

Cuestiones sobre listas y referencias

- ¿qué ocurre en este caso?

$L = [[0, 0]] * 5$

$L[2][0] = 7$

- ¿y en este otro?

$L = ["abc"] * 3$

$L = [L] * 5$

$L[2][0] = 'cde'$

Resumen de Operaciones sobre listas I

- `a = []` crea una lista vacía
- `a = [1, 4.4, 'run.py']` asigna un valor
- `a.append(elem)` añade un elemento al final
- `a + [1,3]` une dos listas
- `a[3]` accede a un elemento
- `a[-1]` accede al último elemento
- `a[1:3]` accede a una sublista
- `del a[3]` elimina un elemento

Resumen de Operaciones sobre listas II

- `a.remove(4.4)` elimina un elemento con ese valor
- `a.index('run.py')` devuelve la posición donde se encuentra un elemento (**ValueError** si no está)
- `a.count(v)` cuenta el número de veces que aparece un elemento (en este caso no hay error)
- `len(a)` devuelve el número de elementos
- `min(a)` devuelve el elemento más pequeño
- `max(a)` devuelve el elemento mayor
- `sum(a)` suma todos los elementos

Resumen de Operaciones sobre listas III

- `a.sort()` ordena una lista
- `as = sorted(a)` devuelve una versión ordenada
- `a.reverse()` invierte el orden de los valores de una lista
- `b[3][0][2]` indexado anidado
- `isinstance(a, list)` devuelve True si a es una lista
- `l1.copy()` devuelve una copia de la lista

Ejemplo string + list

- **readline()** Lee una línea completa
`linea = sys.stdin.readline()`
- **readlines()** Lee una serie de líneas, cada línea va en una posición de la lista
`texto = sys.stdin.readlines()`
- **split()** divide una cadena y devuelve una lista de subcadenas
`palabras = cad.split()`
`datos = cad.split("es")`
- **strip()** elimina separadores de ambos extremos

Ejemplo string + list II

```
['el','gato','sentado','sobre','el','coche'].count('el')
'el gato sentado sobre el coche'.split().count('el')
palabras = 'el gato sentado sobre el coche'.split()
    ' el gato '.strip()
    ' el gato '.lstrip()
```

Ejemplos

primos = [2, 3, 5, 7, 11, 13, 17, 19]

[8 / p for p in primos]

[p for p in primos if p % 3 > 0]

[[0,0,0] for x in range(2,5)]

[93 % p for p in primos if 93 % p != 0]

[5 ** p for p in primos if p % 4 == 0]

[[fil[i] for fil in matriz] for i in range(len(matriz))]

L = [[0 for i in range(5)] for j in range(6)]

Tipo de dato set

- Es una colección no ordenada de datos (pueden ser de distinto tipo).

`s1 = {1,3,5,7,5,4,3,2,1}`

`s2 = {1, 'a' }`

- Debe ser de tipo “hashable”, que se pueda “aplicar una función hash sobre él” (no mutable?)

`>>> s1 = {(1, 'a'), (2, 'b')}`

`>>> s2 = {[1,2,3], [1,3,4]}`

`...`

`TypeError: unhashable type: 'list'`

Tipos de datos conjunto

- Existe `set` como versión mutable y `frozenset` e `immutableSet` como inmutables
- Ejemplo:

```
dias_semana = {'lun', 'mar', 'mié', 'jue', 'vie'}
```

```
dias_semana = set(('lun', 'mar', 'miércoles', 'jueves',  
'viernes'))
```

```
dias_semana.add('sáb')
```

'mié' in `dias_semana` devuelve `True`

`dias_semana.remove('mié')` elimina elemento

Operaciones sobre conjuntos I

- Añadir un elemento `add(x)`
- Eliminar un elemento `remove(x)` `discard(x)`

la diferencia entre ambos:

`s1 ={1,3,5,7,5,4,3,2,1}`

`s1.discard(8) # no ocurre nada`

`s1.remove(8) # Genera excepción KeyError: 8`

- Devuelve y elimina un elemento aleatorio `pop()`
- Copiar todos los elementos `copy()`
- Eliminar todos los elementos `clear()`

