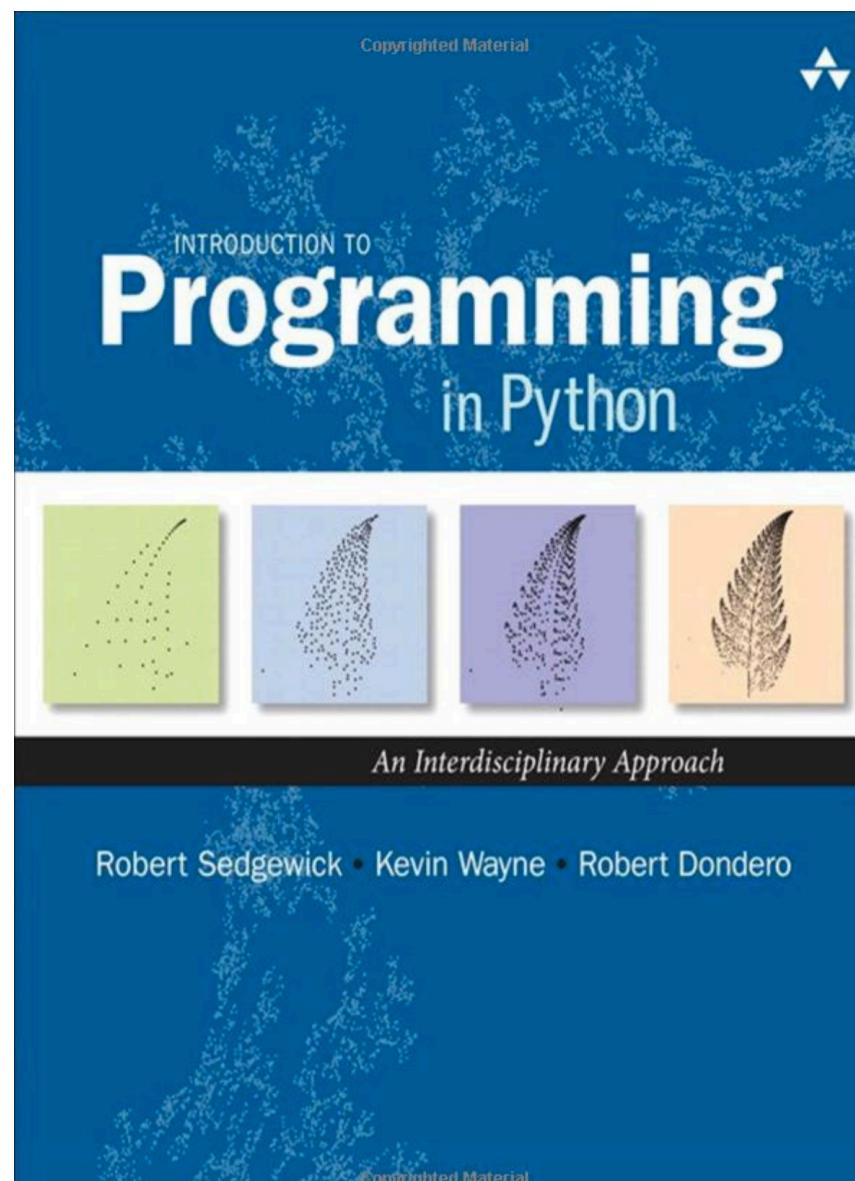


# Parte I: Intro pensamiento computacional

## Clase 12: Manejo de errores y módulos

Diego Caro  
[dcaro@udd.cl](mailto:dcaro@udd.cl)



Basada en presentaciones oficiales de libro *Introduction to Programming in Python* (Sedgewick, Wayne, Dondero).

Disponible en <https://introcs.cs.princeton.edu/python>

# Outline

- Casos de uso para `dict()`
- Rendimiento estructuras de datos `list`, `tuple`, `set`, `frozenset`
- Tipos y manejo de errores
- Módulos

# Ejemplo: ¿Cómo almacenar las notas de un curso?

- Podríamos usar una lista para nombre de alumno, notas y el curso:

```
nombres = ['Diego', 'Francisca', 'Daniela', 'Leo']
notas = [4.1, 5.5, 6.8, 3.9]
cursos = ['progra', 'algebra', 'fisica', 'calculo']
```

