. Estructuras y funciones

Para escribir programas útiles, necesitamos aprender a trabajar con datos. En esta clase introducimos las estructuras de datos elementales de Python que nos faltan: tuplas, conjuntos y diccionarios y profundizamos un poco más en las listas y sus usos. También comenzamos a ver cómo estructurar el código usando funciones y a leer archivos.

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 las 14hs del próximo miércoles.

2.1 Manejo de archivos

2.2 Funciones

2.3 Tipos y estructuras de datos

2.4 Contenedores

2.5 Cierre de la clase

2.1 Manejo de archivos

La mayoría de los programas usa alguna fuente de datos. En esta sección discutimos el acceso a archivos.

Archivos de entrada y salida

Con estos comandos podés abrir dos archivos, una para lectura y otro para escritura:

f = open('foo.txt', 'rt') # Abrir para lectura ('r' de read, 't' de text)

g = open('bar.txt', 'wt') # Abrir para escritura ('w' de write, 't' de text)

Observación: los nombres foo, bar, foobar son genéricos que se usan usualmente para aludir a un nombre que se ignora, son los análogos informáticos a fulano, mengano y zutano.

Para leer el archivo completo, o una parte:

data = f.read()

# Leer 'maxbytes' bytes

data = f.read([maxbytes])

Para escribir un texto en el archivo:

g.write('un texto')

Finalmente, hay que cerrar los archivos cuando terminamos de usarlos.

f.close()

g.close()

Es importante cerrar adecuadamente los archivos y es bastante fácil olvidarse (puede que el programa termine y no se termine de guardar bien). Por eso, preferimos abrir los archivos con el comando with de la siguiente forma.

with open(nombre\_archivo, 'rt') as file:

# Usá el archivo `file`

...comandos que usan el archivo

# No hace falta cerrarlo explícitamente

...comandos que no usan el archivo

Esto cierra automáticamente el archivo cuando se termina de ejecutar el bloque indentado.

Observación: En algunos sistemas operativos es probable que le tengas que especificar el encoding agregando encoding='utf8' como parámetro al comando open.

Comandos usuales para leer un archivo

Para leer un archivo entero, todo de una, como cadena:

with open('foo.txt', 'rt') as file:

data = file.read()

# `data` es una cadena con \*todo\* el texto en `foo.txt`

Para leer línea por línea iterativamente:

with open(nombre\_archivo, 'rt') as file:

for line in file:

# Procesar la línea

Comandos usuales para escribir un archivo

Para escribir cadenas:

with open('outfile', 'wt') as out:

out.write('Hello World\n')

...

También podés simplemente redireccionar la salida del print (de la pantalla a un archivo).

with open('outfile', 'wt') as out:

print('Hello World', file=out)

...

Ejercicios

Estos ejercicios usan el archivo Data/camion.csv. El archivo contiene una lista de líneas con información sobre los cajones de fruta cargados en un camión. Suponemos que estás trabajando en el directorio ejercicios\_python/ del curso. Si no estás segure, podés pedirle al Python que te diga dónde está trabajando con este comando:

>>> import os

>>> os.getcwd()

'/Users/profe/Desktop/ejercicios\_python/Clase02/' # La salida va a cambiar

>>>

Ejercicio 2.1: Preliminares sobre lectura de archivos

Primero, tratá de leer el archivo entero de una en una larga cadena:

>>> with open('../Data/camion.csv', 'rt') as f:

data = f.read()

>>> data

'nombre,cajones,precio\n"Lima",100,32.20\n"Naranja",50,91.10\n"Caqui",150,83.44\n"Mandarina",200,51.23\n"Durazno",95,40.37\n"Mandarina",50,65.10\n"Naranja",100,70.44\n'

>>> print(data)

nombre,cajones,precio

"Lima",100,32.20

"Naranja",50,91.10

"Caqui",150,83.44

"Mandarina",200,51.23

"Durazno",95,40.37

"Mandarina",50,65.10

"Naranja",100,70.44

>>>

En el ejemplo de arriba podrás observar que Python tiene dos modos de salida. En el primero escribiste data en el intérprete y Python mostró la representación cruda de la cadena, incluyendo comillas y códigos de escape. Cuando escribiste print(data), en cambio, obtuviste la salida formateada de la cadena.

Leer un archivo entero y cargarlo en memoria todo de una vez parece simple, pero sólo tiene ventajas si el archivo es pequeño. Si estás trabajando con archivos enormes es mejor procesar las líneas de tu archivo una a una.

Para leer una archivo línea por línea, usá un ciclo for como éste:

>>> with open('../Data/camion.csv', 'rt') as f:

for line in f:

print(line, end = '')

nombre,cajones,precio

"Lima",100,32.2

"Naranja",50,91.1

...

>>>

En ese código, las líneas son leídas una por una hasta el final del archivo, cuando el ciclo se termina.

En ciertas ocasiones, puede pasar que quieras leer una sola línea de texto (por ejemplo, querés saltearte la primera línea del archivo que contiene los nombres de las columnas).

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

>>> headers = next(f)

>>> headers

'nombre,cajones,precio\n'

>>> for line in f:

print(line, end = '')

"Lima",100,32.20

"Naranja",50,91.10

...

