Python

Algoritmia
Grado en Ingeniería Informática
Universidad de Burgos

Juan José Rodríguez Diez

Contenido

Introducción

- Desarrollado por Guido van Rossum, principios de los 90.
- Python 2 en el 2000.
- Python 3 en el 2008.
- Diferencias significativas entre Python 2 y Python 3.
- Hay bibliotecas que solo funcionan en Python 2.
- Usaremos Python 3.
- `www.python.org`

[Goodrich et al., 2013, pág. 2]

Intérprete

- Lenguaje interpretado.
- Al ejecutar `python` sin argumentos se lanza el intérprete.
- El intérprete recibe un comando, lo ejecuta y muestra el resultado.

```
1 >>>2*3
2 6
```

- Código fuente (*scripts*) en ficheros con extensión `.py`.
- Al ejecutar `python` usando como argumento el nombre de un fichero se ejecuta el script.
- Distintos entornos de desarrollo.

[Goodrich et al., 2013, pág. 2]

Un programa Python (I)

```
1  print('Welcome to the GPA calculator.')
2  print('Please enter all your letter grades, one per line.')
3  print('Enter a blank line to designate the end.')
4  # map from letter grade to point value
5  points = {'A+':4.0, 'A':4.0, 'A-':3.67,
6            'B+':3.33, 'B':3.0, 'B-':2.67,
7            'C+':2.33, 'C':2.0, 'C-':1.67,
8            'D+':1.33, 'D':1.0, 'F':0.0}
9  num_courses = 0
10 total_points = 0
11 done = False
```

[Goodrich et al., 2013, pág. 3]

Un programa Python (II)

```
12 while not done:
13     grade = input()                  # read line from user
14     if grade == '':                  # empty line was entered
15         done = True
16     elif grade not in points:        # unrecognized grade entered
17         print("Unknown grade '{0}' being ignored".format(grade))
18     else:
19         num_courses += 1
20         total_points += points[grade]
21 if num_courses > 0:                  # avoid division by zero
22     print('Your GPA is {0:.3}'.format(total_points /
23                                       num_courses))
```

[Goodrich et al., 2013, pág. 3]

Un programa Python (III)

- GPA: *grade point average*.
- Las sentencias individuales normalmente terminan con un salto de línea.
- Para continuar en la siguiente, acabar con contrabarra ("\").

```
1 >>>2+3\
2       *4
3 14
```

- También se continúa si hay algún delimitador izquierdo (por ejemplo, "{") y en la línea actual no está el correspondiente delimitador derecho.
  - En el programa, en la asignación a *points*.

[Goodrich et al., 2013, pág. 3]

Un programa Python (IV)

- Varias sentencias en la misma línea: separar con “;”

```
1 i = 0; j = 1
```

- Espacios para delimitar los cuerpos de las estructuras de control.
- Un bloque de código se indenta para designar que es el cuerpo de una estructura de control.
- Estructuras de control anidadas: varias indentaciones.
- Comentarios: empiezan con “#”, hasta el fin de línea.

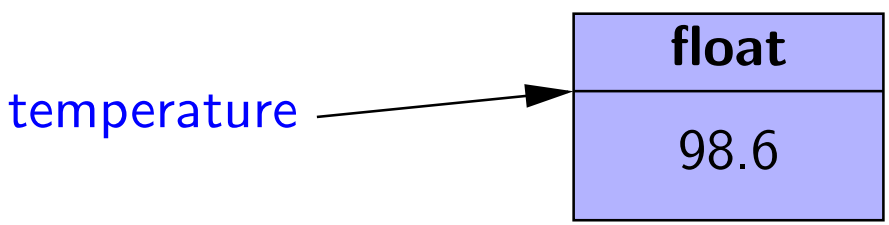
[Goodrich et al., 2013, pág. 3]

Identificadores, objetos y asignaciones

- Es un lenguaje orientado a objetos.
- Los tipos de datos están basados en clases.
- Sentencias de asignación.

```
1 temperature = 98.6
```

- Se establece *temperature* como identificador.
- Se asocia al objeto expresado parte derecha de la asignación.
- En este caso, un objeto del tipo **float**.



[Goodrich et al., 2013, pág. 4]

Identificadores

- Sensible a mayúsculas.
- Combinaciones de letras, números y subrayados.
- No pueden empezar por un número.
- Palabras reservadas que no pueden ser identificadores.

and	continue	except	global	lambda	pass	while
as	def	False	if	None	raise	with
assert	del	finally	import	nonlocal	return	yield
break	elif	for	in	not	True	
class	else	from	is	or	try	

[Goodrich et al., 2013, pág. 4]

Objetos (I)

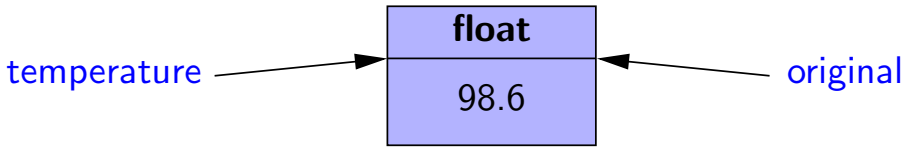
- Los identificadores en Python son similares a las referencias en Java o a los punteros en C.
- Cada identificador tiene asociada la dirección de memoria del objeto al que se refiere.
- A un identificador se puede asociar un objeto especial, *None*.
- Tipado dinámico, no hay una declaración previa indicando el tipo asociado al identificador.
- El objeto asociado a un identificador sí que tiene un tipo concreto.

[Goodrich et al., 2013, pág. 5]

Objetos (II)

- Se puede establecer un *alias*, asignando un segundo identificador a un objeto existente.

```
1  original = temperature
```



- Los cambios realizados en un objeto con un alias, se reflejan en los otros alias.
- Al hacer una reasignación, el objeto no se modifica.

```
1  temperature = temperature + 5.0
```

[Goodrich et al., 2013, pág. 5]

Instanciación

- Crear una nueva instancia de una clase.
- Invocar al constructor.

```
1 >>>a = int()
2 >>>a
3 0
4 >>>a = int(17)
5 >>>a
6 17
```

- En tipos predefinidos, literales para designar a nuevas instancias.
- También se pueden obtener nuevos objetos llamando a funciones.

[Goodrich et al., 2013, pág. 6]

Llamadas a métodos

- Hay funciones tradicionales: **sorted**(*data*), **len**("hola").
- Pero también hay métodos (o funciones miembro) en las clases: *data*.*sort*(), "hola".*upper*().

```
>>> l = [3, 1, 2]
>>> sorted(l)
[1, 2, 3]
>>> l
[3, 1, 2]
>>> l.sort()
>>> l
[1, 2, 3]
```

```
>>> len("hola")
4
>>> "hola".upper()
'HOLA'
```

[Goodrich et al., 2013, pág. 6]

Clases predefinidas

- Clases *inmutables*: los objetos una vez creados no se pueden cambiar (los identificadores pueden reasignarse.)

Clase	Descripción	Inmutable
bool	valor booleano	√
int	entero	√
float	número en punto flotante	√
list	secuencia mutable de objetos	
tuple	secuencia inmutable	√
str	cadena	√
set	conjunto de objetos distintos, sin orden	
frozenset	conjunto inmutable	√
dict	diccionario, "mapa" asociativo	

[Goodrich et al., 2013, pág. 7]

bool

- Solo dos instancias, literales *True* y *False*.
- Constructor por defecto: **bool**(), devuelve *False*.
- Obtener un booleano a partir de no booleanos: **bool**(x).
- Numéricos: *False* si 0, si no *True*.
- Cadenas, secuencias, contenedores: *False* si vacías.
- Utilidad: usar no booleanos en estructuras de control.
  - "**if** x:", "**while** x:"

[Goodrich et al., 2013, pág. 7]

int

- Enteros de magnitud arbitraria.
- Elige la representación interna basada en la magnitud.
- Binario, octal, hexadecimal: 0*b1011*, 0*o52*, 0*x7f*.
- **int**() devuelve 0.
- **int**(*x*) para obtener un entero a partir de algunos tipos.
  - **int**(*True*).
  - Dado un número real trunca: **int**(3.14).
  - Dada una cadena, intenta convertirla en entero: **int**('137').
  - Argumento adicional indicando la base: **int**('7f', 16).