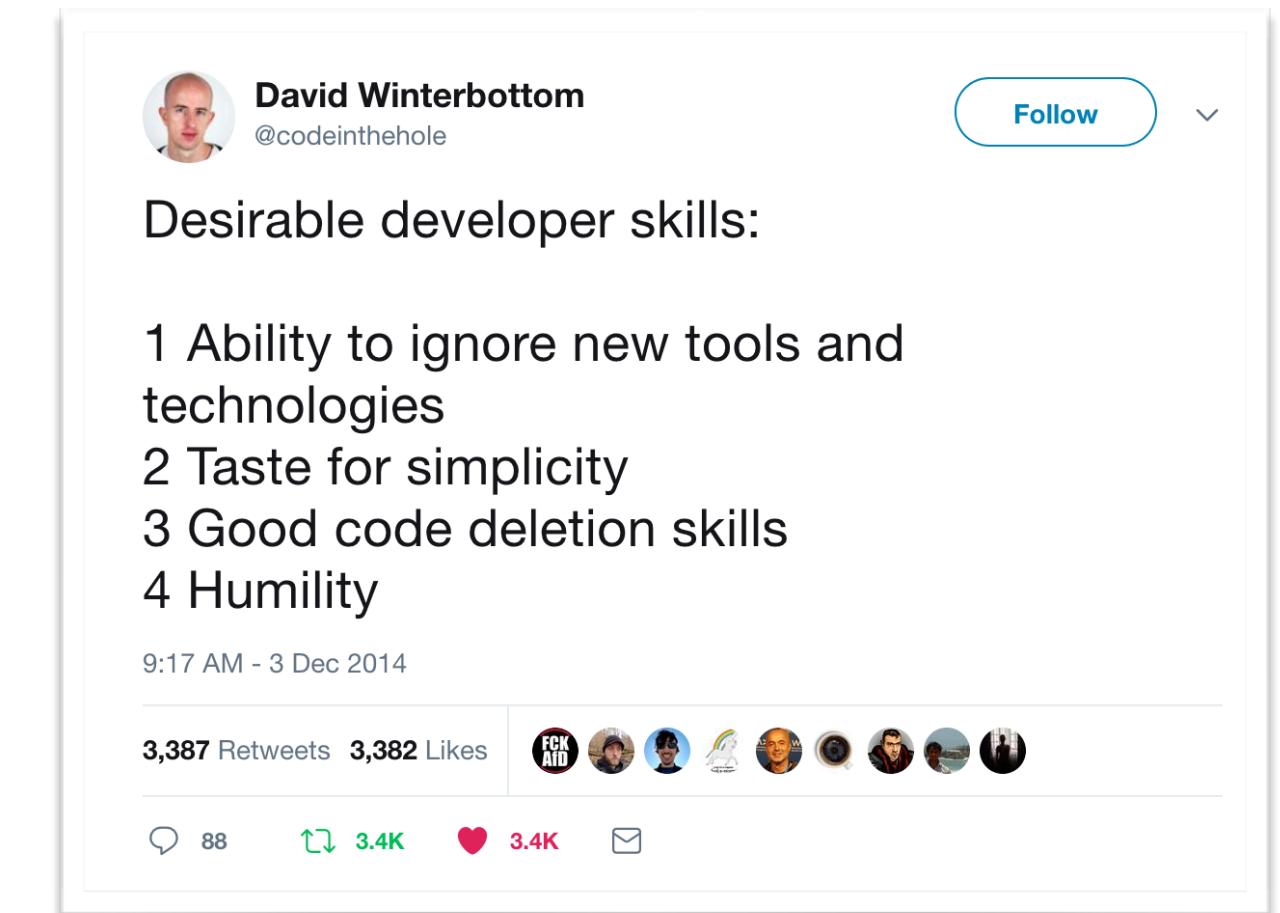
- Cada lista contiene información distinta.
- Las listas deben ser del **mismo tamaño**.
- Información entre las listas deben estar en la misma **posición**.

# Ejemplo: ¿Cómo almacenar las notas de un curso?

```
def obtener_notas(estudiante, lista_nombres, lista_notas, lista_cursos):
    i = lista_nombres.index(estudiante)
    nota = lista_notas[i]
    curso = lista_cursos[i]
    return (curso, nota)

curso, nota = obtener_notas('Diego', nombres, notas, cursos)
print('Diego tiene un', nota, 'en', curso)
```

- Complicado si tienes varios tipos de información que almacenar
- Debes mantener **varias listas**, y pasarlas como argumento
- **Índice** siempre es un **entero** con la posición
- **DIFÍCIL DE MANTENER!**



# Ejemplo: ¿Cómo almacenar las notas de un curso?

```
Clave           Valor           Tuple
Diccionario → 1 notas = { 'Diego'      : ('prograudd',4.1)
                           , 'Francisca'   : ('algebra'    ,5.5)
                           , 'Daniela'     : ('fisica'     ,6.8)
                           , 'Leo'         : ('prograudd',3.9)}
                           5
                           6 def obtener_notas(estudiante, dict_notas):
                           7     curso, nota = dict_notas[estudiante]
                           8     return (curso, nota) ← Tuple
                           9
                           10 curso, nota = obtener_notas('Diego', notas)
                           11 print('Diego tiene un', nota, 'en', curso)
```

# Patrón típico de uso diccionario

- Recorrer todas las llaves del diccionario:

```
for llave in notas.keys():
    print(llave)
```

```
notas = {'Diego' : 4.1,
         'Francisca' : 5.5,
         'Daniela' : 6.8,
         'Leo' : 3.9}
```

