Ordenamiento de listas

Las listas se pueden ordenar fácilmente usando la función sorted:

In [ ]:
```
lista_de_numeros = [1, 6, 3, 9, 5, 2]
lista_ordenada = sorted(lista_de_numeros)
print lista_ordenada
```

Pero, ¿y cómo hacemos para ordenarla de mayor a menor?.
Simple, interrogamos un poco a la función:

```
>>> print sorted.__doc__
sorted(iterable, cmp=None, key=None, reverse=False) --> new so
rted list
```

Entonces, con sólo pasarle el parámetro de *reverse* en True debería alcanzar:

In [ ]:
```
lista_de_numeros = [1, 6, 3, 9, 5, 2]
print sorted(lista_de_numeros, reverse=True)
```

¿Y si lo que quiero ordenar es una lista de registros?.
Podemos pasarle una función que sepa cómo comparar esos registros o una que sepa devolver la información que necesita comparar.

In [ ]:
```

```

```
import random

def crear_alumnos(cantidad_de_alumnos=5):
    nombres = ['Javier', 'Pablo', 'Ramiro', 'Lucas', 'Carlos']
    apellidos = ['Saviola', 'Aimar', 'Funes Mori', 'Alario', 'Sanchez'

    alumnos = []
    for i in range(cantidad_de_alumnos):
        a = {
            'nombre': '{}, {}'.format(
                random.choice(apellidos), random.choice(nombres)),
            'padron': random.randint(90000, 100000),
            'nota': random.randint(4, 10)
        }
        alumnos.append(a)

    return alumnos


def imprimir_curso(lista):
    for idx, x in enumerate(lista, 1):
        print '    {pos:2}. {padron} - {nombre}: {nota}'.format(
            pos=idx, **x)


def obtener_padron(alumno):
    return alumno['padron']


def ordenar_por_padron(alumno1, alumno2):
    if alumno1['padron'] < alumno2['padron']:
        return -1
    elif alumno2['padron'] < alumno1['padron']:
        return 1
    else:
        return 0

curso = crear_alumnos()
print 'La lista tiene los alumnos:'
imprimir_curso(curso)

lista_ordenada = sorted(curso, key=obtener_padron)
print 'Y la lista ordenada por padrón:'
imprimir_curso(lista_ordenada)

otra_lista_ordenada = sorted(curso, cmp=ordenar_por_padron)
print 'Y la lista ordenada por padrón:'
imprimir_curso(otra_lista_ordenada)
```

Búsquedas en listas

Para saber si un elemento se encuentra en una lista, alcanza con usar el operador **in**:

In [ ]:
```
lista = [11, 4, 6, 1, 3, 5, 7]

if 3 in lista:
    print '3 esta en la lista'
else:
    print '3 no esta en la lista'

if 15 in lista:
    print '15 esta en la lista'
else:
    print '15 no esta en la lista'
```

También es muy fácil saber si un elemento **no** esta en la lista:

In [ ]:
```
lista = [11, 4, 6, 1, 3, 5, 7]

if 3 not in lista:
    print '3 NO esta en la lista'
else:
    print '3 SI esta en la lista'
```

En cambio, si lo que queremos es saber es dónde se encuentra el número 3 en la lista es:

In [ ]:
```
lista = [11, 4, 6, 1, 3, 5, 7]

pos = lista.index(3)
print 'El 3 se encuentra en la posición', pos

pos = lista.index(15)
print 'El 15 se encuentra en la posición', pos
```

Funciones anónimas

Hasta ahora, a todas las funciones que creamos les poníamos un nombre al momento de crearlas, pero cuando tenemos que crear funciones que sólo tienen una línea y no se usen en una gran cantidad de lugares se pueden usar las funciones lambda:

In [ ]:
```
help("lambda")
```

In [ ]:
```
mi_funcion = lambda x, y: x+y

resultado = mi_funcion(1,2)
print resultado
```

Si bien no son funciones que se usen todos los días, se suelen usar cuando una función recibe otra función como parámetro (las funciones son un tipo de dato, por lo que se las pueden asignar a variables, y por lo tanto, también pueden ser parámetros). Por ejemplo, para ordenar los alumnos por padrón podríamos usar:

```
sorted(curso, key=lambda x: x['padron'])
```

