

total\_cajones[nombre] += n\_cajones

total\_cajones['Naranja'] # 150

Ejercicios

En este ejercicio vas a probar contadores en un par de ejemplos simples. Cargá tu programa informe.py y ejecutalo en el interprete de forma de tener los datos del camión con cajones cargado en modo interactivo.

Podés usar el interprete desde la línea de comandos ejecutando:

bash % python3 -i informe.py

O podés cargarlo en el Spyder y correrlo.

Ejercicio 3.11: Contadores

Vamos a usar un contador (objeto Counter) para contar cajones de frutas. Probalo:

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

>>> from collections import Counter

>>> tenencias = Counter()

>>> for s in camion:

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

>>> tenencias

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

>>>

Observá que la entradas múltiples como Mandarina y Naranja en camion se combinan en una sola entrada.

Podés usar el contador como un diccionario para recuperar valores individuales:

>>> tenencias['Naranja']

150

>>> tenencias['Mandarina']

250

>>>

Podés listar las tres frutas con mayores tenencias:

>>> # Las 3 frutas con más cajones

>>> tenencias.most\_common(3)

[('Mandarina', 250), ('Naranja', 150), ('Caqui', 150)]

>>>

Carguemos los datos de otro camión con cajones de fruta en un nuevo contador:

>>> camion2 = leer\_camion('../Data/camion2.csv')

>>> tenencias2 = Counter()

>>> for s in camion2:

tenencias2[s['nombre']] += s['cajones']

>>> tenencias2

Counter({'Durazno': 125, 'Frambuesa': 250, 'Lima': 50, 'Mandarina': 25})

>>>

Y finalmente combinemos las tenencias de ambos camiones con una operación simple:

>>> tenencias

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

>>> tenencias2

Counter({'Frambuesa': 250, 'Durazno': 125, 'Lima': 50, 'Mandarina': 25})

>>> combinada = tenencias + tenencias2

>>> combinada

Counter({'Caqui': 150, 'Durazno': 220, 'Frambuesa': 250, 'Lima': 150, 'Mandarina': 275, 'Naranja': 150})