Operaciones sobre conjuntos II

Incluye union, intersection, difference, symmetric_difference

s1 ={1,3,5,7,5,4,3,2,1}

s2 ={2, 4, 6, 8, 6, 4, 3, 2, 1}

union_s1_s2 = s1.union(s2)

intersect_s1_s2 = s1.intersection(s2)

diff = s1.difference(s2) # set((5,6,7,8)) #set((5,7))

s_diff = s1.symmetric_difference(s2) # set((5,6,7,8))

Comprensión de conjuntos

De forma similar a las listas

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}  
set(['r', 'd'])
```

```
>>> vocalesPalabra = {x for x in 'esto es una prueba'  
if x in 'aeiou'}  
set(['a', 'u', 'e', 'o'])
```

```
>>> L = [34, 45, 87, 90, 32, 4]
```

```
>>> mayores50 = {x for x in L if x>50}  
set([90, 87])
```

Tipo de dato dict I

Tipo *diccionario*: Permite guardar pares (clave, valor) y poder acceder a ellos de forma eficiente a través de la clave

Las claves deben ser **inmutables**. Los valores no tienen restricciones.

Tipo de dato dict II

Se pueden guardar claves y valores de distintos tipos

Operaciones: buscar, borrar, modificar o definir pares.

También se les conoce como **tablas hash** o **arrays asociativos**

Asignación y acceso

- Se usa {}, separando clave y valor con : y cada par con ,

```
datos = {'nombre' : 'Pepe', 'edad': 40}
```

```
# datos = dict(nombre='Pepe', edad=40)
```

- Se accede con [] usando la clave

```
datos['nombre']
```

```
datos['edad']
```

```
datos['Pepe'] ?????
```

- Internamente usan hashing

Asignación y acceso

- Ejemplo: temperaturas de ciudades

```
temps = {'Madrid' : 29, 'Granada': 30, 'Valencia' :  
28}
```

```
#o temps = dict(Madrid=29, Granada=30,  
Valencia=28)
```

- Modificar datos (si ya existe)

```
temps['Granada'] = 29
```

- Insertar datos (si no existía)

```
temps['Málaga'] = 30
```

Actualización

- Se actualiza usando la clave en con []
`datos['nombre'] = 'José'`
`datos['domicilio'] = 'Casa'`
`datos['id'] = 4532`
- Los diccionarios no siguen ningún orden preestablecido.

```
>>> datos
```

```
{'nombre': 'José', 'id': 4532, 'edad': 40,  
'domicilio': 'Casa'}
```

Asignación directa

- Ejemplo: temperaturas de ciudades

```
ciudades = ['Madrid', 'Granada', 'Valencia']
```

```
temps = [29, 30, 28]
```

```
d = dict(zip(ciudades, temps))
```

```
print (d)
```

Borrar datos

- Para borrar un elemento se usa **del**
del datos['nombre']
del temps['Valencia']
- También se puede usar **pop**
temps.pop('Málaga')
- Se pueden eliminar sólo los valores
temps['Granada'] = None
- Para borrar todos se usas **clear**
datos.clear()

Iterar sobre un diccionario

- Se puede usar

for i in datos :

 print (i, datos[i])

- Se puede usar **in** para comprobar si una clave está en el diccionario

if 'Granada' in temps :

 print ('La temperatura en Granada es',
 temps['Granada'])

else :

 print ('No hay temperatura para Granada')

Obtener claves y valores I

Los datos y los valores se pueden consultar como listas:

```
>>> temps.keys()  
['Valencia', 'Granada', 'Madrid']  
>>> temps.values()  
[28, 30, 29]
```

Los elementos no están ordenados, para tener ordenadas las claves hay que usar **sorted**

```
for i in sorted(temps.keys()) :  
    temp = temps[i]  
    print (temp)
```

Obtener claves y valores II

Los datos y los valores se pueden consultar como una lista de pares:

```
for ciudad, temp in temps.items() :  
    print (ciudad, ':', temp)
```

O de forma ordenada:

```
for ciudad, temp in sorted(temps.items()) :  
    print (ciudad, ':', temp)
```

```
ciudades = temps.keys()
```

Ejemplo

- Se puede usar

```
estaciones = {'primavera': {'marzo', 'abril', 'mayo'},  
'verano' : {'junio', 'julio', 'agosto'} }
```

