Clase 10. Buenas prácticas 2.0 (Python tricks)

TRABAJADORES INFORMÁTICOS
organizados en AGC

Ministerio de Trabajo, Empleo y Seguridad Social
Argentina

Intercambio de valores in-place

Se pueden intercambiar los valores de dos variables sin necesidad de una tercera.

```
>>> x, y = 7, 56
>>> print(x, y)
7 56
>>> y, x = x, y
>>> print(x, y)
56 7
```

join: Función para unir strings de un vector en un único string

La función *join* nos permite unir todos los strings de un vector utilizando un string como conector.

Sintaxis: *stringConector.join(vector)*
Resultado:
*“elemento1**stringConector**elemento2**stringConector**elemento3...”*

```
>>> listaDeFrutas = ["anana", "manzanas",
"mandarinas"]
>>> " ".join(listaDeFrutas)
'anana manzanas mandarinas'
>>> lista = ", ".join(listaDeFrutas)
>>> print("Lista de compras:", lista)
Lista de compras: anana, manzanas, mandarinas
```

split: Función para separar un único string en múltiples

La función *split* nos permite separar un string en múltiples strings indicando un separador

Sintaxis: *string.split(separador)*
Resultado: [elemento1, elementos2, elemento3, …]

```
>>> frase = "Esto es una frase de ejemplo"
>>> frase.split(" ")
['Esto', 'es', 'una', 'frase', 'de', 'ejemplo']
>>> frase.split("a")
['Esto es un', ' fr', 'se de ejemplo']
>>> frase.split("separadorInventado")
['Esto es una frase de ejemplo']
```

Ejercicio

Tiempo ⏱️

- Escribir una función que reciba una frase y reemplace los espacios por “@” utilizando las funciones anteriormente mencionadas

slices: Extraer un sub-vector de otro

La función *slice* nos sirve para extraer un sub-vector de otro.
Sintaxis: *vector[slice(start, stop, step)]*

- ***start***: (opcional) Índice para comenzar a seleccionar. Default: None
- ***stop***: Va a seleccionar hasta *stop-1*
- ***step***: (opcional) Incremento entre selección. Default: None

También se puede escribir como: *vector[start:stop:step]*

```
>>> vector = [1, 2, 3, 4, 5, 6, 7, 8, 9, 20]
>>> vector[slice(0, 10)]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 20]
>>> vector[slice(0, 3)]
[1, 2, 3]
>>> vector[slice(2, 3)]
[3]
>>> vector[slice(0, 8, 2)]
[1, 3, 5, 7]
>>> vector[1:7]
[2, 3, 4, 5, 6, 7]
>>> vector[1:7: 3]
[2, 5]
>>> vector[::2]
[1, 3, 5, 7, 9]
>>> vector[:4:]
[1, 2, 3, 4]
```

slices: Extraer un sub-vector de otro

Cualquiera de los 3 parámetros puede ser negativo.

```
# Los últimos dos elementos
>>> vector[-2:]
[9, 20]
# Todos los elementos exceptuando los últimos dos
>>> vector[:-2]
[1, 2, 3, 4, 5, 6, 7, 8]
# Todos los elementos pero en orden inverso
>>> vector[::-1]
[20, 9, 8, 7, 6, 5, 4, 3, 2, 1]

# Recuerden que un string se puede ver como un vector
# donde cada posición es un caracter,
>>> "Este es un string que voy a dar vuelta"[::-1]
'atleuv rad a yov euq gnirts nu se etsE'
```

Ejercicio

Tiempo ⏱️

Escribir una función que retorne una lista que sea la reversa de otra lista pasada por parámetros sin los elementos de las posiciones múltiplos de 3

Concatenar condiciones

Las condiciones se puede agrupar.
Ejemplo: *5 < n and n < 10* es lo mismo que *5 < n < 10*

```
>>> n = 10
>>> 5 < n < 10
False
>>> 6 < n >= 10
True
```

Retornar múltiples valores de una función

En Python, las funciones pueden retornar más de un valor. No todos los lenguajes lo permiten. En python se realiza retornando una tupla.

```
def f():
  return 10, [], “Hola”

>>> n, lista, texto = f()
>>> print(n)
10
>>> print(lista)
[]
>>> print(text)
“Hola”
```

Otras funciones útiles

- *min:* Busca el menor número de un vector.
- *max:* Busca el mayor número en un vector.
- *len*: Nos dice la longitud, que el significado depende del tipo del parámetro que se use. Por ejemplo si se usa un vector, nos dice la cantidad de elementos en el mismo.

```
>>> max([1, 34, 56, 12, -90, 101])
101
>>> min([1, 34, 56, 12, -90, 101])
-90
>>> len([1, 34, 56, 12, -90, 101])
6
>>> len("Esto es una frase")
17
```

Listas por comprensión

Las listas por comprensión nos sirven para armar listas en una linea de codigo de una forma más simple y elegante.