Ahora, si quiero ordenar la lista anterior por nota decreciente y, en caso de igualdad, por padrón podríamos usar:

In [ ]:
```
curso = crear_alumnos(15)
print 'Curso original'
imprimir_curso(curso)

lista_ordenada = sorted(curso, key=lambda x: (-x['nota'], x['padron']))
print 'Curso ordenado'
imprimir_curso(lista_ordenada)
```

Otro ejemplo podría ser implementar una búsqueda binaria que permita buscar tanto en listas crecientes como decrecientes:

In [ ]:
```

```

```
es_mayor = lambda n1, n2: n1 > n2
es_menor = lambda n1, n2: n1 < n2


def binaria(cmp, lista, clave):
    """Binaria es una función que busca en una lista la clave pasada.
    Es un requisito de la búsqueda binaria que la lista se encuentre
    ordenada, pero no si el orden es ascendente o descendente. Por
    este motivo es que también recibe una función que le indique en
    que sentido ir.
    Si la lista está ordenada en forma ascendente la función que se
    le pasa tiene que ser verdadera cuando el primer valor es mayor
    que la segundo; y falso en caso contrario.
    Si la lista está ordenada en forma descendente la función que se
    le pasa tiene que ser verdadera cuando el primer valor es menor
    que la segundo; y falso en caso contrario.
    """
    min = 0
    max = len(lista) - 1
    centro = (min + max) / 2
    while (lista[centro] != clave) and (min < max):
        if cmp(lista[centro], clave):
            max = centro - 1
        else:
            min = centro + 1
        centro = (min + max) / 2
    if lista[centro] == clave:
        return centro
    else:
        return -1

print binaria(es_mayor, [1, 2, 3, 4, 5, 6, 7, 8, 9], 8)
print binaria(es_menor, [1, 2, 3, 4, 5, 6, 7, 8, 9], 8)
print binaria(es_mayor, [1, 2, 3, 4, 5, 6, 7, 8, 9], 123)

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

Excepciones

Una excepción es la forma que tiene el intérprete de que indicarle al programador y/o usuario que ha ocurrido un error. Si la excepción no es controlada por el desarrollador ésta llega hasta el usuario y termina abruptamente la ejecución del sistema.
Por ejemplo:

In [ ]:
```
print 1/0
```

Pero no hay que tenerle miedo a las excepciones, sólo hay que tenerlas en cuenta y controlarlas en el caso de que ocurran:

In [ ]:
```
dividendo = 1
divisor = 0
print 'Intentare hacer la división de %d/%d' % (dividendo, divisor)
try:
    resultado = dividendo / divisor
    print resultado
except ZeroDivisionError:
    print 'No se puede hacer la división ya que el divisor es 0.'
```

Pero supongamos que implementamos la regla de tres de la siguiente forma:

In [ ]:
```
def dividir(x, y):
    return x/y

def regla_de_tres(x, y, z):
    return dividir(z*y, x)


# Si de 28 alumnos, aprobaron 15, el porcentaje de aprobados es de...
porcentaje_de_aprobados = regla_de_tres(28, 15, 100)
print 'Porcentaje de aprobados: %0.2f %%' % porcentaje_de_aprobados
```

En cambio, si le pasamos 0 en el lugar de x:

In [ ]:
```
resultado = regla_de_tres(0, 13, 100)
print 'Porcentaje de aprobados: %0.2f %%' % resultado
```

Acá podemos ver todo el *traceback* o *stacktrace*, que son el cómo se fueron llamando las distintas funciones entre sí hasta que llegamos al error.
Pero no es bueno que este tipo de excepciones las vea directamente el usuario, por lo que podemos controlarlas en distintos momentos. Se pueden controlar inmediatamente donde ocurre el error, como mostramos antes, o en cualquier parte de este *stacktrace*.
En el caso de la regla_de_tres no nos conviene poner el try/except encerrando la línea x/y, ya que en ese punto no tenemos toda la información que necesitamos para informarle correctamente al usuario, por lo que podemos ponerla en:

In [ ]:
```
def dividir(x, y):
    return x/y

