

Python (II)

Algoritmia
Grado en Ingeniería Informática
Universidad de Burgos

Juan José Rodríguez Díez



Contenido

- 1 Entrada/salida simple
- 2 Excepciones
- 3 Iteradores y generadores



La función `print`

- Salida estándar.
- Número arbitrario de argumentos.
- Por defecto, imprime los argumentos separados por espacios y salto de línea.
- Los argumentos pueden no ser cadenas, para `x` se imprime `str(x)`.
- Argumentos con nombre:
 - Separador: `print(a, b, c, sep="␣:␣")`.
 - Terminador: `print(a, b, c, end="")`.
 - A un fichero con `file`.

[Goodrich et al., 2013, pág. 30]



La función `input`

- Muestra un mensaje opcional, espera a que el usuario introduzca una línea de caracteres.
- Devuelve una cadena, sin salto de línea.
- Convertir la cadena a otros tipos: `n = int(input("n:"))`

```
1 respuesta = input('Introduzca x_e_y, separados por espacios:')
2 valores = respuesta.split()
3 x = float(valores[0])
4 y = float(valores[1])
5 # alternativamente:
6 (x, y) = map(float, valores)
```

[Goodrich et al., 2013, págs. 30-31]



Ficheros

- Función **open**.
- Primer argumento el nombre del fichero:
`fp = open('datos.txt')`
- Por defecto, solo lectura.
- Segundo argumento (*mode*): 'r' para lectura, 'w' para escritura, 'a' para añadir, 'rb' y 'wb' para binarios.
- Método *close* para cerrar el fichero.

[Goodrich et al., 2013, págs. 31-32]



Operaciones sobre ficheros

<code>fp.read()</code>	Una cadena con el resto del fichero.
<code>fp.read(k)</code>	Una cadena con los siguientes <code>k</code> bytes.
<code>fp.readline()</code>	Cadena con el resto de la línea.
<code>fp.readlines()</code>	Devuelve una lista de cadenas.
<code>for line in fp:</code>	Itera sobre las líneas restantes.
<code>fp.seek(k)</code>	Va al byte <code>k</code> .
<code>fp.tell()</code>	Devuelve posición.
<code>fp.write(string)</code>	Escribe en la posición actual.
<code>fp.writelines(seq)</code>	Escribe las cadenas de la secuencia.
<code>print(..., file=fp)</code>	Redirige la salida de <code>print</code> al fichero

[Goodrich et al., 2013, pág. 32]



Excepciones

- Sucesos inesperados durante la ejecución.
- Objetos que se lanzan por código que encuentra una circunstancia inesperada.
- Las excepciones pueden ser capturadas.
- Las excepciones no capturadas causan que el programa se detenga y se muestra el mensaje de error.

[Goodrich et al., 2013, pág. 33]



Tipos comunes de excepciones

<i>Exception</i>	Clase base.
<i>AttributeError</i>	El objeto no tiene ese miembro.
<i>EOFError</i>	Fin de fichero.
<i>IOError</i>	Entrada/salida.
<i>IndexError</i>	En secuencias, índice fuera de los límites.
<i>KeyError</i>	Clave inexistente en conjuntos o diccionarios.
<i>KeyboardInterrupt</i>	Control-C.
<i>NameError</i>	Identificador inexistente.
<i>StopIteration</i>	Sin elemento siguiente en iterador.
<i>TypeError</i>	Número o tipo de argumentos erróneo.
<i>ValueError</i>	Valor inválido de parámetro.
<i>ZeroDivisionError</i>	División por 0.

[Goodrich et al., 2013, pág. 33]



Lanzando excepciones

- Palabra reservada **raise**.
- En funciones, es habitual comprobar primero el tipo y luego el valor de los argumentos.

```
1 def sqrt(x):  
2     if not isinstance(x, (int, float)):  
3         raise TypeError( 'x_must_be_numeric' )  
4     elif x < 0:  
5         raise ValueError( 'x_cannot_be_negative' )  
6     # ...
```

[Goodrich et al., 2013, pág. 34]



Excepciones: ejemplo

```
1 def sum(values):
2     if not isinstance(values, collections.Iterable):
3         raise TypeError('parameter_must_be_an_iterable_type')
4     total = 0
5     for v in values:
6         if not isinstance(v, (int, float)):
7             raise TypeError('elements_must_be_numeric')
8         total = total + v
9     return total
```

[Goodrich et al., 2013, pág. 35]

```
1 def sum(values):
2     total = 0
3     for v in values:
4         total = total + v
5     return total
```

- También se lanzarían excepciones, si *values* no es iterable al hacer el **for**, si no es numérico al hacer la suma con un número.

Captura de excepciones

- “Mirar antes de saltar”

```
1 if y != 0:
2     ratio = x / y
3 else:
4     # ...
```

- “Es más fácil pedir perdón que permiso”

```
1 try:
2     ratio = x / y
3 except ZeroDivisionError:
4     # ...
```

- Puede haber varios **except** para un **try**.
- Usar **try except** si el caso excepcional es poco probable o si la comprobación previa sería demasiado costosa.