- Recorrer los valores del diccionario:

```
for valor in notas.values():
    print(valor)
```

- Recorrer todas las llaves y valores:

```
for llave, valor in notas.items():
    print(llave, valor)
```

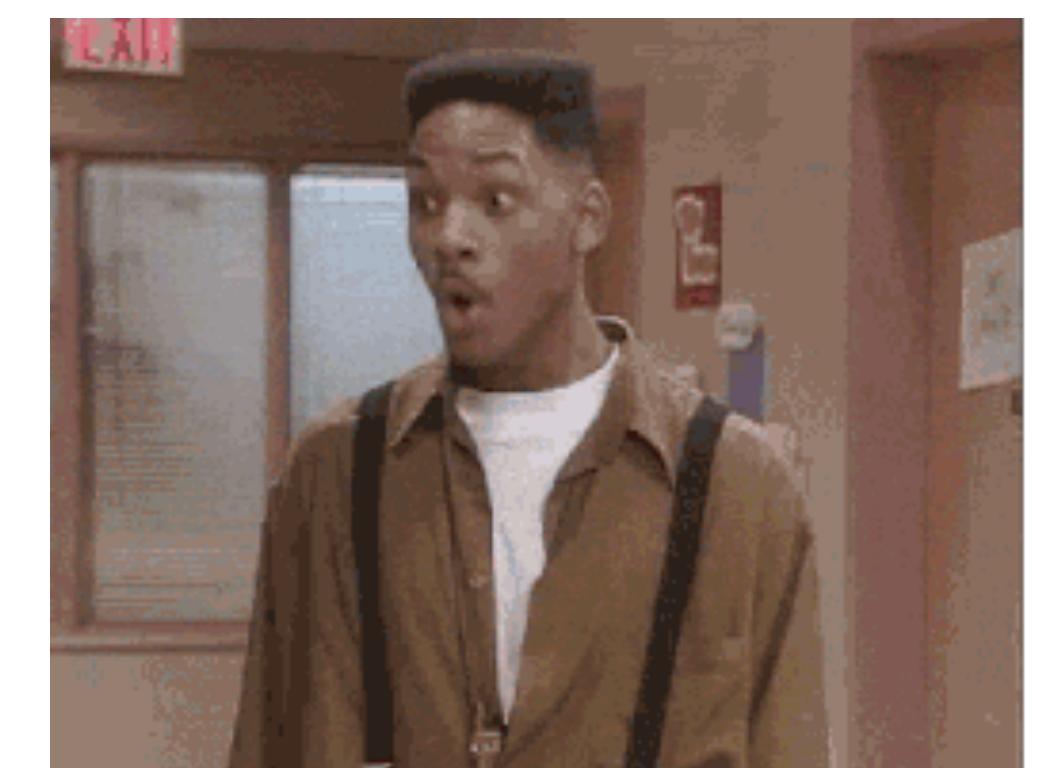
# Desempeño en estructuras de datos

```
1 import timeit
2
3 def find(data, elem):
4     return elem in data
5
6 number_trials = 10
7
8 for size in [100000, 10**6, 10**7, 10**8]:
9     x = size - 1
10
11 dataset = dict()
12 dataset['list'] = list(range(size))
13 dataset['tuple'] = tuple(range(size))
14 dataset['set'] = set(range(size))
15 dataset['frozenset'] = frozenset(range(size))
16
17 for (datatype, datavalue) in dataset.items():
18     code = "find(dataset['{}'], x)".format(datatype)
19     avg_time = timeit.timeit(code, number=number_trials, globals=globals()) / number_trials
20     print('Average time for {} elements using {}: {:.3f}s'.format(size, datatype, avg_time))
```

Datos están en:

- list
- tuple
- set
- frozenset

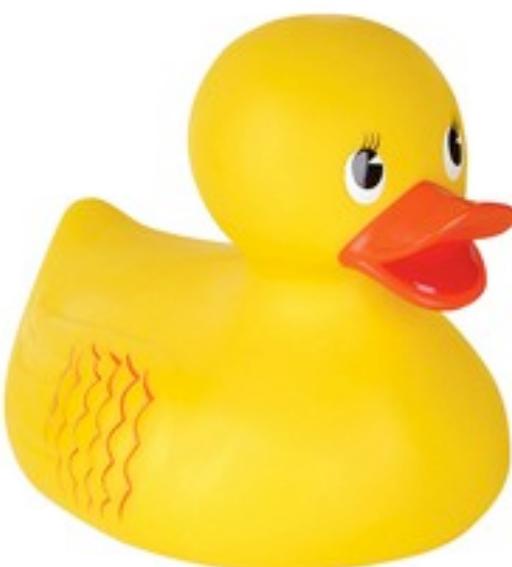
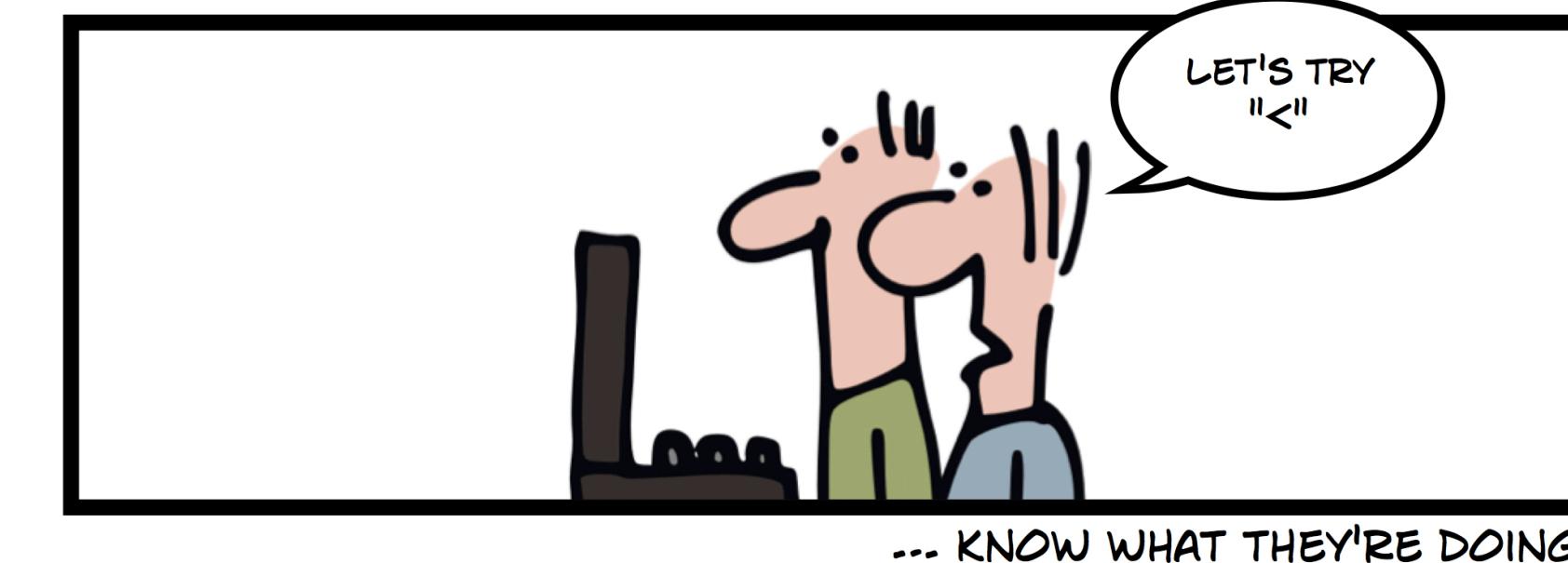
```
Average time for 100000 elements using list: 0.001s
Average time for 100000 elements using tuple: 0.001s
Average time for 100000 elements using set: 0.000s
Average time for 100000 elements using frozenset: 0.000s
Average time for 1000000 elements using list: 0.009s
Average time for 1000000 elements using tuple: 0.010s
Average time for 1000000 elements using set: 0.000s
Average time for 1000000 elements using frozenset: 0.000s
Average time for 10000000 elements using list: 0.094s
Average time for 10000000 elements using tuple: 0.094s
Average time for 10000000 elements using set: 0.000s
Average time for 10000000 elements using frozenset: 0.000s
Average time for 100000000 elements using list: 2.328s
Average time for 100000000 elements using tuple: 3.915s
Average time for 100000000 elements using set: 0.000s
Average time for 100000000 elements using frozenset: 0.000s
```



# Errores y Excepciones

- **Syntax errors:** el código no es sintácticamente válido, falta un paréntesis, falta indentar, etc... (fácil de arreglar)
- **Runtime errors:** código es sintácticamente válido, pero no ejecuta. Por ejemplo, acceder a una posición de arreglo que no existe, ejecutar una función sobre un tipo de dato que no lo implementa.
- **Semantic errors:** código sintácticamente válido y que se ejecuta pero que no hace lo que el programador quería que hiciera.
  - Difícil, hay que pensar para arreglarlo!

GOOD CODERS...



# Runtime errors

```
print(Q)
```

```
-----  
NameError                                 Traceback (most recent call last)  
<ipython-input-1-e796bdcf24ff> in <module>()  
----> 1 print(Q)
```

```
NameError: name 'Q' is not defined
```

```
1 + 'abc'
```

```
-----  
TypeError                                Traceback (most recent call last)  
<ipython-input-2-aab9e8ede4f7> in <module>()  
----> 1 1 + 'abc'
```

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

```
2 / 0
```

```
-----  
ZeroDivisionError                          Traceback (most recent call last)  
<ipython-input-3-ae0c5d243292> in <module>()  
----> 1 2 / 0
```

```
ZeroDivisionError: division by zero
```

```
L = [1, 2, 3]  
L[1000]
```

```
-----  
IndexError                                Traceback (most recent call last)  
<ipython-input-4-06b6eb1b8957> in <module>()  
      1 L = [1, 2, 3]  
----> 2 L[1000]
```

```
IndexError: list index out of range
```

# Try and except

- Cuando queremos mantener ejecutando el programa aunque haya un Runtime error.
- **Ejemplo 1**

```
try:  
    print("Esto se ejecuta primero")  
  
except:  
    print("Esto se ejecuta solo si hay un error")
```

El segundo bloque de código no se ejecuta porque el primero no reportó ningún error.

- **Ejemplo 2**

```
try:  
    print("Intentemos dividir por cero")  
    x = 1 / 0 # ZeroDivisionError  
  
except:  
    print("Algo malo ocurrió")
```

# Try and except

¿Qué pasa al ejecutar?

```
def safe_divide(a, b):  
    try:  
        return a / b  
    except:  
        return 1E100
```



```
safe_divide(1, 2)
```

```
safe_divide(2, 0)
```

# Raiseing Exceptions

Obviamente, es bueno tener excepciones que son informativas. Hay veces que también queremos agregar (**raise**) nuestras propias excepciones!

```
raise RuntimeError("my error message")
```

```
-----
RuntimeError                                Traceback (most recent call last)
<ipython-input-16-c6a4c1ed2f34> in <module>()
----> 1 raise RuntimeError("my error message")

RuntimeError: my error message
```

RunTimeError

# Raiseing Exceptions

```
def fibonacci(N):  
    L = []  
    a, b = 0, 1  
    while len(L) < N:  
        a, b = b, a + b  
        L.append(a)  
    return L
```

```
def fibonacci(N):  
    if N < 0:  
        raise ValueError("N must be non-negative")  
    L = []  
    a, b = 0, 1  
    while len(L) < N:  
        a, b = b, a + b  
        L.append(a)  
    return L
```

# Alternativa: assert

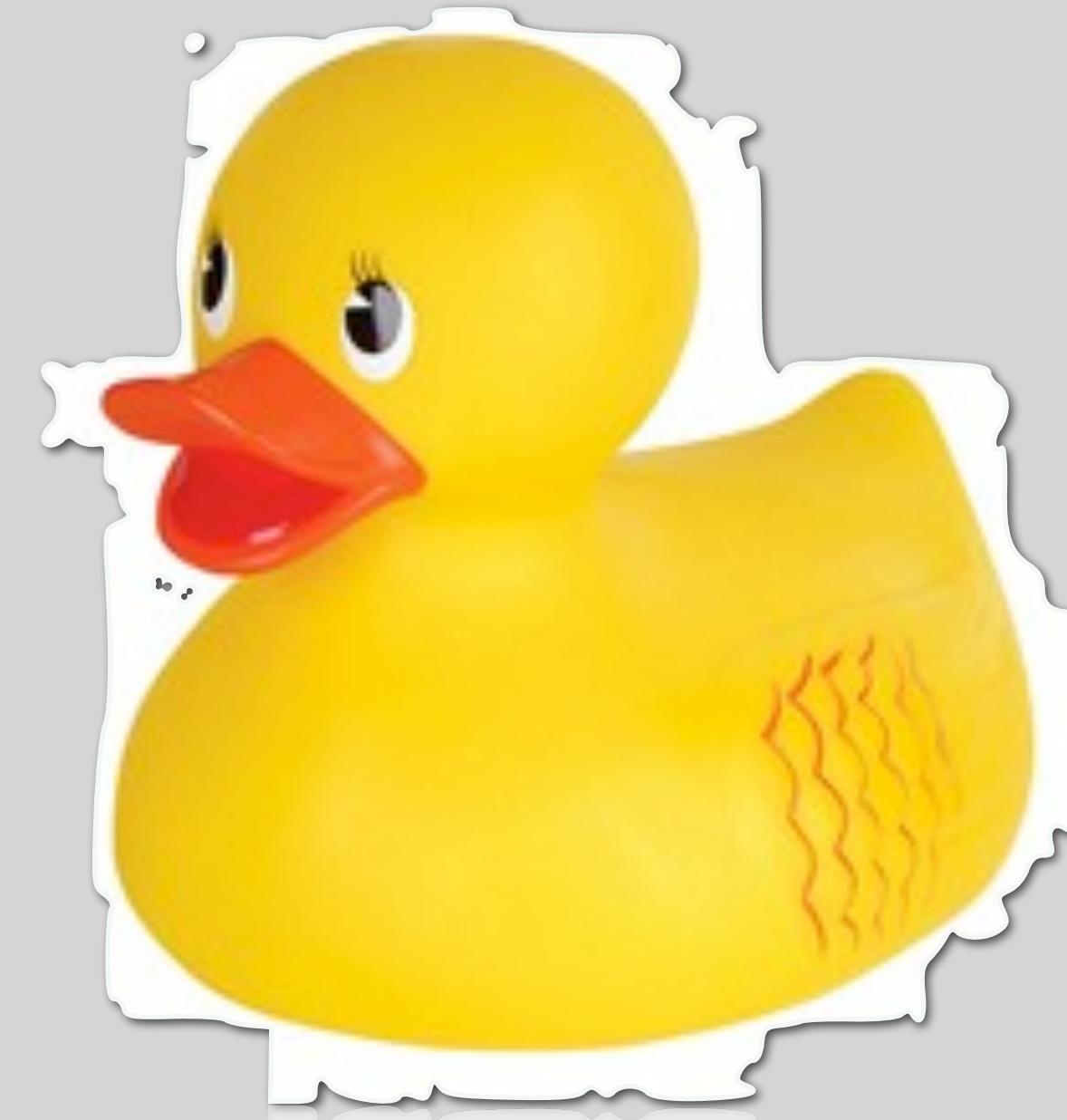
- Assert genera un excepción si la condición no se cumple.
- **Sintaxis:** assert <condición>, mensaje

```
def fibonacci(N):
    assert N >= 0, "N must be non-negative"
    L = []
    a, b = 0, 1
    while len(L) < N:
        a, b = b, a + b
        L.append(a)
    return L
fibonacci(-2)
```

```
Traceback (most recent call last):
  File "exception.py", line 11, in <module>
    fibonacci(-2)
  File "exception.py", line 3, in fibonacci
    assert N >= 0, "N must be non-negative"
AssertionError: N must be non-negative
```