def regla_de_tres(x, y, z):
    resultado = 0
    try:
        resultado = dividir(z*y, x)
    except ZeroDivisionError:
        print 'No se puede calcular la regla de tres ' \
              'porque el divisor es 0'

    return resultado

print regla_de_tres(0, 1, 2)
```

Pero en este caso igual muestra 0, por lo que si queremos, podemos poner los try/except incluso más arriba en el stacktrace:

In [ ]:
```
def dividir(x, y):
    return x/y

def regla_de_tres(x, y, z):
    return dividir(z*y, x)

try:
    print regla_de_tres(0, 1, 2)
except ZeroDivisionError:
    print 'No se puede calcular la regla de tres ' \
          'porque el divisor es 0'
```

Todos los casos son distintos y no hay UN lugar ideal dónde capturar la excepción; es cuestión del desarrollador decidir dónde conviene ponerlo para cada problema.
Incluso, una única línea puede lanzar distintas excepciones, por lo que capturar un tipo de excepción en particular no me asegura que el programa no pueda lanzar un error en esa línea que supuestamente es segura:

Capturar múltiples excepciones

En algunos casos tenemos en cuenta que el código puede lanzar una excepción como la de `ZeroDivisionError`, pero eso puede no ser suficiente:

In [ ]:
```
def dividir_numeros(x, y):
    try:
        resultado = x/y
        print 'El resultado es: %s' % resultado
    except ZeroDivisionError:
        print 'ERROR: Ha ocurrido un error por mezclar tipos de datos'

dividir_numeros(1, 0)
dividir_numeros(10, 2)
dividir_numeros("10", 2)
```

En esos casos podemos capturar más de una excepción de la siguiente forma:

In [ ]:
```
def dividir_numeros(x, y):
    try:
        resultado = x/y
        print 'El resultado es: %s' % resultado
    except TypeError:
        print 'ERROR: Ha ocurrido un error por mezclar tipos de datos'
    except ZeroDivisionError:
        print 'ERROR: Ha ocurrido un error de división por cero'
    except Exception:
        print 'ERROR: Ha ocurrido un error inesperado'

dividir_numeros(1, 0)
dividir_numeros(10, 2)
dividir_numeros("10", 2)
```

Incluso, si queremos que los dos errores muestren el mismo mensaje podemos capturar ambas excepciones juntas:

In [ ]:
```
def dividir_numeros(x, y):
    try:
        resultado = x/y
        print 'El resultado es: %s' % resultado
    except (ZeroDivisionError, TypeError):
        print 'ERROR: No se puede calcular la división'

dividir_numeros(1, 0)
dividir_numeros(10, 2)
dividir_numeros("10", 2)
```

Jerarquía de excepciones

Existe una jerarquía de excepciones (https://docs.python.org/2/library/exceptions.html), de forma que si se sabe que puede venir un tipo de error, pero no se sabe exactamente qué excepción puede ocurrir siempre se puede poner una excepción de mayor

jerarquía:

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StandardError
      |    +-- BufferError
      |    +-- ArithmeticError
      |    |    +-- FloatingPointError
      |    |    +-- OverflowError
      |    |    +-- ZeroDivisionError
      |    +-- AssertionError
      |    +-- AttributeError
      |    +-- EnvironmentError
      |    |    +-- IOError
      |    |    +-- OSError
      |    |         +-- WindowsError (Windows)
      |    |         +-- VMSError (VMS)
      |    +-- EOFError
      |    +-- ImportError
      |    +-- LookupError
      |    |    +-- IndexError
      |    |    +-- KeyError
      |    +-- MemoryError
      |    +-- NameError
      |    |    +-- UnboundLocalError
      |    +-- ReferenceError
      |    +-- RuntimeError
      |    |    +-- NotImplementedError
      |    +-- SyntaxError
      |    |    +-- IndentationError
      |    |         +-- TabError
      |    +-- SystemError
      |    +-- TypeError
      |    +-- ValueError
      |         +-- UnicodeError
      |              +-- UnicodeDecodeError
      |              +-- UnicodeEncodeError
      |              +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning
           +-- PendingDeprecationWarning
           +-- RuntimeWarning
           +-- SyntaxWarning
           +-- UserWarning
           +-- FutureWarning
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
```