[Goodrich et al., 2013, pág. 8]

float

- Único tipo para números reales.
- Precisión fija, similar a un *double* en otros lenguajes.
- `2.` es lo mismo que `2.0`.
- Notación científica: `6.022e23`
- Constructor **float**() devuelve `0.0`.
- **float**(*x*) intenta convertir a float.
  - **float**(2), **float**('3.14').

[Goodrich et al., 2013, pág. 8]

Secuencias

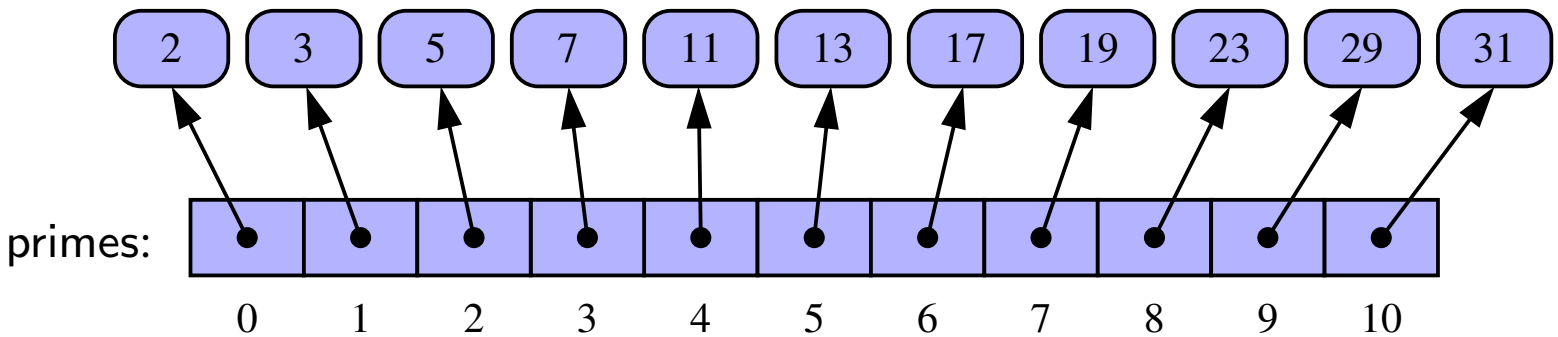
- **list** , **tuple**, **str**.
- Colecciones de valores en las que el orden importa.
- **list** es la más general, secuencia de objetos arbitrarios.
- **tuple** es una versión inmutable de **list** .
- **str**: secuencia inmutable de caracteres.
- No hay un tipo carácter, hay cadenas de longitud 1: `'a'`.

[Goodrich et al., 2013, pág. 9]

list (I)

- Secuencia arbitraria de objetos: [7, *None*, 3.14, 'hola'].
- Lista vacía: [].
- Estructura *referencial*, almacena una secuencia de referencias a sus elementos.

```
1 primes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]
```



[Goodrich et al., 2013, pág. 9]

list (II)

- Elementos indexados, empezando en 0.

```
>>> l = ['a', 'b', 'c']
>>> l[1]
'b'
>>> ['a', 'b', 'c'][1]
'b'
```

- Su longitud puede cambiar.

[Goodrich et al., 2013, pág. 9]

- **list** () devuelve la lista vacía.
- **list** (x) intenta obtener una lista a partir de x.
  - A partir de iterables.

```
>>> list('hola')
['h', 'o', 'l', 'a']
```

  - Si x es una lista, **list** (x) crea una copia.

tuple

- Secuencias inmutables.
- Podría tener una representación más compacta que una **list** .
- Delimitadas por paréntesis.
- (7, *None*, 3.14, 'hola')
- () es la tupla vacía.
- Para tuplas de un elemento, poner una coma después del elemento: (17,).