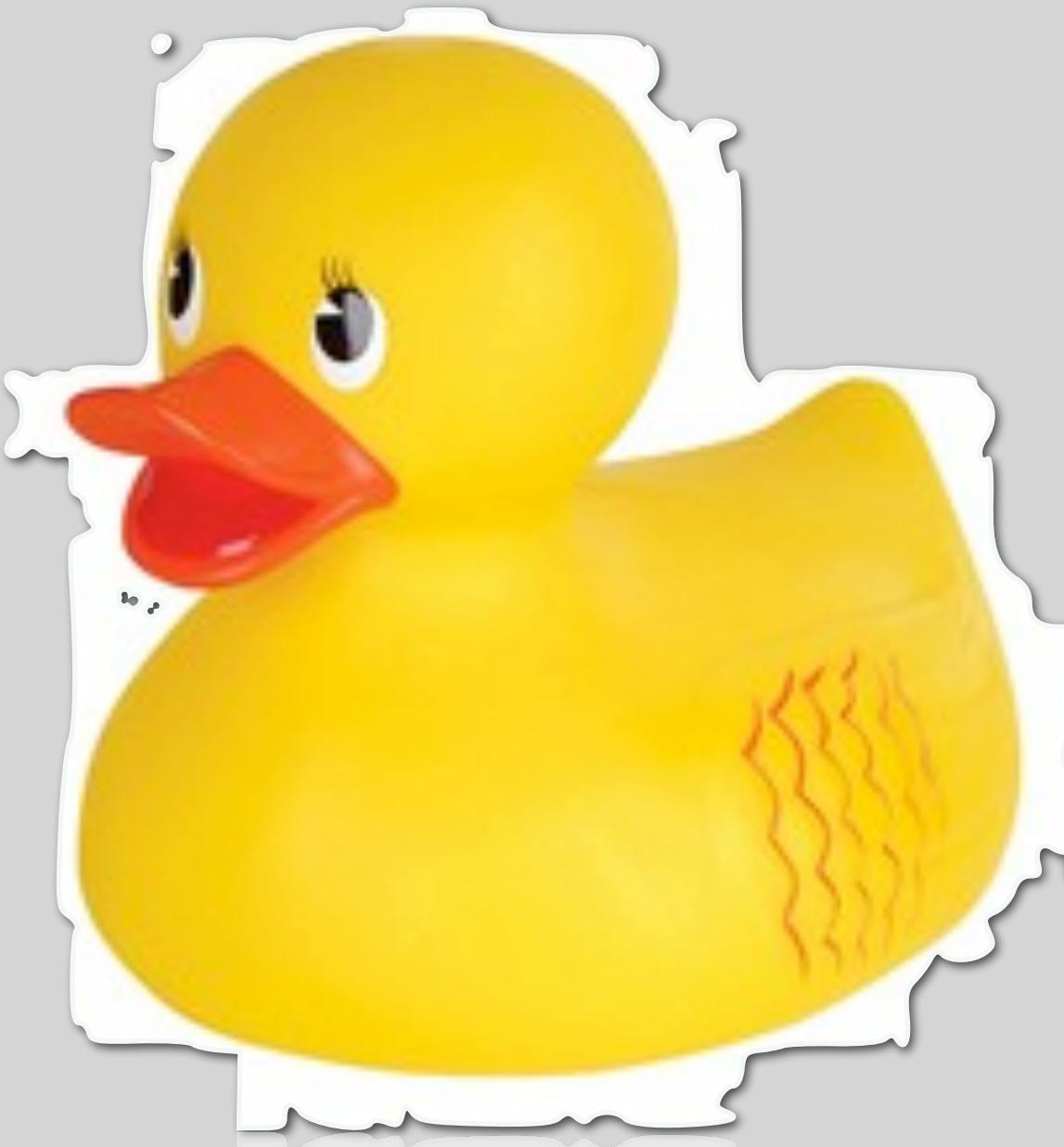
# ¿Cuál tipo de error genera este código?

```
1 felinos = ['leon', 'puma']
2 rapaz = ['aguila']
3 depredadores = [felinos]
4 depredadores.append(rapaz)
5 print(depredadores)
6 rapaz.append('buitre')
7 print(rapaz)
8 print(depredadores)
```



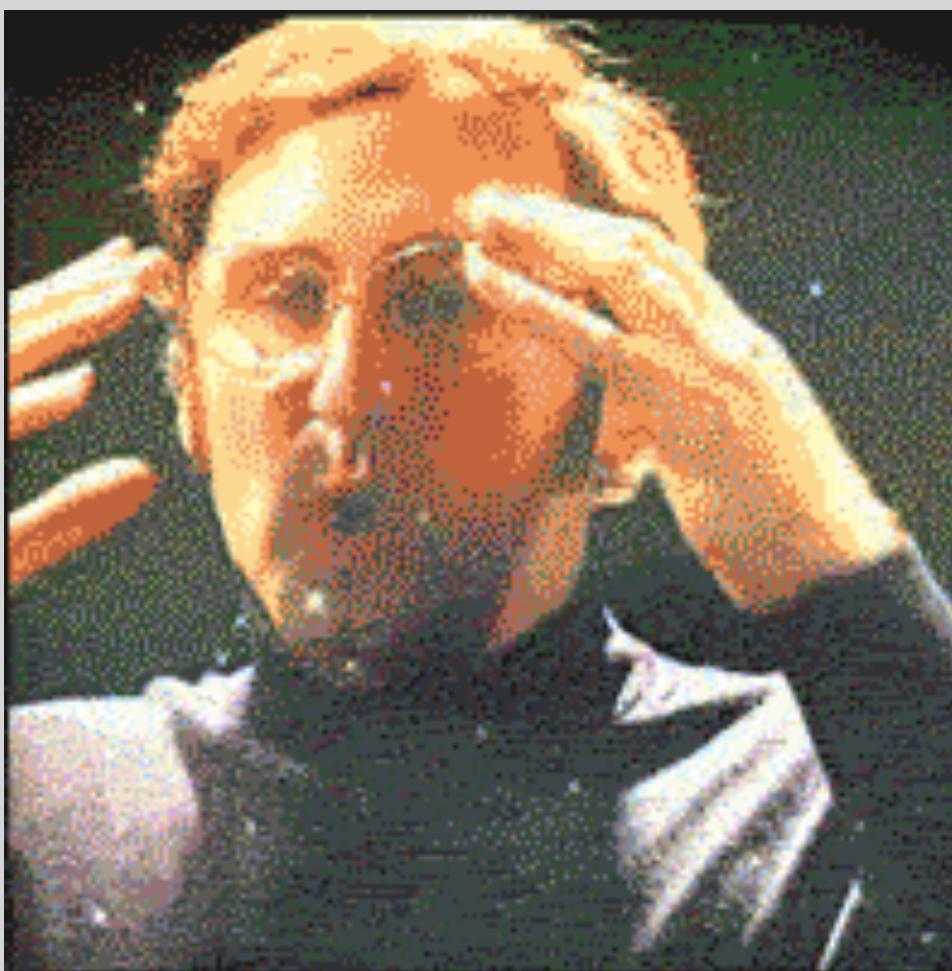
# ¿Qué imprime?

```
1 felinos = ['leon', 'puma']
2 rapaz = ['aguila']
3 depredadores = [felinos]
4 depredadores.append(rapaz)
5 print(depredadores)
6 rapaz.append('buitre')
7 print(rapaz)
8 print(depredadores)
```