>>> f.close()

>>>

El comando next() devuelve la siguiente línea de texto en el archivo. Sin embargo, sólo para que sepas, los ciclos for usan el método next() para obtener sus datos. Por lo tanto, típicamente no forzás un llamado extra a next() salvo que explícitamente quieras saltear o leer una línea particular como en nuestro caso de acá abajo.

Una vez que estés leyendo un archivo línea a línea, podés hacer otras operaciones, como separar los datos dentro de una línea con el método split(). Por ejemplo, probá esto:

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

>>> headers = next(f).split(',')

>>> headers

['nombre', 'cajones', 'precio\n']

>>> for line in f:

row = line.split(',')

print(row)

['"Lima"', '100', '32.20\n']

['"Naranja"', '50', '91.10\n']

...

>>> f.close()

Observación: En estos ejemplos tuvimos que llamar a f.close() explícitamente porque no estamos trabajando con el comando with.

Ejercicio 2.2: Lectura de un archivo de datos

Ahora que sabés leer un archivo, escribamos un programa que haga un cálculo simple con los datos leídos.

Las columnas en camion.csv corresponden a un nombre de fruta, una cantidad de cajones cargados en el camión, y un precio de compra por cada cajón de ese grupo. Escribí un programa llamado costo\_camion.py que abra el archivo, lea las líneas, y calcule el precio pagado por los cajones cargados en el camión.

Ayuda: para interpretar un string s como un número entero, usá int(s). Para leerlo como punto flotante, usá float(s).

Tu programa debería imprimir una salida como la siguiente:

Costo total 47671.15

Acordate de guardar tu archivo en el directorio Clase02; vamos a volver a trabajar sobre él.

Ejercicio 2.3: Precio de la naranja

El archivo Data/precios.csv contiene una serie de líneas con precios de venta de cajones en el mercado al que va el camión. El archivo se ve así:

"Lima",40.22

"Uva",24.85

"Ciruela",44.85

"Cereza",11.27

"Frutilla",53.72

...

Escribí un código que abra el archivo Data/precios.csv, busque el precio de la naranja y lo imprima en pantalla.

>>> f = open('../Data/precios.csv', 'rt')

...

>>> f.close()

El precio de la naranja es: 106.28

Ejercicio 2.4: Archivos comprimidos

¿Que pasaría si quisiéramos leer un archivo comprimido con gzip, por ejemplo? La función primitiva de Python open() no hace esa tarea. Pero hay un módulo de la biblioteca de Python llamado gzip que lee archivos comprimidos.

Probalo:

>>> import gzip

>>> with gzip.open('../Data/camion.csv.gz', 'rt') as f:

for line in f:

print(line, end = '')

... mirá la salida ...

>>>

Observación: La inclusión del modo 'rt' es crítica acá. Si te lo olvidás, vas a estar leyendo cadenas de bytes en lugar de cadenas de caracteres.

Comentario: ¿No deberíamos estar usando Pandas para esto?

Es frecuente que les estudiantes que conocen un poco más de Python rápidamente señalen que hay módulos como Pandas que tienen, entre muchas otras funcionalidades, la posibilidad de leer archivos CSV en una sola instrucción. Es verdad, y funcionan muy bien. Sin embargo, este no es un curso sobre Pandas. Si bien más adelante veremos algo de esta biblioteca, lo que nos interesa en este momento es aprender a manejar archivos directamente. Estamos trabajando con archivos CSV porque es un formato sencillo que es muy útil conocer, pero es principalmente una excusa para mostrar cómo Python maneja archivos de texto. En resumen, cuando tengas que trabajar con datos, definitivamente usá Pandas. Pero para aprender a manejar archivos vamos a seguir usando las funciones básicas de Python.

2.2 Funciones

A medida que tus programas se vuelven más largos y complejos, vas a necesitar organizarte. En esta sección vamos a introducir brevemente funciones y módulos de la biblioteca así como también la administración de errores y excepciones.

Funciones a medida

Usá funciones para encapsular código que quieras reutilizar. El siguiente ejemplo muestra una definición de una función:

def sumcount(n):

'''

Devuelve la suma de los primeros n enteros

'''

total = 0

while n > 0:

total += n

n -= 1

return total

Para llamar a una función:

a = sumcount(100)

Una función es una serie de instrucciones que realiza una tarea y devuelve un resultado. La palabra return es necesaria para explicitar el valor de retorno de la función.

Funciones de la biblioteca

Python trae una gran biblioteca estándar. Los módulos de esta biblioteca se cargan usando import. Por ejemplo:

import math

x = math.sqrt(10)

import urllib.request

u = urllib.request.urlopen('http://www.python.org/')

data = u.read()

Vamos a estudiar bibliotecas y módulos en detalle más adelante.

Errores y excepciones

Las funciones informan los errores como excepciones. Dado que una excepción interrumpe la ejecución de una función, la misma puede generar que todo el programa se detenga si no es administrada adecuadamente.

Probá por ejemplo lo siguiente en tu intérprete:

>>> int('N/A')

Traceback (most recent call last):

File "<stdin>", line 1, in <módulo>

ValueError: invalid literal for int() with base 10: 'N/A'

>>>

Para poder entender qué pasó (debuguear), el mensaje describe cuál fue el problema, dónde ocurrió y un poco de la historia (traceback) de los llamados que terminaron en este error.