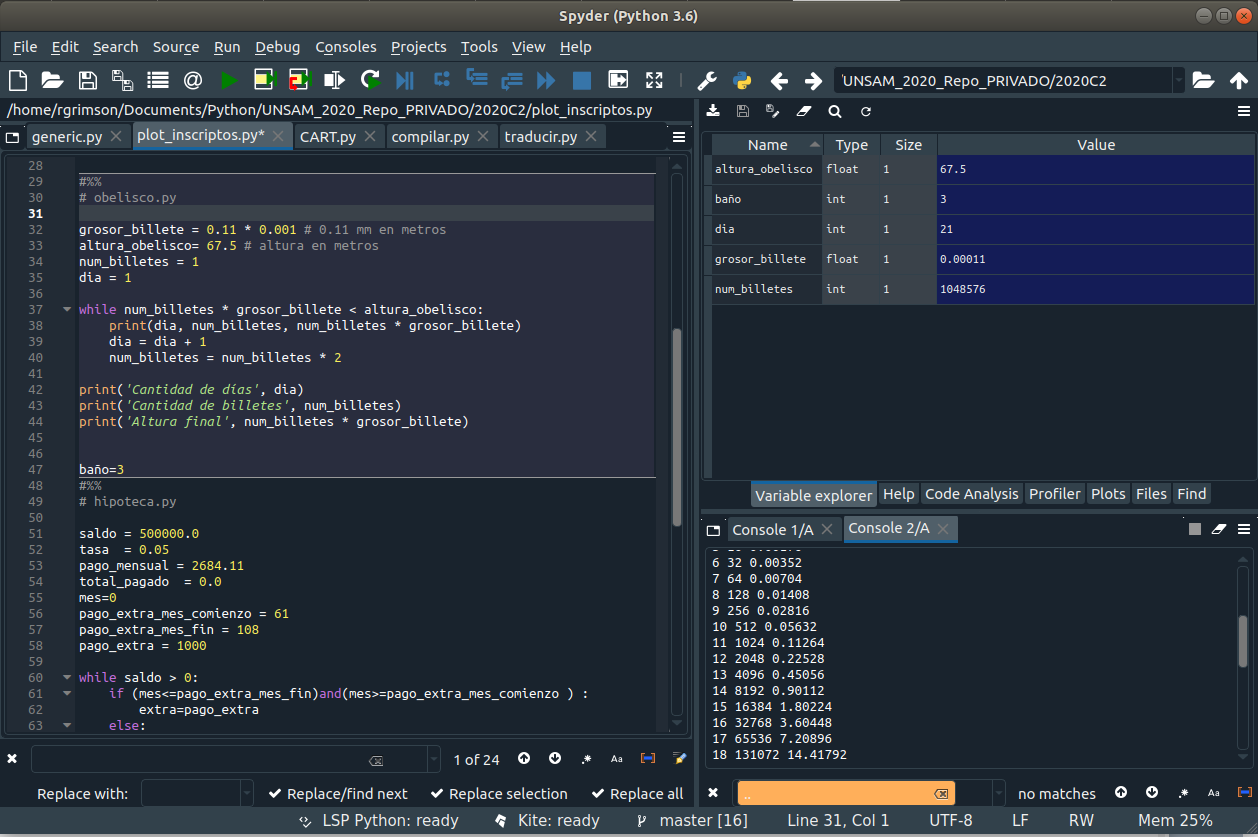
>>>

Esto es solo una pequeña muestra de lo que se puede hacer con contadores. El módulo collections es muy poderoso pero meterse a ver sus detalles sería una distracción ahora. Sigamos con nuestro curso...

**3.4 Entorno de desarrollo integrado**

A partir de aquí te vamos a proponer trabajar principalmente dentro de un entorno de desarrollo integrado (IDE, por sus siglas en inglés).

En particular, sugerimos trabajar con el [Spyder](https://www.spyder-ide.org/) que es un entorno de desarrollo de Python diseñado para científicos, ingenieros y analistas de datos. El Spyder puede descargarse solo o como parte de la distribución de [Anaconda](https://www.anaconda.com/products/individual) que trae, además de Python y del Spyder, una serie de bibliotecas con módulos muy útiles para desarrollos relacionados a la ciencia de datos.

Esta es una imagen de una captura de pantalla del Spyder: [](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/03_Datos/Spyder.png)

En la captura se puede ver que por defecto el Spyder viene estructurado con tres ventanas. Un editor de código ocupa la mitad izquierda de la ventana, mientras que la mitad derecha se divide en una terminal (o consola) interactiva de Python en la mitad inferior y un inspector de variables en la mitad superior. El Spyder nos permite correr línea por línea el código del editor (tecla F9) y ver el estado de las variables en el inspector de variables, o ejecutarlo completo (tecla F5). También nos permite debuguear el código con facilidad (botones azules de la barra superior).

Les recomendamos que le dediquen un tiempo a probar sus últimos ejercicios en este entorno. Verán que es muy cómodo. Pueden mirar un [breve tutorial](https://www.youtube.com/watch?v=0fxURPC1YFs) donde no sólo les enseñan el uso de la tecla F5, sino también una introducción al uso del debugger (le dice depurador) que veremos más adelante en este curso.

**3.5 Impresión con formato**

En esta sección se ven detalles técnicos sobre cómo hacer que la salida por pantalla sea más amena para el usuario.

Cuando trabajás con datos es usual que quieras imprimir salidas estructuradas (tablas, etc.). Por ejemplo:

Nombre Cajones Precio

---------- ---------- -----------

Lima 100 32.20

Naranja 50 91.10

Caqui 150 103.44

Mandarina 200 51.23

Durazno 95 40.37

Mandarina 50 65.10

Naranja 100 70.44

**Formato de cadenas**

Una excelente manera de darle formato a una cadena en Python (a partir de la versión 3.6) es usando f-strings.