[Goodrich et al., 2013, pág. 36]



Excepciones en la entrada

```
1 age = -1                                # an initially invalid choice
2 while age <= 0:
3     try:
4         age = int(input('Enter your age in years: '))
5         if age <= 0:
6             print('Your age must be positive')
7     except (ValueError, EOFError):
8         print('Invalid response')
```

```
7     except ValueError:
8         print('That is an invalid age specification ')
9     except EOFError:
10        print('There was an unexpected error reading input.')
11        raise                        # let's re-raise this exception
```



Final de excepciones

Clausula **except** final

- Sin identificar tipos de error: **except**:
- Capturar otras excepciones que pudieran ocurrir.

Cláusula **finally**

- Cuerpo que se ejecuta siempre, casos normales y excepcionales.
- También con no capturadas y relanzadas
- Ejemplo: cerrar ficheros abiertos.

[Goodrich et al., 2013, pág. 38]



Iteradores (I)

- **for** *elemento* **in** *iterable* :
- Muchos tipos son iterables: **list** , **tuple**, **set**.
- Las cadenas producen una iteración sobre sus caracteres, los diccionarios sobre sus claves, los ficheros sobre sus líneas.
- Un **iterador** es un objeto que gestiona una iteración sobre un conjunto de valores.
 - Para *i*, **next**(*i*) produce el siguiente elemento.
 - Excepción *StopIteration* si no hay más elementos.
- Un **iterable** es un objeto (*obj*) que produce un iterador mediante **iter**(*obj*)

[Goodrich et al., 2013, pág. 39]



Iteradores (II)

- Una instancia de **list** es iterable, pero no un iterador

```
1 >>> data = [1, 2, 3, 4]
2 >>> i = iter(data)
3 >>> next(i); next(i)
4 1
5 2
```

- Los bucles **for** automatizan este proceso.
- Se pueden crear múltiples iteradores a partir un único iterable.
- Los iteradores no mantienen una copia de los elementos, los cambios en el iterable se ven reflejados en el iterador.

[Goodrich et al., 2013, pág. 39]



Iteradores (III)

- Funciones y clases que producen una serie de valores iterable implícitamente, sin almacenar explícitamente todos sus valores.
- **range**(1000000) no devuelve una lista de números (sí en Python 2) sino un objeto de la clase **range** que es iterable.
- *Evaluación perezosa.*
- **for j in range**(1000000): no necesita memoria para un millón de valores.
- Si se sale antes del bucle (**break**), no se pierde el tiempo generando valores que no se van a usar.
- **list** (**range**(1000)), **tuple**(**range**(1000))

[Goodrich et al., 2013, págs. 39-40]



Generadores (I)

- Generadores: método para crear iteradores en Python.
- Como funciones, pero con **yield** en vez de **return**.

```
1 def factors(n):    # traditional function that computes factors
2     results = []    # store factors in a new list
3     for k in range(1,n+1):
4         if n % k == 0:    # divides evenly, thus k is a factor
5             results.append(k)    # add k to the list of factors
6     return results    # return the entire list
```

[Goodrich et al., 2013, pág. 40]



Generadores (II)

```
1 def factors(n):           # generator that computes factors
2     for k in range(1,n+1):
3         if n % k == 0:     # divides evenly, thus k is a factor
4             yield k        # yield this factor as next result
```

● for factor in factors(100):

[Goodrich et al., 2013, pág. 40]



Generadores (III)

- Una función no puede tener **yield** y **return**.
 - Salvo un **return** sin argumento para que el generador termine.
- Se ejecuta la función hasta encontrar un **yield**.
 - Se interrumpe temporalmente su ejecución, hasta que se pide otro valor.
- Al alcanzar el final o un **return** se lanza la excepción *StopIteration*.
- Puede haber varios **yield**.

[Goodrich et al., 2013, págs. 40-41]



Generadores (IV)

```
1 def factors(n):      # generator that computes factors
2     k = 1
3     while k * k < n: # while k < sqrt(n)
4         if n % k == 0:
5             yield k
6             yield n // k
7         k += 1
8     if k * k == n:    # special case if n is perfect square
9         yield k
```

[Goodrich et al., 2013, pág. 41]



Generadores (V)

```
1 def fibonacci():
2     a = 0
3     b = 1
4     while True:
5         yield a
6         future = a + b
7         a = b
8         b = future
```

keep going...

report value, a, during this pass

this will be next value reported

and subsequently this

[Goodrich et al., 2013, pág. 41]



Referencias (I)

[Goodrich et al., 2013] El capítulo 1 contiene una introducción a Python.



Goodrich, M. T., Tamassia, R., and Goldwasser, M. H. (2013).

Data Structures and Algorithms in Python.

Wiley.

<http://bcs.wiley.com/he-bcs/Books?action=index&bcsId=8029&itemId=1118290275>.

