Python (II)

Algoritmia
Grado en Ingeniería Informática
Universidad de Burgos

Juan José Rodríguez Diez

Contenido

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: $n = \textbf{int}(\textbf{input}(\text{"n:"}))$

```
respuesta = input('Introduzca x e y, separados por espacios:')
valores = respuesta.split()
x = float(valores[0])
y = float(valores[1])
# alternativamente:
(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

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

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

[Goodrich et al., 2013, págs. 37-38]

```
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 in iterable, pero no un iterador

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

Si se cambia el iterable inicial, esos cambios se ven reflejados en el iterador.

- 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:                  # keep going...
5         yield a                  # report value, a, during this pass
6         future = a + b
7         a = b                    # this will be next value reported
8         b = future               # 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.