Atrapar y administrar excepciones

Las excepciones pueden ser atrapadas y administradas. Para atrapar una excepción, se usan los comandos try - except. Podés probar el siguiente fragmento de código pegándolo en un archivo foo.py :

numero\_valido=False

while not numero\_valido:

try:

a = input('Ingresá un número entero: ')

n = int(a)

numero\_valido = True

except ValueError:

print('No es válido. Intentá de nuevo.')

print(f'Ingresaste {n}.')

Si en este ejemplo el usuario ingresa por ejemplo una letra, el comando n = int(a) genera una excepción de tipo ValueError: el comando numero\_valido = True no se ejecuta, la excepción es atrapada por el except ValueError y el ciclo se repite. Probalo ingresando letras, números con decimales y números enteros. Probá también qué ocurre si querés salir sin ingresar nada generando una excepción presionando las teclas Ctrl+C. Leé el mensaje que describe lo ocurrido: Ctrl+C genera una excepción de tipo KeyboardInterrupt que no es atrapada.

Si no especificamos el tipo de excepción que queremos atrapar, vamos a terminar atrapando todas la excepciones. Probá lo mismo que antes pero con este código.

numero\_valido=False

while not numero\_valido:

try:

a = input('Ingresá un número entero: ')

n = int(a)

numero\_valido = True

except:

print('No es válido. Intentá de nuevo.')

print(f'Ingresaste {n}.')

Deberías observar una diferencia: al presionar las teclas Ctrl+C la excepción KeyboardInterrupt sí es atrapada y no se termina el ciclo hasta no ingresar un número entero.

Suele ser difícil saber exactamente qué tipo de errores pueden ocurrir por adelantado. Para bien o para mal, la administración de excepciones suele ir creciendo a medida que un programa va generando errores inesperados (al mejor estilo: "Uh! Me olvidé de que podía pasar esto. Deberíamos preverlo y administrarlo adecuadamente para la próxima").

Generar excepciones

Para generar una excepción (también diremos levantar una excepción, porque más cercano al término inglés "raise"), se usa el comando raise. Por ejemplo, si tenemos el siguiente código en el archivo foo.py:

raise RuntimeError('¡Qué moco!')

Al correrlo va a detener la ejecución y permite rastrear la excepción leyendo el mensaje de error que imprime.

bash $ python3 foo.py

Traceback (most recent call last):

File "foo.py", line 21, in <módulo>

raise RuntimeError("¡Qué moco!")

RuntimeError: ¡Qué moco!

Alternativamente, esa excepción puede ser atrapada por un bloque try-except, pudiendo de esta forma evitar que el programa termine.

Ejercicios

Ejercicio 2.5: Definir una función

Probá primero definir una función simple:

>>> def saludar(nombre):

'Saluda a alguien'

print('Hola', nombre)

>>> saludar('Guido')

Hola Guido

>>> saludar('Paula')

Hola Paula

>>>

Si la primera instrucción de una función es una cadena, sirve como documentación de la función. Probalo escribiendo help(saludar) para ver cómo la muestra.

Ejercicio 2.6: Transformar un script en una función

Transformá el programa costo\_camion.py (que escribiste en el Ejercicio 2.2 de la sección anterior) en una función costo\_camion(nombre\_archivo). Esta función recibe un nombre de archivo como entrada, lee la información sobre los cajones que cargó el camión y devuelve el costo de la carga de frutas como una variable de punto flotante.

Para usar tu función, cambiá el programa de forma que se parezca a esto:

def costo\_camion(nombre\_archivo):

...

# Tu código

...

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

print('Costo total:', costo)

Cuando ejecutás tu programa, deberías ver la misma salida impresa que antes. Una vez que lo hayas corrido, podés llamar interactivamente a la función escribiendo esto:

bash $ python3 -i costo\_camion.py

Esto va a ejecutar el código en el programa y dejar abierto el intérprete interactivo.

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

47671.15

>>>

Es útil para testear y debuguear poder interactuar interactivamente con tu código.

Ejercicio 2.7: Buscar precios

A partir de lo que hiciste en el Ejercicio 2.3, escribí una función buscar\_precio(fruta) que busque en archivo ../Data/precios.csv el precio de determinada fruta (o verdura) y lo imprima en pantalla. Si la fruta no figura en el listado de precios, debe imprimir un mensaje que lo indique.

>>> buscar\_precio('Frambuesa')

El precio de un cajón de Frambuesa es: 34.35

>>> buscar\_precio('Kale')

Kale no figura en el listado de precios.

Guardá este código en un archivo buscar\_precio.py para entregar al final de la clase.

Ejercicio 2.8: Administración de errores

Probá correr la siguiente función ingresando tu edad real, una edad escrita con letras (como "ocho") y una edad negativa (-3):

def preguntar\_edad(nombre):

edad = int(input(f'ingresá tu edad {nombre}: '))

if edad<0:

raise ValueError('La edad no puede ser negativa.')

return edad

Ahora probá este ejemplo que atrapa la excepción generada con raise y continúa la ejecución con la siguiente persona.

for nombre in ['Pedro','Juan','Caballero']:

try:

edad = preguntar\_edad(nombre)

print(f'{nombre} tiene {edad} años.')

except ValueError:

print(f'{nombre} no ingresó una edad válida.')

Vamos a usar estas ideas aplicadas al procesamiento de un archivo CSV. ¿Qué pasa si intentás usar la función costo\_camion() con un archivo que tiene datos faltantes?