>>> nombre = 'Naranja'

>>> cajones = 100

>>> precio = 91.1

>>> f'{nombre:>10s} {cajones:>10d} {precio:>10.2f}'

' Naranja 100 91.10'

>>>

La parte {expresion:formato} va a ser reemplazada. Usualmente los f-strings se usan con print.

print(f'{nombre:>10s} {cajones:>10d} {precio:>10.2f}')

**Códigos de formato**

Lo códigos de formato (lo que va luego de : dentro de {}) son similares a los que se usan en el printf() del lenguaje C. Los más comunes son:

d Entero decimal

b Entero binario

x Entero hexadecimal

f Flotante como [-]m.dddddd

e Flotante como [-]m.dddddde+-xx

g Flotante, pero con uso selectivo de la notación exponencial E.

s Cadenas

c Caracter (a partir de un entero, su código)

Los modificadores permiten ajustar el ancho a imprimir o la precisión decimal (cantidad de dígitos luego del punto). Ésta es una lista parcial:

:>10d Entero alineado a la derecha en un campo de 10 caracteres

:<10d Entero alineado a la izquierda en un campo de 10 caracteres

:^10d Entero centrado en un campo de 10 caracteres

:0.2f Flotante con dos dígitos de precisión

**Formato a diccionarios**

Podés usar el método format\_map() para aplicarle un formato a los valores de un diccionario:

>>> s = {

'nombre': 'Naranja',

'cajones': 100,

'precio': 91.1

}

>>> '{nombre:>10s} {cajones:10d} {precio:10.2f}'.format\_map(s)

' Naranja 100 91.10'

>>>

Usa los mismos códigos que los f-strings pero toma los valores que provee el diccionario.

**El método format()**

Existe un método format() que permite aplicar formato a argumentos.

>>> '{nombre:>10s} {cajones:10d} {precio:10.2f}'.format(nombre='Naranja', cajones=100, precio=91.1)

' Naranja 100 91.10'

>>> '{:10s} {:10d} {:10.2f}'.format('Naranja', 100, 91.1)

' Naranja 100 91.10'

>>>

La verdad es que format() nos resulta un poco extenso y preferimos usar f-strings.

**Formato estilo C**

También podés usar el operador %.

>>> 'The value is %d' % 3

'The value is 3'

>>> '%5d %-5d %10d' % (3,4,5)

' 3 4 5'

>>> '%0.2f' % (3.1415926,)

'3.14'

Esto requiere un solo ítem, o una tupla a la derecha. Los códigos están también inspirados en el printf() de C. Tiene la dificultad de que hay que contar posiciones y todas las variables van juntas al final.

**Ejercicios**

**Ejercicio 3.12: Formato de números**

Un problema usual cuando queremos imprimir números es especificar el número de dígitos decimales. Los f-strings nos permiten hacerlo. Probá los siguientes ejemplos:

>>> value = 42863.1

>>> print(value)

42863.1

>>> print(f'{value:0.4f}')

42863.1000

>>> print(f'{value:>16.2f}')

42863.10

>>> print(f'{value:<16.2f}')

42863.10

>>> print(f'{value:\*>16,.2f}')

\*\*\*\*\*\*\*42,863.10

>>>

La documentación completa sobre los códigos de formato usados en f-strings puede consultarse [acá](https://docs.python.org/3/library/string.html#format-specification-mini-language). El formato puede aplicarse también usando el operador % de cadenas.

>>> print('%0.4f' % value)

42863.1000

>>> print('%16.2f' % value)

42863.10

>>>

La documentación sobre códigos usados con % puede encontrarse [acá](https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting).

A pesar de que suelen usarse dentro de un print, el formato de cadenas no está necesariamente ligado a la impresión. Por ejemplo, podés simplemente asignarlo a una variable.