[Goodrich et al., 2013, pág. 10]

str

- Representación eficiente de una secuencia inmutable de caracteres. En unicode.
- Comillas simples o dobles: 'hola', "hola".
- Se puede incluir tal cual en la cadena el otro tipo de comillas: "Don't worry".
- Escape: 'Don\'t worry', 'C:\\Python\\', '\n', '\t'.
- Incluir caracteres unicode: '20\u20AC', 20€.
- Delimitar con tres comillas (simples o dobles), permite incluir saltos de línea tal cual.

```
1 print( ''' Welcome to the GPA calculator.
2 Please enter all your letter grades, one per line.
3 Enter a blank line to designate the end. ''' )
```

[Goodrich et al., 2013, pág. 10]

set y frozenset (I)

- Conjunto: colección de elementos, sin duplicados, sin un orden inherente.
- Ventaja principal frente a listas: eficiencia en determinar si un elemento esa en el conjunto.
- Basados en tablas hash.
- No mantiene los elementos en un orden determinado.
- Solo se pueden añadir a un conjunto elementos de tipos inmutables.
- Podemos tener un conjunto de tuplas, pero no de listas.
- Podemos tener un **set** con **frozenset**s.

[Goodrich et al., 2013, pág. 11]

set y frozenset (II)

- Delimitados por llaves.
- {17}, {'red', 'green', 'blue'}.
- {} no es el conjunto vacío, es un diccionario vacío.
- **set**() devuelve el conjunto vacío.
- **set**(*x*) crea un conjunto con los elementos de *x* si es un iterable.

```
1 >>>set('hola')
2 {'a', 'h', 'l', 'o'}
```

- **frozenset**(), **frozenset**(*x*).
- {17, **frozenset**('hola'), *None*, (3, 14)}
  - No sería válido con **set**('hola') ó [3,14]

[Goodrich et al., 2013, pág. 11]

dict

- Diccionario, aplicación, "mapa".
- De un conjunto de *claves* distintas a sus *valores* asociados.
- Implementación similar a *set*, pero almacenando los valores asociados.
- También se delimitan con llaves.
- Diccionario vacío: $\{\}$.
- Pares clave:valor, separados por comas.
- $\{'uno':1,\ 'dos':2\}$.
- El constructor también acepta una secuencia de pares clave valor.
- $pares = [('uno',\ 1),\ ('dos',\ 2)];\ \mathbf{dict}(pares)$

[Goodrich et al., 2013, pág. 11]

Expresiones y operadores

- Crear objetos de las clases predefinidas con literales y constructores.
- Usar nombres para identificar objetos.
- Combinar valores existentes mediante *operadores* para obtener *expresiones*.
- La semántica de un operador depende del tipo de los operandos.
  - $a + b$: suma de números, concatenación de secuencias.
- Expresiones compuestas, con varios operadores. $a + b * c$
  - Precedencia de operadores, paréntesis.

[Goodrich et al., 2013, pág. 12]

Operadores lógicos, de igualdad

- **and**, **or**, **not**.
- Evaluación en "cortocircuito", no se evalúa el segundo operando si se sabe el resultado con la evaluación del primero.

- **is**, **is not**.
  - Alias del mismo objeto.
- ==, !=.
  - Equivalentes, pueden ser objetos distintos.

```
1 >>>[1, 2] is [1, 2]
2 False
3 >>>[1, 2] == [1, 2]
4 True
```

[Goodrich et al., 2013, págs. 12-13]

De comparación, aritméticos

- $<$, $<=$, $>$, $>=$.
- Lexicográfico para secuencias.
- Sensible a mayúsculas en cadenas.
- Excepción si tipos no compatibles.

- $+$, $-$, $*$, $/$, $//$, $\%$, $**$.
  - División real y entera.

[Goodrich et al., 2013, págs. 13-14]

```
1  >>>27 / 4
2  6.75
3  >>>27 // 4
4  6
5  >>>27 % 4
6  3
7  >>>8.2 // 3.14
8  2.0
9  >>>8.2 % 3.14
10 1.9199999999999995
```

Sobre bits

~	complemento
&	and
|	or
^	xor
<<	desplazamiento a la izquierda, entran 0s
>>	desplazamiento a la derecha, entra el bit de signo

```
>>> ~27
-28
>>> 16 << 2
64
>>> -64 >> 2
-16
```

[Goodrich et al., 2013, pág. 14]

Sobre secuencias (I)