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

Traceback (most recent call last):

File "<stdin>", line 1, in <módulo>

File "costo\_camion.py", line 11, in costo\_camion

ncajones = int(fields[1])

ValueError: invalid literal for int() with base 10: ''

>>>

El programa termina con un error. A esta altura tenés que tomar una decisión. Para que el programa funcione podés editar el archivo CSV de entrada de manera de corregirlo (borrando líneas o adecuando la información) o podés modificar el código para que maneje las líneas incorrectas de alguna manera.

Modificá el programa costo\_camion.py para que atrape la excepción con un bloque try - except, imprima un mensaje de aviso (warning) y continúe procesando el resto del archivo.

Vamos a profundizar en la administración de errores en las próximas clases.

Ejercicio 2.9: Funciones de la biblioteca

Python viene con una gran biblioteca estándar de funciones útiles. En este caso el módulo csv podría venirnos muy bien. Podés usarlo cada vez que tengas que leer archivos CSV. Acá va un ejemplo de cómo funciona.

>>> import csv

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

>>> rows = csv.reader(f)

>>> headers = next(rows)

>>> headers

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

>>> for row in rows:

print(row)

['Lima', '100', '32.2']

['Naranja', '50', '91.1']

['Caqui', '150', '103.44']

['Mandarina', '200', '51.23']

['Durazno', '95', '40.37']

['Mandarina', '50', '65.1']

['Naranja', '100', '70.44']

>>> f.close()

>>>

Una cosa buena que tiene el módulo csv es que maneja solo una gran variedad de detalles de bajo nivel como el problema de las comillas, o la separación con comas de los datos. En la salida del último ejemplo podés ver que el lector ya sacó las comillas dobles de los nombres de las frutas de la primera columna.

Modificá tu programa costo\_camion.py para que use el módulo csv para leer los archivos CSV y probalo en un par de los ejemplos anteriores.

Ejercicio 2.10: Ejecución desde la línea de comandos con parámetros

En el programa costo\_camion.py, el nombre del archivo de entrada '../Data/camion.csv' fue escrito en el código.

# costo\_camion.py

import csv

def costo\_camion(nombre\_archivo):

...

# Tu código

...

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

print('Costo total:', costo)

Esto está bien para ejercitar, pero en un programa real probablemente no harías eso ya que querrías una mayor flexibilidad. Una posibilidad es pasarle al programa el nombre del archivo que querés procesar como un parámetro cuando lo llamás desde la línea de comandos.

Copiá el contenido de costo\_camion.py a un nuevo archivo llamado camion\_commandline.py que incorpore la lectura de parámetros por línea de comando según la sugerencia del siguiente ejemplo:

# camion\_commandline.py

import csv

import sys

def costo\_camion(nombre\_archivo):

...

# Tu código

...

if len(sys.argv) == 2:

nombre\_archivo = sys.argv[1]

else:

nombre\_archivo = '../Data/camion.csv'

costo = costo\_camion(nombre\_archivo)

print('Costo total:', costo)

sys.argv es una lista que contiene los argumentos que le pasamos al script al momento de llamarlo desde la línea de comandos (si es que le pasamos alguno). Por ejemplo, desde una terminal de Unix (en Windows es similar), para correr nuestro programa y que procese el mismo archivo podríamos escribir:

bash $ python3 camion\_commandline.py ../Data/camion.csv

Costo total: 47671.15

bash $

O con el archivo missing.csv:

bash $ python3 camion\_commandline.py ../Data/missing.csv

...

Costo total: 30381.15

bash $

Si no le pasamos ningún archivo, va a mostrar el resultado para camion.csv porque lo indicamos con la línea nombre\_archivo = '../Data/camion.csv'.

Guardá tu programa en el archivo camion\_commandline.py para entregar al final de la clase.

2.3 Tipos y estructuras de datos

Esta sección introduce dos estructuras de datos elementales: las tuplas y los diccionarios.

Tipos de datos primitivos

Python tiene pocos tipos primitivos de datos.

Números enteros

Números de punto flotante

Cadenas de texto

Algo ya sabemos sobre estos tipos de datos por el capítulo anterior.

Tipo None

email\_address = None

None suele utilizarse como un comodín para reservar el lugar para un valor opcional o faltante. En los condicionales, evalúa como False.

if email\_address:

send\_email(email\_address, msg)

Estructuras de datos

Los programas reales tienen datos más complejos que los que podemos almacenar en los tipos primitivos. Por ejemplo, información sobre un pedido de frutas:

100 cajones de Manzanas a $490.10 cada uno

Podemos ver esto como un "objeto" con tres partes:

Nombre del símbolo ("Manzanas", una cadena)

Número o cantidad (100, un entero)

Precio (490.10, un flotante)

Tuplas

Una tupla es una colección con valores agrupados juntos.

Ejemplo:

s = ('Manzanas', 100, 490.1)

Las tuplas suelen usarse para representar registros o estructuras simples. Típicamente, una tupla representa un solo objeto con múltiples partes. Una analogía posible es la siguiente: Una tupla es como una fila de una base de datos.

Los contenidos de una tupla están ordenados (como en una lista).

s = ('Manzana', 100, 490.1)

nombre = s[0] # 'Manzana'

cantidad = s[1] # 100

precio = s[2] # 490.1

El contenido de las tuplas no puede ser modificado.