```
for i, j in estaciones.items() :
```

```
    for k in j :
```

```
        print (k)
```

Tipo de dato array

- Hay que usar el módulo array
- Se crean indicando el tipo de dato

array(<tipo>, <datos>)

```
>>> arr = array('l', [1, 2, 3, 4, 5])
```

```
>>> arr[0:3]
```

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

```
>>> arr[3:-1]
```

```
[4, 5]
```

```
>>> arr[-1:0:-1]
```

```
[5, 4, 3, 2]
```

Número de parámetros variable en una función

- Se indica con * delante del parámetro

```
def func(*datos) :  
    for i in datos :  
        print (i)  
func(5)  
func(5,6,7)  
func([1,'a', 'c'])
```

Parámetros con nombre en una función

- Se indica con ** delante del parámetro

```
def func(**datos) :  
    for i, j in datos.items() :  
        print (i, j)  
func(pepe=1234,juan=34545)
```

Tipo de dato fichero

- Un fichero es una secuencia de bytes almacenada en memoria externa.
- Python maneja los archivos como secuencias de caracteres. Sólo se leen y escriben cadenas
- Es necesario abrir el fichero para poder acceder y posteriormente cerrarlo.

Operaciones sobre ficheros I

- Abrir un fichero (debe poderse abrir).

`open(<nombre>, <modo>)`

donde `<modo>` puede ser r, w, a

`f = open("datos.txt")`

debe existir y poderse acceder a los datos

- Cerrar un fichero.

`f.close()`

- Comprobar si existe un fichero.

`os.path.isfile(<nombre>)`

Operaciones sobre ficheros II

- Comprobar permisos sobre el fichero.

`os.stat(<nombre>)`

- Leer un número de bytes de un archivo

`f.read(<num>)`

si no indica un número se lee hasta el final

- Leer una línea completa

`f.readline()`

- Leer todo el archivo por líneas (en una lista)

`f.readlines()`

Operaciones sobre ficheros III

- Para posicionarse sobre un byte concreto
`seek(<num>, <donde>)`
donde puede ser 0:inicio, 1:actual, 2:final
- Conocer posición del fichero.
`tell()`
- Iterar sobre las líneas del fichero como una lista
`for linea in fich :`
- Como un bloque
`with open("fichero.txt", "rb") as f:`

Operaciones sobre ficheros IV

- Para ficheros de texto

```
f = open("datos.txt")
```

```
for line in f :
```

```
    print (line)
```

```
f.close()
```

- Para ficheros binarios

```
with open("fichero", "rb") as f:
```

```
    byte = f.read(1)
```

```
    while byte:
```

```
        byte = f.read(1)
```

Operaciones sobre ficheros V

- Consultas

name Devuelve el nombre del archivo

closed Devuelve si el fichero está cerrado

mode Devuelve el método de apertura

Ejemplo I

- Leer datos y ponerlos en dos listas de números reales:

```
x = []; y = []
```

```
lines = open("datos.txt")
```

```
for line in lines:
```

```
    xval, yval = line.split()
```

```
    x.append(float(xval))
```

```
    y.append(float(yval))
```

```
lines.close()
```

Ejemplo II

- Leer datos a un diccionario

```
codigos = {}
```

```
f = open("codigos.txt", "r")
```

```
for line in f :
```

```
    [codigo, lugar] = line.rstrip().split(' ')
```

```
    codigos[codigo] = lugar
```

```
f.close()
```

Guardar datos

- Escribir cadenas

`write(<cadena>)`

cada cadena se guarda a continuación de la anterior (sin separadores)

`f.write('Hola')`

`f.write ('Adiós')`

guardaría `HolaAdiós`

- Varías líneas al mismo tiempo

`f.writelines(<secuencia>)`

Ejemplo I

- Escribir cadenas

```
with open("salida.txt", "w") as f :
```

```
    f.write("Esto")
```

```
    f.write(" es ")
```

```
    f.write("una prueba")
```

```
L = ['esto', ' es ', 'una prueba']
```

```
with open("salida2.txt", "w") as f :
```

```
    f.writelines(L)
```

Ejemplo II

- Escribir datos formateados

```
with open("datos.txt", "w") as f :  
    for i in range(100, 200) :  
        f.write('{:3d} {:7d}'.format(i, i**2))  
        f.write('\n')
```

Entrada y salida estándar

- Módulo sys

`sys.stdin` Entrada estándar abierto para lectura

`sys.stdout` Salida estándar, abierta para escritura

`sys.stderr` Salida errores, abierta para escritura

```
import sys  
for line in sys.stdin :  
    sys.stdout.write(line)
```

Entrada y salida estándar

```
line = sys.stdin.readline()
```

```
while line :
```

```
    sys.stdout.write(line)
```

```
    line = sys.stdin.readline()
```

Información sobre el sistema

sys.path Devuelve una lista de strings con el path del sistema

Ejemplo ficheros csv

- Leer datos separados por comas

```
with open("datos.csv", "r") as f:  
    datos = f.readline().split(',')  
  
import csv  
  
with open('datos.csv', 'Ur') as f:  
    data = list(tuple(rec) for rec in csv.reader(f,  
delimiter=','))
```

Leer datos de la web

Leer datos

```
import urllib.request  
f = urllib.request.urlopen('http://www.python.org/')  
print(f.read(100).decode('utf-8'))  
  
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML  
1.0 Transitional//EN"  
"http://www.w3.org/TR/xhtml1/DTD/xhtml
```

Enviar datos en la web

```
import urllib.request  
  
req = urllib.request.Request(url='  
https://localhost/cgi-bin/test.cgi', data=b'Datos pasados a  
stdin del CGI')  
  
f = urllib.request.urlopen(req)  
print(f.read().decode('utf-8'))
```

_____ En el servidor:

```
#!/usr/bin/env python  
  
import sys  
  
datos = sys.stdin.read()  
  
print('Content-type: text-plain\n\nDatos:', datos)
```

Autenticación HTTP

```
import urllib.request  
  
# OpenerDirector con autenticación HTTP  
  
auth_handler = urllib.request.HTTPBasicAuthHandler()  
auth_handler.add_password(realm='PDQ Application',  
                           uri='https://mahler:8092/site-updates.py',  
                           user='klem', passwd='kadidd!ehopper')  
  
opener = urllib.request.build_opener(auth_handler)  
# instalado para poder usarse con urlope  
  
urllib.request.install_opener(opener)  
  
urllib.request.urlopen('http://www.example.com/login.html')
```

Guardar estructuras de datos

```
lista = ['text1', 'text2']
a = [[1.3, lista], 'texto']
f = open('tmp.dat', 'w')

# convertimos los datos a string
f.write(str(a)) # o f.write(repr(a))
f.close()
```

Leer estructuras de datos

```
f = open('tmp.dat', 'r')  
  
datos = eval(f.readline())  
# [[1.3, ['text1', 'text2']], 'texto']  
# a = eval(repr(a))
```

Guardar datos en binario

- Usando cadenas

```
myFile = open('datos.bin', 'wb')  
myFile.write("\x5F\x9D\x3E");  
myFile.close()
```

- Usando el módulo struct

```
st = struct.pack(<formato>, <dato1>, <dato2>, ...)
```

- Ejemplo:

```
with open('prueba.bin', 'wb') as f :
```

```
    st= struct.pack('2d', 1.345, 3.456)
```

```
    f.write(st)
```

Recuperar datos en binario

- Usando `ord()`

```
ch1 = myFile.read(1)  # read 1 byte  
d1 = ord(ch1)
```

- Usando el módulo `struct`

```
lista = struct.unpack(<formato>, <datos>)
```

- Ejemplo:

```
with open('prueba.bin', 'rb') as f :
```

```
    datos = f.read()
```

```
    valores = struct.unpack('2d', datos[0:8*2])
```

```
    print (valores[0], valores[1])
```

Guardar datos completos

- Se debe usar el módulo pickle o cPickle (más eficiente)

cPickle.dump(<variable>, <fichero>)

```
>>> cPickle.dump(L, f)
```

- Se puede recuperar

<variable> = cPickle.load(<fichero>)

```
>>> L = cPickle.load(f)
```

- Se recupera el tipo y contenido de la variable/objeto

Trabajar con datos en disco

- Permite manejar los datos sin cargarlos.