Sintaxis: [expresion for item in iterable]

Opcionalmente se puede agregar un *if:*
[expresion for item in iterable if condicion]

También se puede construir una lista por comprensión a partir de otra lista por comprensión.

```
>>> numeros = [1, 1, 2, 3, 5, 8, 13]
>>> [i for i in numeros]
[1, 1, 2, 3, 5, 8, 13]
>>> [i + 5 for i in numeros]
[6, 6, 7, 8, 10, 13, 18]
>>> [i -1 for i in numeros if i < 5]
[0, 0, 1, 2]

>>> [letra for letra in "Hola"]
['H', 'o', 'l', 'a']

>>> [i + 1 for i in [a + 2 for a in [1, 2 ,3]]]
[4, 5, 6]
```

Ejercicio

Tiempo ⏱️

Escribir una función que a partir de una lista de nombres, retorne otra lista de nombres insertando un “Hola “ antes de cada nombre pero solo a los nombres que contengan una “i” en el nombre

Ejemplo
Entrada: [“Fede”, “Lucia”, “Alex”, “Simon”]
Salida: [“Hola Lucia”, “Hola Simon”]

map

La función *map* recibe dos parámetros:

1. Una función
2. Un ***iterable*** (una tupla, un vector, una lista, etc).

Retorna un *map objet* de los resultados luego de aplicar la función provista a cada elemento del iterable. Aplica la función provista a cada elemento, por lo que la longitud del resultado va a ser la misma que la original (no se van a perder o aparecer elementos).

Un *map object* es un iterable, para mayor comodidad se puede convertir en una lista utilizando la función *list* o en un set utilizando la función *set*.

```
def duplicar(n):
  return n * 2

numeros = [1, 5, 6, 12]
resultado = map(duplicar, numeros)
print(resultado)
# <map object at 0x10df1a190>
print(list(resultado))
# [2, 10, 12, 24]

textos = ["hola", "casa", "apuntes", "Python"]
# Se usa la función list que convierte textos en
# listas de caracteres
resultado = map(list, textos)
print(resultado)
# <map object at 0x10293d210>
print(list(resultado))
# [['h', 'o', 'l', 'a'], ['c', 'a', 's', 'a'], ['a',
# 'p', 'u', 'n', 't', 'e', 's'], ['P', 'y', 't', 'h',
# 'o', 'n']]
```

filter

La función *filter* recibe dos parámetros:

1. Una función
2. Un ***iterable*** (una tupla, un vector, una lista, etc).

Retorna un *filter objet* de los resultados luego de aplicar la función provista a cada elemento del iterable. *filter* va a filtrar los elementos y el iterable resultante va a tener solo los elementos del iterable original para los cuales la función provista retorno ***True***

Un *filter object* es un iterable, para mayor comodidad se puede convertir en una lista utilizando la función *list* o en un set utilizando la función *set*.

```
def mayorA10(numero):
 return numero > 10

numeros = [1, 2, 5, 8, 10, 12, 20, 37]
resultado = filter(mayorA10, numeros)
print(resultado)
# <filter object at 0x10f5c9250>
print(list(resultado))
# [12, 20, 37]


def esVocal(caracter):
 return caracter in ['a', 'e', 'i', 'o', 'u']

texto = "Se va a convertir en una lista de vocales"
resultado = filter(esVocal, texto)
print(resultado)
# <filter object at 0x100aca210>
print(list(resultado))
# ['e', 'a', 'a', 'o', 'e', 'i', 'e', 'u', 'a', 'i',
'a', 'e', 'o', 'a', 'e']
```

filter

Atención: Todo lo que se puede lograr con un filter se puede lograr utilizando listas por comprensión. La diferencia es que *filter* retorna un iterable (que se puede convertir en lo que queramos) y la lista por comprensión retorna una lista.

```
def mayorA10(numero):
 return numero > 10

numeros = [1, 2, 5, 8, 10, 12, 20, 37]
print([i for i in numeros if mayorA10(i)])
# [12, 20, 37]


def esVocal(caracter):
 return caracter in ['a', 'e', 'i', 'o', 'u']

texto = "Se va a convertir en una lista de vocales"
print([letra for letra in texto if esVocal(letra)])
# ['e', 'a', 'a', 'o', 'e', 'i', 'e', 'u', 'a', 'i',
'a', 'e', 'o', 'a', 'e']
```

lambda

Las funciones *lambda* son funciones **anónimas** que pueden tener más de un parametro pero tienen que ser compuestas por **una sola** expresión.

Se utilizan para representar funciones de una sola expresión de una forma sencilla y cómoda.

Sintaxis: lambda argumentos: expresion.

```
# Las siguientes funciones son equivalentes
def sumar(a, b):
  return a  b

# Ambos “sumar” son equivalentes
sumarL = lambda a, b: a + b
print(sumarL(9 , 11))
# 20

# Pueden sernos útiles para usar en funciones que
# requieran una función por parámetro (como map o
# filter)
numeros = [1, 2, 5, 11, 200]
print(list(filter(lambda n: n > 10, numeros)))
# [11, 200]
```

Ejercicio

Tiempo ⏱️

- Utilizando *filter, map y lambdas* escribir una función que reciba una lista de números y retorne otra con los números pares de la lista original elevados al cuadrado

Manejo de excepciones

Las **excepciones** son errores **imprevistos** que afectan el flujo normal de la aplicación.
NO son excepciones:

- Validaciones de entradas del usuario. Ejemplo: No es una excepción que el usuario haya ingresado 90 cuando se le pedía un número del 1 al 10
- Errores de sintaxis.