>>> f = '%0.4f' % value

>>> f

'42863.1000'

>>>

**Ejercicio 3.13: Recolectar datos**

En el [Ejercicio 2.18](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/02_Estructuras_y_Funciones/04_Contenedores.md#ejercicio-218-balances), escribiste un programa llamado informe.py que calculaba las ganancias o pérdidas de un camión que compra a productores y vende en el mercado. Copiá su contenido en un archivo tabla\_informe.py. En este ejercicio, vas a comenzar a modificar tabla\_informe.py para producir una tabla como ésta:

Nombre Cajones Precio Cambio

---------- ---------- ---------- ----------

Lima 100 32.2 8.02

Naranja 50 91.1 15.18

Caqui 150 103.44 2.02

Mandarina 200 51.23 29.66

Durazno 95 40.37 33.11

Mandarina 50 65.1 15.79

Naranja 100 70.44 35.84

En este informe, el "Precio" es el precio en el mercado y el "Cambio" es la variación respecto al precio cobrado por el productor.

Para generar un informe como el de arriba, primero tenés que recolectar todos los datos de la tabla. Escribí una función hacer\_informe() que recibe una lista de cajones y un diccionario con precios como input y devuelve una lista de tuplas conteniendo la información mostrada en la tabla anterior.

Agregá esta función a tu archivo tabla\_informe.py. Debería funcionar como se muestra en el siguiente ejemplo:

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

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

>>> informe = hacer\_informe(camion, precios)

>>> for r in informe:

print(r)

('Lima', 100, 32.2, 8.019999999999996)

('Naranja', 50, 91.1, 15.180000000000007)

('Caqui', 150, 103.44, 2.019999999999996)

('Mandarina', 200, 51.23, 29.660000000000004)

('Durazno', 95, 40.37, 33.11000000000001)

('Mandarina', 50, 65.1, 15.790000000000006)

('Naranja', 100, 70.44, 35.84)

...

>>>

**Ejercicio 3.14: Imprimir una tabla con formato**

Volvé a hacer el ciclo for del ejercicio anterior pero cambiando la forma de imprimir como sigue:

>>> for r in informe:

print('%10s %10d %10.2f %10.2f' % r)

Lima 100 32.20 8.02

Naranja 50 91.10 15.18

Caqui 150 103.44 2.02

Mandarina 200 51.23 29.66

Durazno 95 40.37 33.11

Mandarina 50 65.10 15.79

Naranja 100 70.44 35.84

...

>>>

O directamente usando f-strings. Por ejemplo:

>>> for nombre, cajones, precio, cambio in informe:

print(f'{nombre:>10s} {cajones:>10d} {precio:>10.2f} {cambio:>10.2f}')

Lima 100 32.20 8.02

Naranja 50 91.10 15.18

Caqui 150 103.44 2.02

Mandarina 200 51.23 29.66

Durazno 95 40.37 33.11

Mandarina 50 65.10 15.79

Naranja 100 70.44 35.84

...

>>>

Agregá estos últimos comandos a tu programa tabla\_informe.py. Hacé que el programa tome la salida de la función hacer\_informe() e imprima una tabla bien formateada.

**Ejercicio 3.15: Agregar encabezados**

Suponete que tenés una tupla con nombres de encabezado como ésta:

headers = ('Nombre', 'Cajones', 'Precio', 'Cambio')

Agregá el código necesario a tu programa para que tome una tupla de encabezados como la de arriba y cree una cadena donde cada nombre de encabezado esté alineado a la derecha en un campo de 10 caracteres de ancho y separados por un solo espacio.