```
import shelve  
  
database = shelve.open('datos_shelve.bin')  
  
database['a1'] = 1234  
  
database['a2'] = 543  
  
database['a3'] = 234  
  
database['a123'] = (1234, 543, 234)
```

```
if 'a1' in database:  
    a1 = database['a1']
```

```
del database['a2']  
database.close()
```

Programación Funcional

Programación funcional

- Python usa funciones de orden superior (similar a Scheme). Son funciones con funciones como parámetros:
 - `map`
 - `reduce`
 - `filter`
- Las funciones lambda se suelen utilizar como parámetros.

Programación funcional: map

- `map(<función>, <objeto iterable>, ...)`
- Se aplica la función a cada uno de los elementos del objeto y crea una secuencia con los resultados.
- Hay que hacer un cambio de tipo a `list()` (porque devuelve un iterador)

```
nums = [0, 4, 7, 2, 1, 0 , 9 , 3, 5, 6, 8, 0, 3]
nums = list( map(lambda x : x % 5, nums) )
print(nums)
# [0, 4, 2, 2, 1, 0, 4, 3, 0, 1, 3, 0, 3]
```

Programación funcional I

```
def cuadrado(x):
    return x*x
>>> list( map(cuadrado, range(10,20)) )
[100, 121, 144, 169, 196, 225, 256, 289, 324, 361]
```

- Muchos programadores de python prefieren usar comprensión de listas

Programación funcional II

- Usando comprensión de listas

```
[ 2*x+1 for x in [10, 20, 30] ]
```

- Usando programación funcional

```
list( map( lambda x: 2*x+1, [10, 20, 30] ) )
```

Ejemplo map

¿Qué hace el siguiente código?

```
sec = range(8)
```

```
map(lambda x,y : (x,y), sec, map(lambda x:  
x*x, sec))
```

```
[ (0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25), (6, 36), (7, 49)]
```

Ejemplo map

a = [1,2,3,4]

b = [17,12,11,10]

c = [-1,-4,5,9]

map(lambda x,y:x+y, a,b)

[18, 14, 14, 14]

map(lambda x,y,z:x+y+z, a,b,c)

[17, 10, 19, 23]

map(lambda x,y,z:x+y-z, a,b,c)

[19, 18, 9, 5]

Programación funcional: problema

Dada una lista de puntos 3D en forma de tuplas, crea una lista con la distancia de cada uno de ellos al origen de coordenadas

Usando bucles

- distancia(x, y, z) = $\sqrt{x^2 + y^2 + z^2}$
- iterar sobre la lista y añadir los resultados a la nueva lista

Programación funcional: problema

```
from math import sqrt
```

```
puntos = [(2, 1, 3), (5, 7, -3), (2, 4, 0), (9, 6, 8)]
```

```
def distancia(punto) :  
    x, y, z = punto  
    return sqrt(x**2 + y**2 + z**2)
```

```
distancias = list(map(distancia, puntos))
```

Programación funcional: filter

`filter(<función>, <objeto iterable>)`

- Procesa cada elemento del objeto iterable aplicándole la función.
- Si la función devuelve True para ese elemento, entonces es devuelto por la función.
- En versiones anteriores a la 3 se devuelve una lista
- En la versión 3 es necesario hacer un cambio de tipo a `list()`

Programación funcional: filter

```
nums = [0, 4, 7, 2, 1, 0 , 9 , 3, 5, 6, 8, 0, 3]
nums = list(filter(lambda x : x != 0, nums))
```

```
print(nums)      #[4, 7, 2, 1, 9, 3, 5, 6, 8, 3]
```

```
def par(x):
    return 0 == x % 2
cuadrado = lambda x : x*x
list( map(cuadrado, filter(par, range(10,20))) )
[100, 144, 196, 256, 324]
```

Programación funcional: problema

```
NaN = float("nan")
scores = [[NaN, 12, .5, 78, math.pi],
          [2, 13, .5, .7, math.pi / 2],
          [2, NaN, .5, 78, math.pi],
          [2, 14, .5, 39, 1 - math.pi]]
```

Dada una lista de listas que contiene respuestas a un examen, filtra aquellas preguntas para las que no hay respuesta (indicada por NaN)

Programación funcional: problema