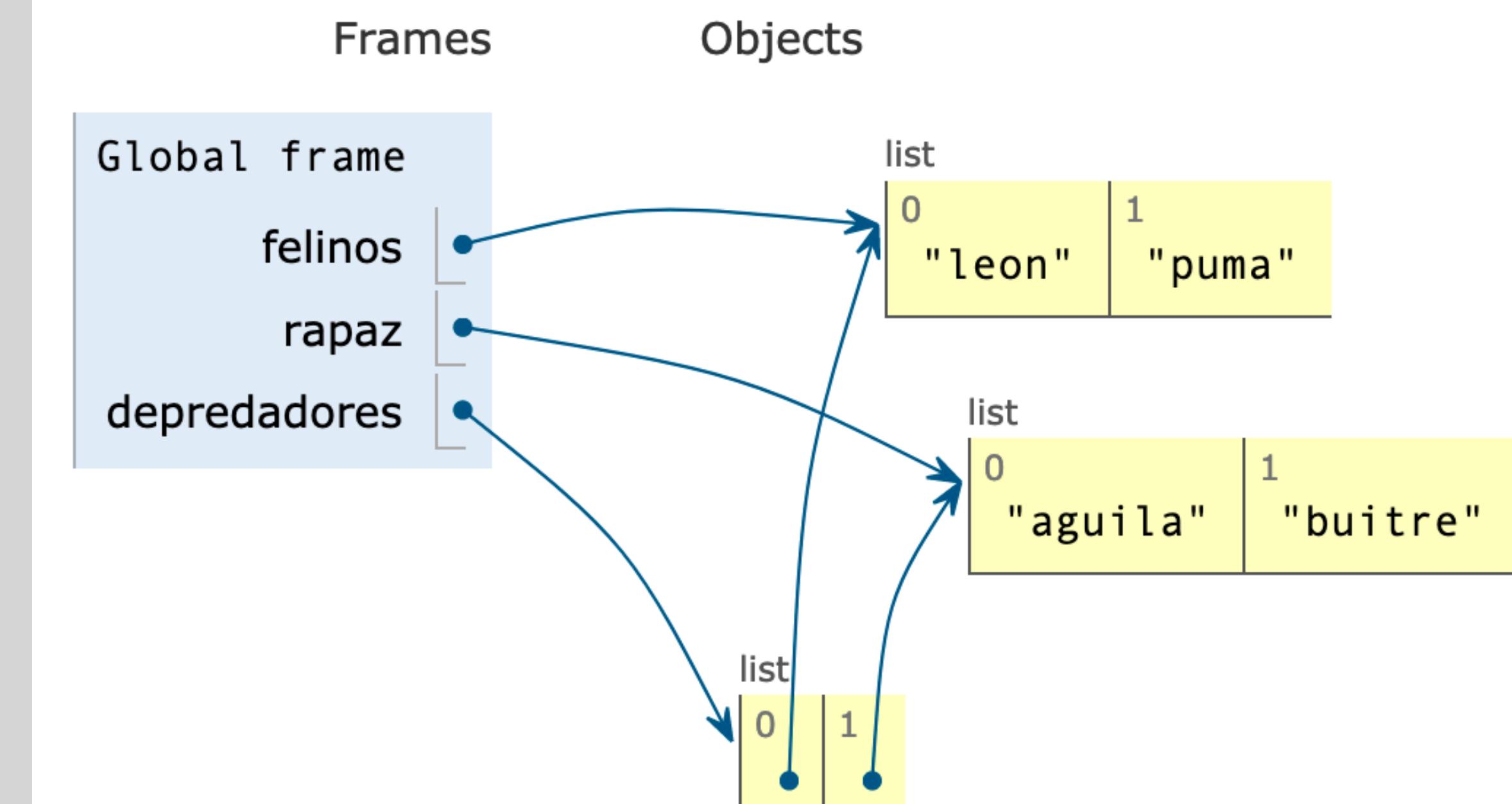
# ¿Qué imprime?

```
1 felinos = ['leon', 'puma']
2 rapaz = ['aguila']
3 depredadores = [felinos]
4 depredadores.append(rapaz)
5 print(depredadores)
6 rapaz.append('buitre')
7 print(rapaz)
8 print(depredadores)
```



Print output (drag lower right corner to resize)

```
[['leon', 'puma'], ['aguila']]
['aguila', 'buitre']
[['leon', 'puma'], ['aguila', 'buitre']]
```



# Módulos

- Hasta ahora todos nuestros programas son un simple archivo .py
- Si nuestro programa es muy grande, es complicado mantenerlo actualizado
- En Python se pueden ejecutar funciones definidas en otros archivos
  - Permite reutilizar código: un programa usar código que ya ha sido escrito, sin necesariamente copiar-pegar
  - Permite hacer programación modular: construir programas componiendo código de diversas fuentes (otra vez, sin copiar y pegar)

**cliente.py**

```
import matriz

m = matriz.crear(3,3)
matriz.asignar(m, 0, 1, 9)
matriz.asignar(m, 2, 2, 3)
matriz.asignar(m, 1, 2, 1)
matriz.imprimir(m)
```

"importa el código en matriz.py"

**matriz.py**

```
def crear(m,n):
    #Retorna una matriz de ceros con m filas por n columnas
    matriz = []
    for i in range(m):
        matriz.append([0]*n)
    return matriz
```

```
def imprimir(matriz):
    #Imprime cada fila de una matriz
    for fila in matriz:
        for elem in fila:
            print(elem, end=' ')
        print()
```

```
def asignar(matriz, i,j,v):
    matriz[i][j] = v
```

```
def main():
    print(crear(10,20))
```

```
if __name__ == '__main__':
    main()
```

Código para probar el módulo

# Ejercicio

- Extender módulo matriz
  - Sumar dos matrices y un escapar
  - Multiplicar dos matrices y un escalar
  - Producto cruz entre dos matrices
  - Traspuesta de una matriz