' Nombre Cajones Precio Cambio'

Escribí el código que recibe los encabezados y crea una cadena de separación entre los encabezados y los datos que siguen. Esta cadena es simplemente una tira de caracteres "-" bajo cada nombre de campo. Por ejemplo:

'---------- ---------- ---------- ----------'

Cuando esté listo, tu programa debería producir una tabla como esta:

Nombre Cajones Precio Cambio

---------- ---------- ---------- ----------

Lima 100 32.20 8.02

Naranja 50 91.10 15.18

Caqui 150 103.44 2.02

Mandarina 200 51.23 29.66

Durazno 95 40.37 33.11

Mandarina 50 65.10 15.79

Naranja 100 70.44 35.84

**Ejercicio 3.16: Un desafío de formato**

Por último, modificá tu código para que el precio mostrado incluya un símbolo de pesos ($) y la salida se vea como esta tabla:

Nombre Cajones Precio Cambio

---------- ---------- ---------- ----------

Lima 100 $32.2 8.02

Naranja 50 $91.1 15.18

Caqui 150 $103.44 2.02

Mandarina 200 $51.23 29.66

Durazno 95 $40.37 33.11

Mandarina 50 $65.1 15.79

Naranja 100 $70.44 35.84

Guardá estos cambios en el archivo tabla\_informe.py que más adelante los vas a necesitar.

**Ejercicio 3.17: Tablas de multiplicar**

Escribí un programa tablamult.py que imprima de forma prolija las tablas de multiplicar del 1 al 9 usando f-strings. Si podés, evitá usar la multiplicación, usando sólo sumas alcanza.

0 1 2 3 4 5 6 7 8 9

---------------------------------------------

0: 0 0 0 0 0 0 0 0 0 0

1: 0 1 2 3 4 5 6 7 8 9

2: 0 2 4 6 8 10 12 14 16 18

3: 0 3 6 9 12 15 18 21 24 27

4: 0 4 8 12 16 20 24 28 32 36

5: 0 5 10 15 20 25 30 35 40 45

6: 0 6 12 18 24 30 36 42 48 54

7: 0 7 14 21 28 35 42 49 56 63

8: 0 8 16 24 32 40 48 56 64 72

9: 0 9 18 27 36 45 54 63 72 81

**3.6 Arbolado porteño**

En esta sección haremos algunos ejercicios que integran los conceptos aprendidos en las clases anteriores. Vamos a manejar archivos, diccionarios, listas, contadores y el comando zip, entre otras cosas. Entregá lo que puedas hacer.

**Ejercicios**

Vamos a repasar las herramientas que vimos en esta clase aplicándolas a una base de datos sobre árboles en parques de la Ciudad de Buenos Aires. Para empezar, descargá el archivo CSV de "[Arbolado en espacios verdes](https://data.buenosaires.gob.ar/dataset/arbolado-espacios-verdes)" en tu carpeta Data. Vamos a estudiar esta base de datos y responder algunas preguntas. Guardá los ejercicios de esta sección en un archivo arboles.py.

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/03_Datos/arboles.jpg)

**Descripción de la base**

| **Título de la columna** | **Tipo de dato** | **Descripción** |
| --- | --- | --- |
| long | Número flotante (float) | Coordenadas para geolocalización |
| lat | Número flotante (float) | Coordenadas para geolocalización |
| id\_arbol | Número entero (integer) | Identificador único del árbol |
| altura\_tot | Número entero (integer) | Altura del árbol (m) |
| diametro | Número entero (integer) | Diámetro del árbol (cm) |
| inclinacio | Número entero (integer) | Inclinación del árbol (grados) |
| id\_especie | Número entero (integer) | Identificador de la especie |
| nombre\_com | Texto (string) | Nombre común del árbol |
| nombre\_cie | Texto (string) | Nombre científico del árbol |
| tipo\_folla | Texto (string) | Tipo de follaje del árbol |
| espacio\_ve | Texto (string) | Nombre del espacio verde |
| ubicacion | Texto (string) | Dirección del espacio verde |
| nombre\_fam | Texto (string) | Nombre de la familia del árbol |
| nombre\_gen | Texto (string) | Nombre del género del árbol |
| origen | Texto (string) | Origen del árbol |
| coord\_x | Número flotante (float) | Coordenadas para localización |
| coord\_y | Número flotante (float) | Coordenadas para localización |

**Ejercicio 3.18: Lectura de los árboles de un parque**

Definí una función leer\_parque(nombre\_archivo, parque) que abra el archivo indicado y devuelva una **lista de diccionarios** con la información del parque especificado. La función debe devolver, en una lista un diccionario con todos los datos por cada árbol del parque elegido (recordá que cada fila del csv es un árbol).