```
NaN = float("nan")
puntuaciones = [[NaN, 12, .5, 78, pi],[2, 13, .5, .7, pi / 2],
                [2,NaN, .5, 78, pi],[2, 14, .5, 39, 1 - pi]]
#solución 1
def es_NaN(respuestas) :
    for num in respuestas :
        if isnan(float(num)) :
            return False
    return True
valid = list(filter(es_NaN, puntuaciones))
print(valid)
#solución 2
valid = list(filter(lambda x : NaN not in x, puntuaciones))
print(valid)
```

Programación funcional: reduce

`reduce(<función>, <objeto iterable>[,inicializador])`

- Se aplica a cada elemento del objeto iterable junto con la suma hasta ese momento y realiza una suma acumulada
- **función** debe tener dos argumentos
- Si existe el inicializador, se usará como primer argumento de la suma
- En python 3 reduce() necesita importar un módulo:
`from functools import reduce`

Programación funcional: reduce

```
def mas(x,y):  
    return (x + y)  
from functools import reduce  
lista = [1,2,3,4,5,6]  
reduce(mas, lista)
```

21

```
def mas(x,y):  
    return (x + y)  
from functools import reduce  
lista = ['h','e','l','l','o']  
reduce(mas, lista)  
'hello'
```

Programación funcional: reduce

```
nums = [1, 2, 3, 4, 5, 6, 7, 8]
```

```
from functools import reduce
```

```
nums = list(reduce(lambda x, y : (x, y), nums))
```

```
print(nums) #((((((1, 2), 3), 4), 5), 6), 7), 8)
```

Programación funcional: problema

Dada una lista de números, calcula su media usando
reduce()

Usando bucles:

- sumar todos los elementos de la lista
- dividir la suma por la longitud de la lista

Programación funcional: media

```
nums = [92, 27, 63, 43, 88, 8, 38, 91, 47, 74, 18, 16,  
       29, 21, 60, 27, 62, 59, 86, 56]
```

```
media = reduce(lambda x, y : x + y, nums) / len(nums)
```

Ejemplos de programación funcional

```
def pot(x, y):  
    return x*y**2
```

```
print (reduce(pot, [1,2,3,4,5,6]))
```

```
map(lambda x: x*x*x, range(1, 11))
```

¿Qué hace el siguiente código?

```
f = lambda a,b: a if (a > b) else b  
reduce(f, [47,11,42,102,13])
```

map reduce

Problema: Dado un email determina si es spam

Solución: Cuenta las ocurrencias de ciertas palabras y si aparecen con demasiada frecuencia, es considerado spam.

map reduce

```
email = ['the', 'this', 'annoy', 'the', 'the', 'annoy']
```

```
def inEmail (x):  
    if (x == "the"):  
        return 1;  
    else:  
        return 0;
```

```
map(inEmail, email)      #[1, 0, 0, 0, 1, 1, 0]
```

```
reduce((lambda x, xs: x + xs), map(inEmail, email)) #3
```

Manejo de errores

Manejo de errores

Solución básica para que no ocurran errores:
evitar que se realice la operación

```
c = [1,"2",2,3,4,5.2]
sumOfc = 0
for numero in c:
    if isinstance(numero,int):
        sumOfc += numero

if "dato" in B:
    A["dato"] = B["dato"]
```

Manejo de excepciones

O bien se puede manejar el error una vez producido (**try except**)

```
sumOfc = ""  
for number in c:  
    try:  
        sumOfc += number  
    except TypeError: # indicando cada opción  
        pass  
    try:  
        A["dato"] = B["dato"]  
    except KeyError:  
        pass
```

Excepciones

- Las excepciones son eventos que pueden modificar el flujo de un programa.
- Se lanzan automáticamente con cada error.
- **try/except**: capturan y recuperan una excepción generada por el programador o por Python.
- **try/finally**: realiza acciones si la excepción se produce o no.
- **raise**: genera una excepción manualmente.
- **assert**: genera una excepción de forma condicional.

Tipos de excepciones

- Manejo de errores
 - Cada vez que Python detecta un error, lanza una excepción.
 - Por defecto se detiene la ejecución.
 - En otro caso, se trata de capturar y recuperar la excepción
- Notificación de eventos
 - Puede enviar una señal indicando una situación válida (por ejemplo en una búsqueda)
- Manejo de casos especiales
 - Maneja situaciones inusuales.
- Finalización de acciones
 - Garantiza que se termina una acción (try/finally)
- Control de flujo inusual
 - Un tipo de “goto” de alto nivel

Ejemplo I

```
>>> print (3/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

```
def division(x, y) :
    try :
        return x/y
    except ZeroDivisionError :
        print "división por cero"
```

Ejemplo II

```
def division(x, y) :  
    try :  
        If y == 0 :  
            raise 'cero'  
        return x/y  
    except 'cero' :  
        print "división por cero"
```

Ejemplo III

```
n = int(input("Introduce un entero: "))
Introduce un entero: 23.5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '23.5'"
```

```
while not Terminar ::  
    try:  
        n = int(input('Introduce un entero: '))  
        Terminar = True  
    except ValueError:  
        print('no es un entero! Inténtalo de nuevo')
```

try / except / else

```
try:  
    <bloque de sentencias>      #código principal  
except <nombre1>:  
    <bloque de sentencias>  
except <nombre2>,<datos>:  
    <bloque de sentencias>  
except (<nombre3>,<nombre4>):  
    <bloque de sentencias>  
except:  
    <bloque de sentencias>  
else:          # opcional, si no hay excepción  
    <bloque de sentencias>
```

Ejemplo

try:

```
<bloque>
except NameError(): ...
except IndexError(): ...
except KeyError(): ...
except (AttributeError,TypeError,SyntaxError):...
```

else:

- Capturar todas las excepciones: **except** vacío.
- No es recomendable porque evita que podamos encontrar errores.
- **else** se realiza si no hay ninguna excepción

try / finally

Con **finally** se realiza el bloque, se produzca o no una excepción

```
try:  
    <bloque>  
finally:  
    <bloque>
```

- Garantiza que algo se haga pase lo que pase.

Ejemplo I

```
>>> try:  
>>> print (3/0)  
>>> finally: print "Terminado"  
Terminado  
Traceback...  
....  
ZeroDivisionError: integer division...
```

```
>>> try:  
>>>     try:  
>>>         print (3/0)  
>>>     except ZeroDivisionError : print ("Excepción")  
>>> finally: print ("Terminado")
```

Excepción
Terminado

Ejemplo II

```
>>> f = open("datos.txt", 'r')
>>> try:
>>> ...
>>> finally:
>>>   f.close()
```

Traceback (most recent call last):

```
  File "<stdin>", line 2, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'datos.txt'
```

raise

Permite lanzar de forma explícita una excepción

raise <nombre>

raise <nombre>,<datos># proporciona datos al manejador

raise #reenvía la última excepción

>>>try:

 raise 'cero', (3,0)

except 'cero': print "argumento cero"

except 'cero', datos: print(datos)

Ejemplo de raise

La última opción es útil si se quiere propagar la excepción capturada a otro manejador

```
try:  
    raise InterrupcionTeclado  
except:  
    print ("propagar")  
    raise
```

```
propagar  
Traceback (most recent call last):  
  File "<stdin>", line 2, in <module>  
NameError: name 'InterrupcionTeclado' is not defined
```

Ejemplo con múltiples excepciones

```
import sys

try:
    f = open('integers.txt')
    s = f.readline() # uno por línea
    i = int(s.strip())
except IOError as (errno, strerror):
    print "I/O error({0}): {1}".format(errno, strerror)
except ValueError:
    print "No valid integer in line."
except:
    print "Unexpected error:", sys.exc_info()[0]
    raise
```

assert

Es como **raise** pero de forma condicional

assert <condicion>, <datos>

assert <condicion>

Si la condición es falsa se lanza una excepción **AssertionError**

```
def f(x,y)
    assert x>0, 'x debe ser positivo'
    assert y<0, 'y debe ser negativo'
    return y**x
```

Cuestiones adicionales

- Todos los errores son excepciones pero no todas las excepciones son errores. Pueden ser señales o avisos

`while True:`

```
    try: line = input()
    except EOFError: break
    else: # procesa siguiente línea
```

- Se puede enviar una señal con **raise** para distinguir si es correcto o erróneo
- Trazar errores

`try:`

```
    ... # ejecutar programa
```

```
except: import sys; print sys.exc_type,sys.exc_value
```

Ejemplo

```
import sys
try:
    f = open('enteros.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as (errno, strerror):
    print "I/O error({0}): {1}".format(errno, strerror)
except ValueError:
    print "No valid integer in line."
except:
    print "Unexpected error:", sys.exc_info()[0]
    raise
```

Cuestiones de diseño

- Se usa try en operaciones que usualmente pueden fallar como apertura de ficheros y llamadas a sockets
- Es posible que interese que el programa muera en ciertas situaciones
- Se usa *try/finally* para garantizar su ejecución.
- Es preferible usar una única instrucción try con varios casos en lugar de varios try.
- La vida de una excepción termina cuando es capturada.

Información sobre la excepción

```
try:  
    raise NotImplementedError("No error")  
except Exception as e:  
    exc_type, exc_obj, exc_tb = sys.exc_info()  
    fname =  
    os.path.split(exc_tb.tb_frame.f_code.co_filename)[1]  
    print(exc_type, fname, exc_tb.tb_lineno)
```

Unicode

Unicode

- Los strings en ACSII tienen 7 bits. No es suficiente para expresar toda la información en distintos idiomas.
- Existen distintas codificaciones, en particular se usan uno o más bytes.
- Los strings Unicode proporcionan una codificación común a todos los idiomas.
- Es posible convertir información entre Unicode y otras codificaciones

Ejemplo de Unicode

```
>>> s = u'blå' # Carácter noruego å
>>> s.encode() # convierte a ASCII
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
UnicodeEncodeError: 'ascii' codec can't encode
  character u'\xe5' in position 2: ordinal not in range(128)
>>> s.encode('utf-8') # convierte aUTF-8
'b\xc3\xa5'
>>> s.encode('latin-1')
'b\xe5'
>>> s.encode('utf-16')
'\xff\xfeb\x00\x00\xe5\x00'
```

Ejemplo de Unicode y ficheros

```
import codecs  
fi = codecs.open(filename, 'r', encoding='utf-16')  
fo = codecs.open(filename, 'w', encoding='utf-8')  
for line in fi:  
    # line es un string unicode  
    fo.write(line) # convierte automáticamente a UTF-8
```

Generadores, anotaciones y cálculo simbólico

Generadores I

- Permiten iterar sobre un conjunto de elementos una única vez
- No se guardan los elementos

```
def pares(lista):
```

```
    for i in lista:
```

```
        if i%2==0:
```

```
            yield i
```

Generadores II

```
lista = [1, 2, 3, 4, 5]
```

```
nums = pares(lista)
```

```
for i in nums:
```

```
    print (i)
```

Generadores III

```
>>> def generador(n):
    yield n
    yield n + 1
```

```
>>> g = generador(6)
>>> next(g)
```

6

```
>>> next(g)
```

7

```
>>> next(g)
```

Traceback (most recent call last):

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

StopIteration

Generadores IV

Se pueden generar secuencias infinitas

```
>>> def fib():
...     a, b = 0, 1
...     while True:
...         yield a
...         a, b = b, a + b
...
>>> import itertools
>>> list(itertools.islice(fib(), 10))
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Generar infinitos datos

```
from time import gmtime, strftime  
def generador():  
    while True:  
        yield strftime("%a, %d %b %Y %H:%M:%S +0000",  
                      gmtime())  
generadorInstancia = generador()  
next(generadorInstancia)
```

Tipo de dato generador

```
>>> g = (x for x in range(10))  
>>> g  
<generator object <genexpr> at 0x000000005693558>
```

Anotaciones en funciones

- Ejemplo:

```
def posint(n: int) -> bool:  
    return n > 0
```

- Se asocia un atributo llamado `__annotations__` que corresponde al diccionario
`{'n': <class 'int'>, 'return': <class 'bool'>}`
- En PEP (Python Enhancement Proposal) aparecen casos de uso que incluyen chequeo de tipos.

Cálculo simbólico

```
>>> from sympy import *
>>> x = symbols('x')
>>> a = Integral(cos(x)*exp(x), x)
>>> Eq(a, a.doit())
Integral(exp(x)*cos(x), x) == exp(x)*sin(x)/2 +
exp(x)*cos(x)/2
```