SI son:

- Condiciones externas a la aplicación (problemas de hardware, de memoria, etc).
- Falla en la conexión a sistemas externos (bases de datos, servidores, etc).

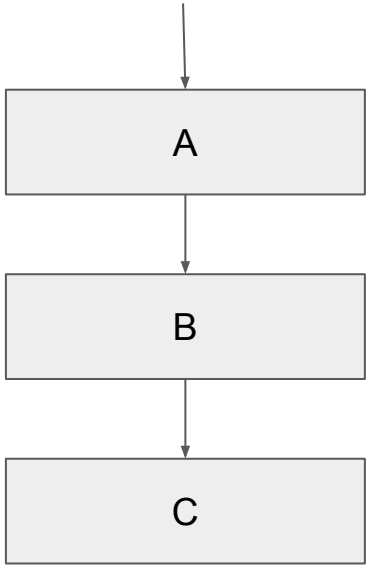
Manejo de excepciones

Supongamos el siguiente escenario: Tenemos las funcionas A, B y C.

- El usuario llama a A
- A internamente utiliza a B, por lo que eventualmente lo utiliza.
- C internamente utiliza C, por lo que eventualmente lo utiliza.
- C se quiere conectar a la base de datos y falla, por lo que lanza una excepción.

Las excepciones una vez que son lanzadas, “escalan” hasta que se las atrapen, si no llegan hasta los usuarios de la aplicación.

- B recibe la excepción de C, como no la atrapa llega hasta A
- A recibe la excepción de B pero si la atrapa y muestra un mensaje amigable al usuario.



Manejo de excepciones

Para lanzar una excepción se utiliza *raise*.
Sintaxis: *raise excepcion(textoDeLaExcepcion)*

Para atraparla se utiliza la estructura *try-except*. Dentro de **try** ponemos el código que podría lanzar la excepción y dentro del **except** el código a ejecutar en el caso que haya una excepción.

Repetimos: Una excepción es un caso **excepcional**. Solo hay que utilizar try-except para bloques de código que pueden lanzar excepciones.

```
# Sin atrapar la excepcion
def A():
 return B()

def B():
 return C()

def C():
 raise Exception("Esto es un error")

A()
# Salida
Traceback (most recent call last):
  File "delete.py", line 10, in <module>
    A()
  File "delete.py", line 2, in A
    return B()
  File "delete.py", line 5, in B
    return C()
  File "delete.py", line 8, in C
    raise Exception("Esto es un error")
Exception: Esto es un error
```

Excepciones pre fabricadas

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      |    +-- ModuleNotFoundError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- OSError
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError
      |    |    +-- ConnectionAbortedError
      |    |    +-- ConnectionRefusedError
      |    |    +-- ConnectionResetError
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- InterruptedError
      |    +-- IsADirectoryError
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- ProcessLookupError
      |    +-- TimeoutError
      +-- ReferenceError
      +-- RuntimeError
      |    +-- NotImplementedError
      |    +-- RecursionError
      +-- SyntaxError
      |    +-- IndentationError
      |         +-- TabError
      +-- SystemError
      +-- TypeError
      +-- ValueError
      |    +-- UnicodeError
      |         +-- UnicodeDecodeError
      |         +-- UnicodeEncodeError
      |         +-- UnicodeTranslateError
      +-- Warning
```

Manejo de excepciones

Para lanzar una excepción se utiliza *raise*.
Sintaxis: *raise excepcion(textoDeLaExcepcion)*

Para atraparla se utiliza la estructura *try-except*. Dentro de **try** ponemos el código que podría lanzar la excepción y dentro del **except** el código a ejecutar en el caso que haya una excepción.
Sintaxis:
try:
  codigo normal
execept:
 código a ejecutar en caso de excepcion

```
# Atrapandola
def A():
 try:
  return B()
 except:
  print(“Ocurrio un error”)

def B():
 return C()

def C():
 raise Exception("Esto es un error")

A()
# Salida
Ocurrio un error
```

Manejo de excepciones

En el ejemplo anterior se lanza la excepcion “*Exception*”, se puede lanzar dicha excepción o crear una personalizada.

Si al bloque *except* se le especifica una (o más) excepciones, solo va a atrapar las excepciones especificadas. En caso que se lance otra, no se atrapara.

```
class miExcepcion(Exception):
 pass

def f():
 raise miExcepcion("Hola, falle")

try:
 f()
except miExcepcion as ex:
 print(ex)

# Salida:
Hola, Falle
```

Ejercicio

Tiempo ⏱️

Escribir una función “compra” que reciba una lista de precios y un saldo total y calcule el vuelto. En caso de que el saldo sea insuficiente lanzar una excepción llamada “SaldoInsuficiente” incluyendo en el mensaje el saldo y el total de la compra