$s[j]$	elemento en j (empezando en 0)
$s[\ inicio : fin\ ]$	trozo de $inicio$ (incluido) a fin (sin incluir)
$s[\ inicio : fin : paso]$	$s[\ inicio\ ] + s[\ inicio + paso] + s[\ inicio + 2 * paso] \ldots$
$s + t$	concatenación
$k * s$	$s + s + s + \ldots$ (k veces)
val **in** s	comprobación de pertenencia
val **not in** s	no pertenencia

[Goodrich et al., 2013, pág. 14]

Sobre secuencias (II)

- *inicio* y *fin* son opcionales.
  - $s\,[2{:}]$, $s\,[{:}10]$
- Admite índices negativos.
  - $s[-1]$ es el último elemento.
- Admite pasos negativos.
  - $s\,[::-1]$ es la secuencia inversa.
- En listas, al ser mutables, podemos hacer: $s[\,i\,] = v$, **del** $s[\,i\,]$.
  - No se puede en **tuple**, **str**.
- Para cadenas, **in** comprueba si una cadena es subcadena de otra.
- Las secuencias admiten operadores de comparación: ==, !=, <...

[Goodrich et al., 2013, págs. 14-15]

Sobre conjuntos

k **in** *s*	pertenencia
k **not in** *s*	no pertenencia
s1 == *s2*	equivalencia
s1 != *s2*	no equivalencia
s1 <= *s2*	*s1* es subconjunto de *s2*
s1 < *s2*	*s1* es subconjunto propio de *s2*
s1 >= *s2*	*s1* es superconjunto *s2*
s1 > *s2*	*s1* es superconjunto propio *s2*
s1 | *s2*	unión
s1 & *s2*	intersección
s1 − *s2*	elementos en *s1* pero no en *s2*
s1 ^ *s2*	elementos que están en solo uno de los dos

[Goodrich et al., 2013, pág. 15]

Sobre diccionarios

- Equivalencia: == y !=.
  - Mismos pares clave/valor.
- Sin operaciones de sub/superconjunto (<, <=...).

$d[k]$	valor asociado a la clave k
$d[k] = v$	asignar un valor a la clave
del $d[k]$	eliminar la clave
k **in** d	pertenencia
k **not in** d	no pertenencia
$d1 == d2$	equivalencia
$d1$!= $d2$	no equivalencia

[Goodrich et al., 2013, pág. 16]

Asignaciones extendidas

- Para la mayoría de los operadores binarios.
- *contador* += 5.
- En tipos inmutables, no se cambia el objeto existente, se reasigna el identificador.
- Es posible que un tipo redefina la semántica para mutar el objetos (+= para **list**).

```
1 alpha = [1, 2, 3]
2 beta = alpha              # alias para alpha
3 beta += [4, 5]            # extiende la lista original
4 beta = beta + [6, 7]      # reasigna beta a una lista nueva
5 print(alpha)              # será [1, 2, 3, 4, 5]
6 print(beta)               # será [1, 2, 3, 4, 5, 6, 7]
```

[Goodrich et al., 2013, pág. 16]

Expresiones compuestas y precedencia

Operator Precedence		
	Type	**Symbols**
1	member access	expr.member
2	function/method calls container subscripts/slices	expr(...) expr[...]
3	exponentiation	**
4	unary operators	+expr, −expr, ~expr
5	multiplication, division	*, /, //, %
6	addition, subtraction	+, −
7	bitwise shifting	<<, >>
8	bitwise-and	&
9	bitwise-xor	^
10	bitwise-or	|
11	comparisons containment	**is**, **is not**, ==, !=, <, <=, >, >= **in**, **not in**
12	logical-not	**not** expr
13	logical-and	**and**
14	logical-or	**or**
15	conditional	val1 **if** cond **else** val2
16	assignments	=, +=, −=, *=, etc.

Asignación encadenada:
$x = y = 0$.

Comparaciones encadenadas:
$1 <= x + y <= 10$

[Goodrich et al., 2013, pág. 17]

Control de flujo

- En las distintas estructuras, se usa dos puntos (:) para delimitar el inicio de un bloque que es el cuerpo de la estructura de control.
- Si el cuerpo es una línea puede estar en la misma línea.
- Un cuerpo normalmente será un bloque indentado.
- La indentación determina la extensión del cuerpo.

