

Map eficiente usando mprocessing

```
import multiprocessing
```

```
import math
```

```
pool = multiprocessing.Pool()
```

```
print (pool.map(math.sqrt, range(100)))
```

```
pool.close()
```

Manejo de excepciones

Manejo de excepciones (I)

Solución básica para que no ocurran errores:
evitar que se realice la acción que provoca el error

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

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

© Prof.Miguel García Silvente

193

Manejo de excepciones (II)

O bien se puede manejar el error una vez producido

```
suma_de_c = 0
for numero in c:
    try:
        suma_de_c += numero
    except TypeError: # indicando cada opción
        pass
try:
    A["dato"] = B["dato"]
except KeyError:
    pass
```

© Prof.Miguel García Silvente

194

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')
```

© Prof. Miguel García Silvente

199

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>
```

© Prof. Miguel García Silvente

200

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

© Prof.Miguel García Silvente

201

try / finally

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

try:

```
<bloque-try>
```

finally:

```
<bloque-finally>
```

Garantiza que algo se haga pase lo que pase.

Ejemplo try/finally (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

© Prof.Miguel García Silvente

203

Ejemplo try/finally (II)

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

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('exterros.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 hay un entero en la línea")
except:
    print ("Error:inesperado", sys.exc_info()[0])
    raise
```

© Prof.Miguel García Silvente

207

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
```

© Prof.Miguel García Silvente

208

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
```

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

- 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 a UTF-8
'bl\xc3\xa5'
>>> s.encode('latin-1')
'bl\xe5'
>>> s.encode('utf-16')
'\xff\xfebl\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  
f.close()
```

Generadores, anotaciones y cálculo simbólico

Generadores (I)

- Permiten iterar sobre un conjunto de elementos una única vez
- No se guardan los elementos, se producen con `yield`

```
def pares(lista):
    for i in lista:
        if i%2==0:
            yield i
```

© Prof.Miguel García Silvente

217

Generadores (II)

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

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

```
for i in nums:
    print (i)
```

© Prof.Miguel García Silvente

218

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
```

© Prof. Miguel García Silvente

219

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]
```

© Prof. Miguel García Silvente

220

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)
```

© Prof.Miguel García Silvente

221

Tipo de dato generador

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

© Prof.Miguel García Silvente

222

Anotaciones en funciones (I)

- Ejemplo:

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

- `posint("hola")`

Traceback (most recent call last):

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

File "<stdin>", line 2, in posint

`TypeError: '>' not supported between
instances of 'str' and 'int'`

© Prof. Miguel García Silvente

223

Anotaciones en funciones (II)

- 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: módulo sympy

```
import sympy as sym  
a = sym.Rational(1, 2)  
b = sym.Rational(1, 3)  
print(a+b)
```

5/6

Valores como

- pi: sympy.pi
- Infinito: sympy.oo

Evaluar una valor: evalf

```
>>> sympy.exp(1).evalf()  
2.71828182845905  
>>> sympy.exp(1).evalf(2)  
2.7  
>>> sympy.exp(1).evalf(3)  
2.72  
>>> sympy.exp(1).evalf(123)  
2.71828182845904523536028747135266249775724709369  
9959574966967627724076630353547594571382178525166  
42742746639193200305992182
```

Declaración de símbolos: symbol

```
x = sym.Symbol('x')
```

```
y = sym.Symbol('y')
```

```
print(x + y + x - y)
```

```
2*x
```

```
print((x + y) ** 2)
```

```
(x + y) ** 2
```

Expansión de expresiones: expand

```
x = sym.Symbol('x')
```

```
y = sym.Symbol('y')
```

```
print(sym.expand((x + y) ** 2))
```

```
x**2 + 2*x*y + y**2
```

```
print(sym.expand((x - y) ** 3))
```

```
x**3 - 3*x**2*y + 3*x*y**2 - y**3
```

Simplificación: simplify

```
x = sym.Symbol('x')
y = sym.Symbol('y')
print(sym.simplify((6*x + 2*x * y) / x))
2*y + 6
```

© Prof.Miguel García Silvente

229

Cálculo de límites: limit

```
import sympy as sym
x = sym.symbols('x')
print(sym.limit(sym.sin(x) / x, x, 0))
exp(x)*sin(x) + exp(x)*cos(x)

1
print(sym.limit(sym.sin(x) / x, x, sym.oo))
0
```

© Prof.Miguel García Silvente

230

Diferenciación: diff

```
import sympy as sym  
x = sym.symbols('x')  
print(sym.diff(sym.sin(x)*sym.exp(x), x))  
exp(x)*sin(x) + exp(x)*cos(x)  
print(sym.diff(x*sym.log(x) - x, x))  
log(x)  
print(sym.diff(x**5, x, 2))  
20*x**3
```

© Prof. Miguel García Silvente

231

Descomposición serie de Taylor: series

```
import sympy as sym  
x = sym.symbols('x')  
print(sym.series(sym.exp(x), x))  
1 + x + x**2/2 + x**3/6 + x**4/24 + x**5/120 +  
O(x**6)
```

© Prof. Miguel García Silvente

232

Integración: integrate

```
import sympy as sym  
x = sym.symbols('x')  
print(sym.integrate(sym.cos(x)*sym.exp(x), x)  
exp(x)*sin(x)/2 + exp(x)*cos(x)/2  
sym.integrate(sym.log(x), x)  
x*log(x) - x
```

Resolver sistemas de ecuaciones: solve

```
import sympy as sym  
x = sym.symbols('x')  
y = sym.symbols('y')  
solucion = sym.solve((x + 5 * y - 2, -3 * x + 6 *  
y - 15), (x, y))  
print(solucion)  
{x: -3, y: 1}
```

Álgebra lineal

- Matrices
- Ecuaciones diferenciales