*Sugerencia: basate en la función leer\_camion() y usá también el comando zip como hiciste en el* [Ejercicio 3.9](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/03_Datos/02_Secuencias.md#ejercicio-39-la-funci%C3%B3n-zip) *para combinar el encabezado del archivo con los datos de cada fila. Inicialmente no te preocupes por los tipos de datos de cada columna, pero cuando empieces a operar con una columna modificá esta función para que ese dato sea del tipo adecuado para operar.*

*Observación: La columna que indica el nombre del parque en el que se encuentra el árbol se llama 'espacio\_ve' en el archivo CSV.*

Probá con el parque "GENERAL PAZ" para tener un ejemplo de trabajo, debería darte una lista con 690 árboles.

**Ejercicio 3.19: Determinar las especies en un parque**

Escribí una función especies(lista\_arboles) que tome una lista de árboles como la generada en el ejercicio anterior y devuelva el conjunto de especies (la columna 'nombre\_com' del archivo) que figuran en la lista.

*Sugerencia: Usá el comando set como en la*[*Sección 2.4*](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/02_Estructuras_y_Funciones/04_Contenedores.md#conjuntos)*.*

**Ejercicio 3.20: Contar ejemplares por especie**

Usando contadores como en el [Ejercicio 3.11](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/03_Datos/03_Contadores.md#ejercicio-311-contadores), escribí una función contar\_ejemplares(lista\_arboles) que, dada una lista como la que generada con leer\_parque(), devuelva un diccionario en el que las especies (recordá, es la columna 'nombre\_com' del archivo) sean las claves y tengan como valores asociados la cantidad de ejemplares en esa especie en la lista dada.

Luego, combiná esta función con leer\_parque() y con el método most\_common() para informar las cinco especies más frecuentes en cada uno de los siguientes parques:

* 'GENERAL PAZ'
* 'ANDES, LOS'
* 'CENTENARIO'

**Resultados** de cantidad por especie en tres parques:

| **General Paz** | **Los Andes** | **Centenario** |
| --- | --- | --- |
| Casuarina: 97 | Jacarandá: 117 | Plátano: 137 |
| Tipa blanca: 54 | Tipa blanca: 28 | Jacarandá: 45 |
| Eucalipto: 49 | Ciprés: 21 | Tipa blanca: 42 |
| Palo borracho rosado: 44 | Palo borracho rosado: 18 | Palo borracho rosado: 41 |
| Fenix: 40 | Lapacho: 12 | Fresno americano: 38 |

**Ejercicio 3.21: Alturas de una especie en una lista**

Escribí una función obtener\_alturas(lista\_arboles, especie) que, dada una lista de árboles como la anterior y una especie de árbol (un valor de la columna 'nombre\_com' del archivo), devuelva una lista con las alturas (columna 'altura\_tot') de los ejemplares de esa especie en la lista.

*Observación: Acá sí, fijate de devolver las alturas como números (de punto flotante) y no como cadenas de caracteres. Podés hacer esto modificando leer\_parque*.

Usala para calcular la altura promedio y altura máxima de los 'Jacarandá' en los tres parques mencionados.

**Resultados** de alturas de Jacarandás en tres parques:

| **Medida** | **General Paz** | **Los Andes** | **Centenario** |
| --- | --- | --- | --- |
| max | 16.0 | 25.0 | 18.0 |
| prom | 10.2 | 10.54 | 8.96 |

**Ejercicio 3.22: Inclinaciones por especie de una lista**

Escribí una función obtener\_inclinaciones(lista\_arboles, especie) que, dada una especie de árbol y una lista de árboles como la anterior, devuelva una lista con las inclinaciones (columna 'inclinacio') de los ejemplares de esa especie.