[Goodrich et al., 2013, pág. 18]

Condicionales (I)

```
1 if primera_condición:
2     primer_cuerpo
3 elif segunda_condición:
4     segundo_cuerpo
5 elif tercera_condición:
6     tercer_cuerpo
7 else:
8     cuarto_cuerpo
```

[Goodrich et al., 2013, pág. 18]

- Los cuerpos contienen uno o más comandos.
- Número cualquiera de **elif**.
- **else** opcional.
- Las condiciones son expresiones booleanas.
- Los no booleanos se intentan convertir a booleanos.

```
1 if respuesta:
```

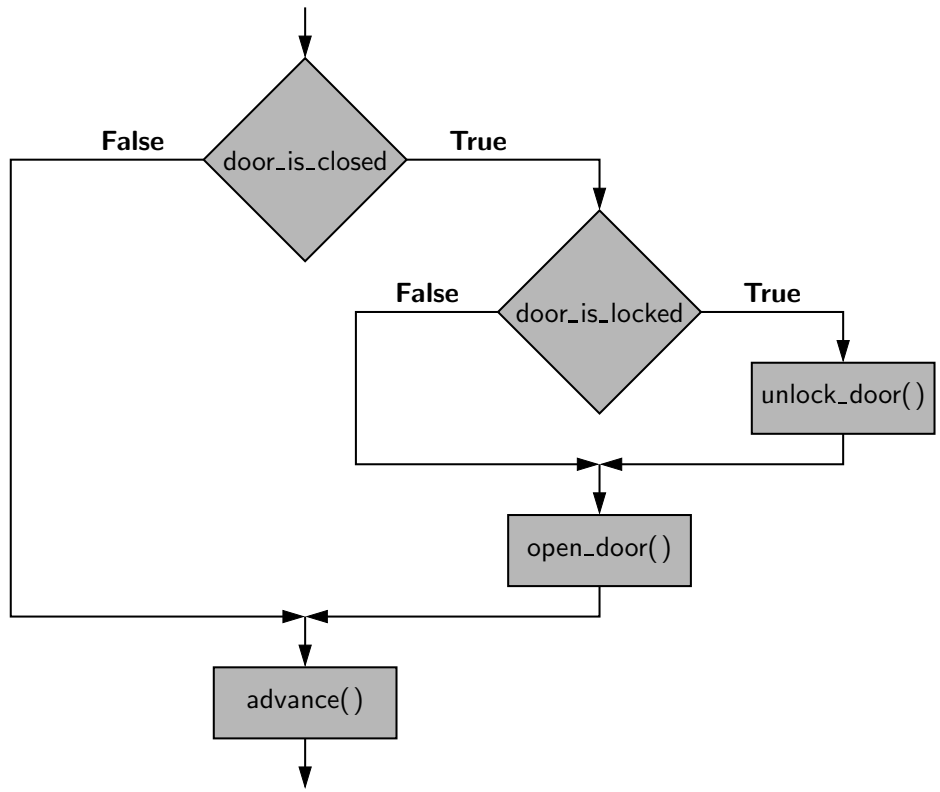
Si *respuesta* es una cadena, es lo mismo que

```
1 if respuesta != '':
```

Condicionales (II)

```
1 if door_is_closed :
2     open_door( )
3 advance( )
```

```
1 if door_is_closed :
2     if door_is_locked :
3         unlock_door( )
4     open_door( )
5 advance( )
```



[Goodrich et al., 2013, pág. 19]

Bucles **while**

- Bucle general basado en la comprobación repetida de una condición booleana.

```
while condicion:
    cuerpo
```

```
j=0
while j < len(data) and data[j] != X:
    j += 1
```

[Goodrich et al., 2013, pág. 20]

Bucles for

- Para iterar sobre los valores de una serie definida.
  - Caracteres de una cadena, elementos de una lista, números en un rango.

```
for elemento in iterable :
    cuerpo
```

[Goodrich et al., 2013, pág. 21]

Suma de elementos

```
total = 0
for val in data:
    total += val
```

Máximo de una lista

```
biggest = data[0]
for val in data:
    if val > biggest:
        biggest = val
```