>>> s[1] = 75

TypeError: object does not support item assignment

Podés, sin embargo, hacer una nueva tupla basada en el contenido de otra, que no es lo mismo que modificar el contenido.

s = (s[0], 75, s[2])

Empaquetar tuplas

Las tuplas suelen usarse para empaquetar información relacionada en una sola entidad.

s = ('Manzanas', 100, 490.1)

Una tupla puede ser pasada de un lugar a otro de un programa como un solo objeto.

Desempaquetar tuplas

Para usar una tupla en otro lado, debemos desempaquetar su contenido en diferentes variables.

fruta, cajones, precio = s

print('Costo:', cajones \* precio)

El número de variables a la izquierda debe ser consistente con la estructura de la tupla.

nombre, cajones = s # ERROR

Traceback (most recent call last):

...

ValueError: too many values to unpack

Tuplas vs. Listas

Las tuplas parecieran ser listas de solo-lectura. Sin embargo, las tuplas suelen usarse para un solo ítem que consiste de múltiples partes mientras que las listas suelen usarse para una colección de diferentes elementos, típicamente del mismo tipo.

record = ('Manzanas', 100, 490.1) # Una tupla representando un registro dentro de un pedido de frutas

symbols = [ 'Manzanas', 'Peras', 'Mandarinas' ] # Una lista representando tres frutas diferentes.

Diccionarios

Un diccionario es una función que manda claves en valores. Las claves sirven como índices para acceder a los valores.

s = {

'fruta': 'Manzana',

'cajones': 100,

'precio': 490.1

}

Operaciones usuales

Para obtener el valor almacenado en un diccionario usamos las claves.

>>> print(s['fruta'], s['cajones'])

Manzanas 100

>>> s['precio']

490.10

>>>

Para agregar o modificar valores, simplemente asignamos usando la clave.

>>> s['cajones'] = 75

>>> s['fecha'] = '6/8/2020'

>>>

para borrar un valor, usamos el comando del.

>>> del s['fecha']

>>>

¿Por qué diccionarios?

Los diccionarios son útiles cuando hay muchos valores diferentes y esos valores pueden ser modificados o manipulados. Dado que el acceso a los elementos se hace por clave, no es necesario recordar una posición para cierto dato, lo que muchas veces cumple un objetivo fundamental: hacer que el código sea más legible (y con esto menos propenso a errores).

s['precio'] # diccionario

# vs

s[2] # lista

Ejercicios

Anteriormente escribiste un programa que leía el archivo camion.csv usando el módulo csv para leer el archivo fila por fila.

>>> import csv

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

>>> filas = csv.reader(f)

>>> next(filas)

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

>>> fila = next(filas)

>>> fila

['Lima', '100', '32.2']

>>>

A veces, además de leerlo, queremos hacer otras cosas con el archivo CSV, como por ejemplo usar los datos que contiene para hacer un cálculo. Lamentablemente una fila de datos en crudo no es suficiente para operar aritméticamente. Vamos a querer interpretar los elementos de la fila de datos de alguna manera particular, convirtiéndolos a otro tipo de datos que resulte más adecuado para trabajar. Es frecuente además de convertir los elementos de las filas, transformar las filas enteras en tuplas o diccionarios.

Ejercicio 2.11: Tuplas

En el intérprete interactivo, creá la siguiente tupla que representa la fila de antes, pero con las columnas numéricas pasadas a formatos adecuados:

>>> t = (fila[0], int(fila[1]), float(fila[2]))

>>> t

('Lima', 100, 32.2)

>>>

A partir de esta tupla, ahora podés calcular el costo total multiplicando cajones por precio:

>>> cost = t[1] \* t[2]

>>> cost

3220.0000000000005

>>>

¿Qué pasó? ¿Qué hace ese 5 al final?

Este error no es un problema de Python, sino de la forma en la que la máquina representa los números de punto flotante. Así como en base 10 no podemos escribir un tercio de manera exacta, en base 2 escribir un quinto requiere infinitos dígitos. Al usar una representación finita (una cantidad acotada de dígitos) la máquina redondea los números. La aritmética de punto flotante no es exacta.

Esto pasa en todos los lenguajes de programación que usan punto flotante, pero en muchos casos estos pequeños errores quedan ocultos al imprimir. Por ejemplo:

>>> print(f'{cost:0.2f}')

3220.00

>>>

Las tuplas son de sólo lectura. Verificalo tratando de cambiar el número de cajones a 75.

>>> t[1] = 75

Traceback (most recent call last):

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

TypeError: 'tuple' object does not support item assignment

>>>

Aunque no podés cambiar al tupla, sí podés reemplazar la tupla por una nueva.

>>> t = (t[0], 75, t[2])

>>> t

('Lima', 75, 32.2)

>>>

Siempre que reasignes una variable como recién lo hiciste con t, el valor anterior de la variable se pierde. Aunque la asignación de arriba pueda parecer como que estás modificando la tupla, en realidad estás creando una nueva tupla y tirando la vieja.

Las tuplas muchas veces se usan para empaquetar y desempaquetar valores dentro de variables. Probá esto:

>>> nombre, cajones, precio = t

>>> nombre

'Lima'

>>> cajones

75

>>> precio

32.2

>>>

Tomá las variables de arriba y empaquetalas en una tupla.

>>> t = (nombre, 2\*cajones, precio)

>>> t

('Lima', 150, 32.2)

>>>

Ejercicio 2.12: Diccionarios como estructuras de datos

Una alternativa a la tupla es un diccionario.

>>> d = {

'nombre' : fila[0],

'cajones' : int(fila[1]),

'precio' : float(fila[2])

}

>>> d

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

>>>

Calculá el costo total de este lote:

>>> cost = d['cajones'] \* d['precio']

>>> cost

3220.0000000000005

>>>

Compará este ejemplo con el mismo cálculo hecho con tuplas más arriba. Cambiá el número de cajones a 75.

>>> d['cajones'] = 75

>>> d

{'nombre': 'Lima', 'cajones': 75, 'precio': 32.2 }

>>>

A diferencia de las tuplas, los diccionarios se pueden modificar libremente. Agregá algunos atributos:

>>> d['fecha'] = (14, 8, 2020)

>>> d['cuenta'] = 12345

>>> d

{'nombre': 'Lima', 'cajones': 75, 'precio':32.2, 'fecha': (14, 8, 2020), 'cuenta': 12345}

>>>

Ejercicio 2.13: Más operaciones con diccionarios

Si usás el comando for para iterar sobre el diccionario, obtenés las claves:

>>> for k in d:

print('k =', k)

k = nombre

k = cajones

k = precio

k = fecha

k = cuenta

>>>

Probá esta variante:

>>> for k in d:

print(k, '=', d[k])

nombre = 'Lima'

cajones = 75

precio = 32.2

fecha = (14, 8, 2020)

cuenta = 12345

>>>

Una manera más elegante de trabajar con claves y valores simultáneamente es usar el método items(). Esto te devuelve una lista de tuplas de la forma (clave,valor) sobre la que podés iterar.

>>> items = d.items()

>>> items

dict\_items([('nombre', 'Lima'), ('cajones', 75), ('precio', 32.2), ('fecha', (14, 8, 2020))])

>>> for k, v in d.items():

print(k, '=', v)

nombre = Lima

cajones = 75

precio = 32.2

fecha = (14, 8, 2020)

>>>

Si pasás un diccionario a una lista, obtenés sus claves.

>>> list(d)

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

>>>

También podés obtener todas las claves del diccionario usando el método keys():

>>> claves = d.keys()

>>> claves

dict\_keys(['nombre', 'cajones', 'precio', 'fecha', 'cuenta'])

>>>

Si tenés tuplas como en items podés crear un diccionario usando la función dict(). Probá esto:

>>> nuevos\_items = [('nombre', 'Manzanas'), ('cajones', 100), ('precio', 490.1), ('fecha', (13, 8, 2020))]

>>> nuevos\_items

[('nombre', 'Manzanas'), ('cajones', 100), ('precio', 490.1), ('fecha', (13, 8, 2020))]

>>> d = dict(nuevos\_items)

>>> d

{'nombre': 'Manzanas', 'cajones': 100, 'precio': 490.1, 'fecha': (13, 8, 2020)}

Ejercicio 2.14: Diccionario geringoso.

Construí una función que, a partir de una lista de palabras, devuelva un diccionario geringoso. Las claves del diccionario deben ser las palabras de la lista y los valores deben ser sus traducciones al geringoso (como en el Ejercicio 1.18). Probá tu función para la lista ['banana', 'manzana', 'mandarina']. Guardá este ejercicio en un archivo diccionario\_geringoso.py para entregar al final de la clase.

2.4 Contenedores

En esta sección trataremos listas, diccionarios y conjuntos.

Panorama

Los programas suelen trabajar con muchos objetos.

Un camión con cajones de fruta

Una tabla de precios de cajones de fruta

En Python hay tres opciones principales para elegir.

Listas. Datos ordenados.

Diccionarios. Datos desordenados.

Conjuntos. Colección desordenada de elementos únicos.

Listas como contenedores

Usá listas cuando el orden de los datos importe. Acordate de que las listas pueden contener cualquier tipo de objeto. Por ejemplo, una lista de tuplas.

camion = [

('Pera', 100, 490.1),

('Naranja', 50, 91.3),

('Limon', 150, 83.44)

]

camion[0] # ('Pera', 100, 490.1)

camion[2] # ('Limon', 150, 83.44)

Construcción de una lista

Cómo armar una lista desde cero.

registros = [] # Empezamos con una lista vacía

# Usamos el .append() para agregar elementos

registros.append(('Pera', 100, 490.10))

registros.append(('Naranja', 50, 91.3))

...

Un ejemplo de cómo cargar registros desde un archivo.

registros = [] # Empezamos con una lista vacía

with open('../Data/camion.csv', 'rt') as f:

next(f) # Saltear el encabezado

for line in f:

row = line.split(',')

registros.append((row[0], int(row[1]), float(row[2])))

Diccionarios como contenedores

Los diccionarios son útiles si vamos a querer buscar rápidamente (por claves). Por ejemplo, un diccionario de precios de cajones.

precios = {

'Pera': 513.25,

'Limon': 87.22,

'Naranja': 93.37,

'Mandarina': 44.12

}

Así podemos buscar datos:

>>> precios['Naranja']

93.37

>>> precios['Pera']

513.25

>>>

Construcción de diccionarios

Ejemplo de armado de un diccionario desde cero.

precios = {} # Empezamos con un diccionario vacío

# Agregamos elementos

precios['Pera'] = 513.25

precios['Limon'] = 87.22

precios['Naranja'] = 93.37

Un ejemplo de cómo armar un diccionario a partir del contenido de un archivo.

precios = {} # Empezamos con un diccionario vacío

with open('../Data/precios.csv', 'rt') as f:

for line in f:

row = line.split(',')

precios[row[0]] = float(row[1])

Nota: Si probás estos comandos en el archivo Data/precios.csv, vas a ver que casi anda. Pero hay una línea en blanco al final que genera un error. Usando lo que ya vimos, en el Ejercicio 2.17 vas a tener que modificar el código para resolver el problema.

Búsquedas en un diccionario

Podés verificar si una clave existe:

if key in d:

# YES

else:

# NO

Claves compuestas

Casi cualquier valor puede usarse como clave en un diccionario de Python. La principal restricción es que una clave debe ser de tipo inmutable. Por ejemplo, tuplas:

feriados = {

(1, 1) : 'Año nuevo',

(1, 5) : 'Día del trabajador',

(13, 9) : "Día del programador",

}

Luego, podemos acceder al diccionario así:

>>> feriados[(1, 5)]

'Día del trabajador'

>>>

Las listas, los conjuntos y los diccionarios no pueden ser usados como claves de diccionarios, porque son mutables.

Conjuntos

Un conjunto es una colección de elementos únicos sin orden y sin repetición.

citricos = { 'Naranja','Limon','Mandarina' }

# Alternativamente podemos escribirlo así:

citricos = set(['Naranja', 'Limon', 'Mandarina'])

Los conjuntos son útiles para evaluar pertenencia.

>>> citricos

set(['Naranja', 'Limon', 'Mandarina'])

>>> 'Naranja' in citricos

True

>>> 'Manzana' in citricos

False

>>>

Los conjuntos también son útiles para eliminar duplicados.

nombres = ['Naranja', 'Manzana', 'Pera', 'Naranja', 'Pera', 'Banana']

unicos = set(nombres)

# unicos = {'Manzana', 'Banana', 'Naranja', 'Pera'}

Más operaciones en conjuntos:

citricos.add('Banana') # Agregar un elemento

citricos.remove('Limon') # Eliminar un elemento

s1 | s2 # Unión de conjuntos s1 y s2

s1 & s2 # Intersección de conjuntos

s1 - s2 # Diferencia de conjuntos

Ejercicios

En estos ejercicios, vas a empezar a construir un programa más largo. Trabajá en el archivo ejercicios\_python/informe.py.

Ejercicio 2.15: Lista de tuplas

El archivo Data/camion.csv contiene la lista de cajones cargados en un camión. En el Ejercicio 2.6 de la sección anterior escribiste una función costo\_camion(nombre\_archivo) que leía el archivo y realizaba un cálculo.

La función debería verse parecida a ésta:

# fragmento de costo\_camion.py

import csv

...

def costo\_camion(nombre\_archivo):

'''Computa el precio total del camion (cajones \* precio) de un archivo'''

total = 0.0

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

rows = csv.reader(f)

headers = next(rows)

for row in rows:

ncajones = int(row[1])

precio = float(row[2])

total += ncajones \* precio

return total

...

Usando este código como guía, creá un nuevo archivo informe.py. En este archivo, definí una función leer\_camion(nombre\_archivo) que abre un archivo con el contenido de un camión, lo lee y devuelve la información como una lista de tuplas. Para hacerlo vas a tener que hacer algunas modificaciones menores al código de arriba.

Primero, en vez de definir total = 0, tenés que empezar con una variable que empieza siendo una lista vacía Por ejemplo:

camion = []

Después, en vez de sumar el costo, tenés que pasar cada fila a una tupla igual a como lo hiciste en el último ejercicio, y agregarla a la lista. Por ejemplo:

for row in rows:

lote = (row[0], int(row[1]), float(row[2]))

camion.append(lote)

Por último, la función debe devolver la lista camion.

Experimentá con tu función interactivamente (acordate de que primero tenés que correr el programa informe.py en el intérprete):

Ayuda: Usá -i para ejecutar un archivo en la terminal y quedar en el intérprete

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

>>> camion

[('Lima', 100, 32.2), ('Naranja', 50, 91.1), ('Limon', 150, 83.44), ('Mandarina', 200, 51.23),('Durazno', 95, 40.37), ('Mandarina', 50, 65.1), ('Naranja', 100, 70.44)]

>>>

>>> camion[0]

('Lima', 100, 32.2)

>>> camion[1]

('Naranja', 50, 91.1)

>>> camion[1][1]

50

>>> total = 0.0

>>> for s in camion:

total += s[1] \* s[2]

>>> print(total)

47671.15

>>>

Esta lista de tuplas que creaste es muy similar a un array o matriz bidimensional. Por ejemplo, podés acceder a una fila específica y columna específica usando una búsqueda como camion[fila][columna] donde fila y columna son números enteros.

También podés reescribir el último ciclo for usando un comando como éste:

>>> total = 0.0

>>> for nombre, cajones, precio in camion:

total += cajones\*precio

>>> print(total)

47671.15

>>>

Observación: la instrucción += es una abreviación. Poner a += b es equivalente a poner a = a + b

Ejercicio 2.16: Lista de diccionarios

Tomá la función que escribiste en el ejercicio anterior y modificala para representar cada cajón del camión con un diccionario en vez de una tupla. En este diccionario usá los campos "nombre", "cajones" y "precio" para representar las diferentes columnas del archivo de entrada.

Experimentá con esta función nueva igual que en el ejercicio anterior.

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

>>> camion

[{'nombre': 'Lima', 'cajones': 100, 'precio': 32.2}, {'nombre': 'Naranja', 'cajones': 50, 'precio': 91.1}, {'nombre': 'Limon', 'cajones': 150, 'precio': 83.44}, {'nombre': 'Mandarina', 'cajones': 200, 'precio': 51.23}, {'nombre': 'Durazno', 'cajones': 95, 'precio': 40.37}, {'nombre': 'Mandarina', 'cajones': 50, 'precio': 65.1}, {'nombre': 'Naranja', 'cajones': 100, 'precio': 70.44}]

>>> camion[0]

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

>>> camion[1]

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

>>> camion[1]['cajones']

50

>>> total = 0.0

>>> for s in camion:

total += s['cajones']\*s['precio']

>>> print(total)

47671.15

>>>

Fijate que acá los distintos campos para cada entrada se acceden a través de claves en vez de la posición en la lista. Muchas veces preferimos esto porque el código resulta más fácil de leer. Tanto para otres como para nosotres en el futuro.

Mirar diccionarios y listas muy grandes puede ser un lío. Para limpiar el output para debuguear, probá la función pprint (Pretty-print) que le da un formato más sencillo de interpretar.

>>> from pprint import pprint

>>> pprint(camion)

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

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

{'nombre': 'Limon', 'precio': 83.44, 'cajones': 150},

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

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

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

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

>>>

Ejercicio 2.17: Diccionarios como contenedores

Los diccionarios son útiles si querés buscar elementos usando índices que no sean números enteros. En la terminal de Python, jugá con un diccionario:

>>> precios = {}

>>> precios['Naranja'] = 92.45

>>> precios['Mandarina'] = 45.12

>>> precios

... mirá el resultado ...

>>> precios['Naranja']

92.45

>>> precios['Manzana']

... mirá el resultado ...

>>> 'Manzana' in precios

False

>>>

En el Ejercicio 2.7 escribiste una función que busca el precio de una determinada fruta o verdura en el archivo ../Data/precios.csv. Esto es útil para saber sobre un producto en particular, pero si necesitás tener los precios de toda la mercadería, no resulta práctico abrir y cerrar el archivo para consultar cada precio. Por eso ahora te proponemos generar un diccionario que contenga todos los precios. En este diccionario, podés consultar luego los precios de cada producto.

Escribí una función leer\_precios(nombre\_archivo) que a partir de un conjunto de precios como éste arme un diccionario donde las claves sean los nombres de frutas y verduras, y los valores sean los precios por cajón.

Para hacerlo, empezá con un diccionario vacío y andá agregándole valores igual a como hiciste antes, pero ahora esos valores los vas leyendo del archivo.

Vamos a usar esta estructura de datos para buscar rápidamente los precios de las frutas y verduras.

Un par de consejos: Usá el módulo csv igual que antes.

>>> import csv

>>> f = open('../Data/precios.csv', 'r')

>>> rows = csv.reader(f)

>>> for row in rows:

print(row)

['Lima', '40.22']

['Uva', '24.85']

...

[]

>>>

El archivo Data/precios.csv puede tener líneas en blanco, esto te puede traer complicaciones. Observá que arriba figura una lista vacía (la última), porque la última línea del archivo no tenía datos.

Puede suceder que esto haga que tu programa termine con una excepción. Usá los comandos try y except para evitar el problema. Para pensar: ¿Sería mejor prevenir estos problemas con el comando if en vez de try y except?

Una vez que hayas escrito tu función leer\_precios(), testeala interactivamente para asegurarte de que funciona bien:

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

>>> precios['Naranja']

106.28

>>> precios['Mandarina']

80.89

>>>

Ejercicio 2.18: Balances

Supongamos que los precios en camion.csv son los precios pagados al productor de frutas mientras que los precios en precios.csv son los precios de venta en el lugar de descarga del camión.

Ahora vamos calcular el balance del negocio: juntá todo el trabajo que hiciste recién en tu programa informe.py (usando las funciones leer\_camion() y leer\_precios()) y completá el programa para que con los precios del camión (Ejercicio 2.16) y los de venta en el negocio (Ejercicio 2.17) calcule lo que costó el camión, lo que se recaudó con la venta, y la diferencia. ¿Hubo ganancia o pérdida? El programa debe imprimir por pantalla un balance con estos datos.

2.5 Cierre de la clase

En esta clase comenzamos a trabajar con funciones y estructuras de datos.

Te recordamos que leas el código de honor 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 estás de acuerdo con este texto. En caso contrario no envíes tus archivos y contactate con les docentes.

Para cerrar esta clase te pedimos dos cosas:

Que recopiles las soluciones de los siguientes ejercicios:

El archivo buscar\_precios.py del Ejercicio 2.7.

El archivo costo\_camion.py del Ejercicio 2.9.

El archivo camion\_commandline.py del Ejercicio 2.10.

El archivo diccionario\_geringoso.py del Ejercicio 2.14.

El archivo informe.py del Ejercicio 2.18.

Que completes este formulario 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 que entregues como mínimo tres archivos de los 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.