**Ejercicio 3.23: Especie con el ejemplar más inclinado**

Combinando la función especies() con obtener\_inclinaciones() escribí una función especimen\_mas\_inclinado(lista\_arboles) que, dada una lista de árboles devuelva la especie que tiene el ejemplar más inclinado y su inclinación.

Correlo para los tres parques mencionados anteriormente.

**Resultados.** Deberías obtener, por ejemplo, que en el Parque Centenario hay un *Falso Guayabo* inclinado 80 grados.

**Ejercicio 3.24: Especie más inclinada en promedio**

Volvé a combinar las funciones anteriores para escribir la función especie\_promedio\_mas\_inclinada(lista\_arboles) que, dada una lista de árboles devuelva la especie que en promedio tiene la mayor inclinación y el promedio calculado..

**Resultados.** Deberías obtener, por ejemplo, que los *Álamos Plateados* del Parque Los Andes tiene un promedio de inclinación de 25 grados.

**Preguntas extras:** ¿Qué habría que cambiar para obtener la especie con un ejemplar más inclinado de toda la ciudad y no solo de un parque? ¿Podrías dar la latitud y longitud de ese ejemplar? ¿Y dónde se encuentra (lat,lon) el ejemplar más alto? ¿De qué especie es?

**3.7 Cierre de la clase**

En esta clase aprendimos a trabajar con datos. Manejamos archivos CSV y estructuras un poco más complejas como tuplas, conjuntos y diccionarios y profundizamos un poco más en las listas.

Te recordamos que leas el [código de honor](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/Codigo.md) del curso en el que hablamos de las reglas que rigen en este curso para evitar el plagio así como otros aspectos importantes sobre qué se puede compartir y qué no. Al enviar tus archivos entendemos que leíste y estas de acuerdo con este texto. En caso contrario no envíes tus archivos y contactate con los docentes.

Para cerrar esta clase te pedimos dos cosas:

* Que recopiles las soluciones de los siguientes ejercicios:
  1. El archivo solucion\_de\_errores.py 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) y siguientes.
  2. El archivo informe.py del [Ejercicio 3.9](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/03_Datos/02_Secuencias.md#ejercicio-39-la-funci%C3%B3n-zip).
  3. El archivo tabla\_informe.py del [Ejercicio 3.16](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/03_Datos/05_Formato.md#ejercicio-316-un-desaf%C3%ADo-de-formato) (o del [Ejercicio 3.15](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/03_Datos/05_Formato.md#ejercicio-315-agregar-encabezados) si no salió lo del signo $).
  4. El archivo tablamult.py del [Ejercicio 3.17](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/03_Datos/05_Formato.md#ejercicio-317-tablas-de-multiplicar).
  5. El archivo arboles.py sobre arbolado porteño incluyendo todos los ejercicios que hayas hecho (esperamos al menos el [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) y uno más).
* Que completes [este formulario](https://docs.google.com/forms/d/19zpbCFLl2jigepmsaSUoitWge8qpRL1Xg8tqynj1pnI) usando como identificación tu dirección de mail. Al terminar vas a obtener un link para enviarnos tus ejercicios y tendrás la opción de participar en la revisión de pares.

Esperamos en esta clase que entregues los cinco ejercicios que te pedimos.

Acordate, usá siempre la misma dirección de mail con la que te inscribiste al curso así podemos llevar registro de tus entregas.

Observación: Si el enunciado de un ejercicio te pide que lo corras con un input particular, por favor poné la salida que obtuviste como comentario en tu código.

Por último te recordamos que si te quedaron dudas, querés discutir algún tema de interés o pedirnos a los docentes que resolvamos un ejercicio particular para la próxima clase, podés hacelo en el [grupo de Slack](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/Slack.md).