Bucles **for** con índices

- Clase **range** que genera secuencias de enteros.
- **range**(n) genera n valores, de 0 a $n-1$.
- Para recorrer los índices de una secuencia de elementos:
  for *j* **in** **range**(**len**(*data*)):

Índice del máximo

```
1 big_index = 0
2 for j in range(len(data)):
3     if data[j] > data[big_index]:
4         big_index = j
```

[Goodrich et al., 2013, pág. 22]

break y continue

- **break** termina un bucle (**while** o **for**).
- En estructuras anidadas, termina el bucle más cercano que lo contiene.
- **continue** termina la iteración actual, no se ejecuta el resto del cuerpo (en esa iteración).

```
1 found = False
2 for item in data:
3     if item == target:
4         found = True
5         break
```

[Goodrich et al., 2013, pág. 22]

Funciones

- Funciones tradicionales (no métodos).
- Signatura: nombre, parámetros.
- Sin tipos de los parámetros y del resultado.
- Cuerpo de la función indentado.
- Ámbito local de los identificadores.

```
1 def count(data, target ):
2     n = 0
3     for item in data:
4         if item == target:                    # found a match
5             n += 1
6     return n
```

[Goodrich et al., 2013, pág. 23]

return

- Sin argumentos, devuelve *None*.
- También se devuelve *None* si se alcanza el final del cuerpo.
- A menudo habrá un **return** al final, pero puede haber más.

```
1 def contains(data, target):
2     for item in data:
3         if item == target:                    # found a match
4             return True
5     return False
```

[Goodrich et al., 2013, pág. 24]

Paso de información

- Parámetros formales (identificadores) y actuales (objetos enviados).
- El paso de parámetros se comporta como la asignación.
- También la comunicación del valor de retorno.
- No hay copia de objetos.
- Si el parámetro es un objeto de un tipo mutable, se puede cambiar el objeto.

```
1 def scale(data, factor):
2     for j in range(len(data)):
3         data[j] *= factor
```

[Goodrich et al., 2013, págs. 24-25]

Parámetros con valores por defecto (I)

- Las funciones pueden tener valores por defecto para algunos parámetros.
- Llamar a una función con un número variable de parámetros.
- **def** $f(a,\ b{=}15,\ c{=}27)$ se puede llamar con 1, 2 ó 3 parámetros.
- Si un parámetro tiene un valor por defecto, todos los siguientes también tienen que tenerlo.
- También se puede dar valores a los argumentos por nombre, en vez de posición: $f(10,\ c{=}2)$

[Goodrich et al., 2013, pág. 26-27]

Parámetros con valores por defecto (II)

```
1  def compute_gpa(grades, points={'A+':4.0, 'A':4.0,  'A-':3.67, 'B+':3.33,
2                                  'B':3.0,  'B-':2.67,'C+':2.33, 'C':2.0,
3                                  'C':1.67,  'D+':1.33, 'D':1.0,  'F':0.0}):
4      num_courses = 0
5       total_points  = 0
6      for g in grades:
7          if g in points:                    # a recognizable grade
8              num_courses += 1
9              total_points  += points[g]
10     return  total_points / num_courses
```

- Posible llamada: *compute_gpa*(['A-', 'C'])

[Goodrich et al., 2013, pág. 26]

Parámetros con valores por defecto (III)

- **range** se puede llamar de tres maneras: **range**(*n*), **range**(*start*, *stop*), **range**(*start*, *stop*, *step*).
- Parece que *n* se correspondería con *stop*.

```
1 def range(start, stop=None, step=1):
2     if stop is None:
3         stop = start
4         start = 0
5     ...
```

[Goodrich et al., 2013, pág. 27]

Algunas funciones predefinidas

- Entrada/salida: **print**, **input**, **open**...
- Codificación de caracteres: **ord**('A'), **chr**(65).
- Matemáticas: **abs**, **divmod**, **pow**, **round**, **sum**...
- Orden: **max**, **min**, **sorted**
- Colecciones/iteraciones: **range**, **len**, **reversed**, **all**, **any**, **map**...

[Goodrich et al., 2013, págs. 28-29]

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.