Por lo que el error de división por cero se puede evitar como:

In [ ]:
```
try:
    print 1/0
except ZeroDivisionError:
    print 'Ha ocurrido un error de división por cero'
```

Y también como:

In [ ]:
```
try:
    print 1/0
except Exception:
    print 'Ha ocurrido un error inesperado'
```

Si bien siempre se puede poner Exception en lugar del tipo de excepción que se espera, no es una buena práctica de programación ya que se pueden esconder errores indeseados. Por ejemplo, un error de sintaxis. Además, cuando se lanza una excepción en el bloque `try`, el intérprete comienza a buscar entre todas cláusulas `except` una que coincida con el error que se produjo, o que sea de mayor jerarquía. Por lo tanto, es recomendable poner siempre las excepciones más específicas al principio y las más generales al final:

```
def dividir_numeros(x, y):
    try:
        resultado = x/y
        print 'El resultado es: %s' % resultado
    except TypeError:
        print 'ERROR: Ha ocurrido un error por mezclar tipos d
e datos'
    except ZeroDivisionError:
        print 'ERROR: Ha ocurrido un error de división por cer
o'
    except Exception:
        print 'ERROR: Ha ocurrido un error inesperado'
```

Si el error no es capturado por ninguna clausula se propaga de la misma forma que si no se hubiera puesto nada.

Otras cláusulas para el manejo de excepciones

Además de las cláusulas `try` y `except` existen otras relacionadas con las excepciones que nos permiten manejar de mejor manera el flujo del programa:

- **else**: se usa para definir un bloque de código que se ejecutará **sólo si no ocurrió ningún error**.
- **finally**: se usa para definir un bloque de código que se ejecutará **siempre**, independientemente de si se lanzó una excepción o no.

In [ ]:
```
def dividir_numeros(x, y):
    try:
        resultado = x/y
        print 'El resultado es {}'.format(resultado)
    except ZeroDivisionError:
        print 'Error: División por cero'
    else:
        print 'Este mensaje se mostrará sólo si no ocurre ningún error
    finally:
        print 'Este bloque de código se muestra siempre'

dividir_numeros(1, 0)
print '-------------'
dividir_numeros(10, 2)
```

Pero entonces, ¿por qué no poner ese código dentro del `try-except`?. Porque tal vez no queremos capturar con las cláusulas `except` lo que se ejecute en ese bloque de código:

In [ ]:
```
def dividir_numeros(x, y):
    try:
        resultado = x/y
        print 'El resultado es {}'.format(resultado)
    except ZeroDivisionError:
        print 'Error: División por cero'
    else:
        print 'Ahora hago que ocurra una excepción'
        print 1/0
    finally:
        print 'Este bloque de código se muestra siempre'

dividir_numeros(1, 0)
print '-------------'
dividir_numeros(10, 2)
```

Lanzar `excepciones`

Hasta ahora vimos cómo capturar un error y trabajar con él sin que el programa termine abruptamente, pero en algunos casos somos nosotros mismos quienes van a querer lanzar una excepción. Y para eso, usaremos la palabra reservada `raise`:

In [ ]:
```
def dividir_numeros(x, y):
    if y == 0:
        raise Exception('Error de división por cero')

    resultado = x/y
    print 'El resultado es {0}'.format(resultado)

try:
    dividir_numeros(1, 0)
except ZeroDivisionError as e:
    print 'ERROR: División por cero'
except Exception as e:
    print 'ERROR: ha ocurrido un error del tipo Exception'

print '----------'
dividir_numeros(1, 0)
```

Crear excepciones

Pero así como podemos usar las excepciones estándares, también podemos crear nuestras propias excepciones:

```
class MiPropiaExcepcion(Exception):

    def __str__(self):
        return 'Mensaje del error'
```

Por ejemplo:

In [ ]:

```
class ExcepcionDeDivisionPor2(Exception):

    def __str__(self):
        return 'ERROR: No se puede dividir por dos'


def dividir_numeros(x, y):
    if y == 2:
        raise ExcepcionDeDivisionPor2()

    resultado = x/y

try:
    dividir_numeros(1, 2)
except ExcepcionDeDivisionPor2:
    print 'No se puede dividir por 2'

dividir_numeros(1, 2)
```

Para más información, ingresar a https://docs.python.org/2/tutorial/errors.html (https://docs.python.org/2/tutorial/errors.html)

Ejercicios

1. Se leen dos listas A y B, de N y M elementos respectivamente. Construir un algoritmo que halle las listas unión e intersección de A y B. Previamente habrá que ordenarlos.
2. Escribir una función que reciba una lista desordenada y un elemento, que:
   A. Busque todos los elementos coincidan con el pasado por parámetro y

devuelva la cantidad de coincidencias encontradas.
   B. Busque la primera coincidencia del elemento en la lista y devuelva su posición.
3. Escribir una función que reciba una lista de números no ordenada, que:
   A. Devuelva el valor máximo.
   B. Devuelva una tupla que incluya el valor máximo y su posición.
   C. ¿Qué sucede si los elementos son cadenas de caracteres?
   Nota: no utilizar `lista.sort()` ni la función `sorted`.
4. Se cuenta con una lista ordenada de productos, en la que uno consiste en una tupla de (identificador, descripción, precio), y una lista de los productos a facturar, en la que cada uno consiste en una tupla de (identificador, cantidad). Se desea generar una factura que incluya la cantidad, la descripción, el precio unitario y el precio total de cada producto comprado, y al final imprima el total general.
   Escribir una función que reciba ambas listas e imprima por pantalla la factura solicitada.
5. Leer de teclado (usando la función raw_input) los datos de un listado de alumnos terminados con padrón 0. Para cada alumno deben leer:
   # Padrón
   # Nombre
   # Apellido
   # Nota del primer parcial
   # Nota del primer recuperatorio (en caso de no haber aprobado el parcial)
   # Nota del segundo recuperatorio (en caso de no haber aprobado en el primero)
   # Nombre del grupo
   # Nota del TP 1
   # Nota del TP 2
   Si el padrón es 0, no deben seguir pidiendo el resto de los campos.
   Tanto el padrón, como el nombre y apellido deben leerse como strings (existen padrones que comienzan con una letra b), pero debe validarse que se haya ingresado algo de por lo menos 2 caracteres.
   Todas las notas serán números enteros entre 0 y 10, aunque puede ser que el usuario accidentalmente ingrese algo que no sea un número, por lo que deberán validar la entrada y volver a pedirle los datos al usuario hasta que ingrese algo válido. También deben validar que las notas pertenezcan al rango de 0 a 10.
   Se asume que todos los alumnos se presentan a todos los parciales hasta aprobar o completar sus 3 chances.
   Al terminar deben:
   A. imprimir por pantalla un listado de todos los alumnos en condiciones de rendir coloquio (último parcial aprobado y todos los TP aprobados) en el mismo orden en el que el usuario los ingreso.
   B. imprimir por pantalla un listado de todos los alumnos en condiciones de rendir coloquio (último parcial aprobado y todos los TP aprobados)

ordenados por padrón en forma creciente.

C. imprimir por pantalla un listado de todos los alumnos en condiciones de rendir coloquio (último parcial aprobado y todos los TP aprobados) ordenados por nota y, en caso de igualdad, por padrón (ambos en forma creciente).
D. Calcular para cada alumno el promedio de sus notas del parcial y luego el promedio del curso como el promedio de todos los promedios.
E. Informar cuál es la nota que más se repite entre todos los parciales (sin importar si es primer, segundo o tercer parcial) e indicar la cantidad de ocurrencias.
F. listar todas las notas que se sacaron los alumnos en el primer parcial y los padrones de quienes se sacaron esas notas con el siguiente formato:

```
Nota: 2
* nnnn1
* nnnn2
* nnnn3
* nnnn4
Nota: 4
* nnnn1
* nnnn2
...
```

Tener en cuenta que las notas pueden ser del 2 al 10 y puede ocurrir que nadie se haya sacado esa nota (y en dicho caso no esa nota no tiene que aparecer en el listado)

In [ ]: