Slides

January 26, 2017

Contents

Chapter 1

Python

1.1 Módulo 1 - Introducción

1.1.1 £Por qué?

- Gratuito, portable, potente y sencillo de utilizar.
- Quinto lenguaje más utilizado según el TIOBE index
- Como comparación, R ocupa la posición 19
- Más de 600,000 preguntas con la etiqueta Python en StackOverflow
- Más de 80,000 librerías en el Python Package Index

1.1.2 £Quién?

Guido van Rossum

Benevolent Dictator For Life

Over six years ago, in December 1989, I was looking for a "hobby" programming project that would keep me occupied during the week around Christmas. My office . . . would be closed, but I had a home computer, and not much else on my hands. I decided to write an interpreter for the new scripting language I had been thinking about lately: a descendant of ABC that would appeal to Unix/C hackers. I chose Python as a working title for the project, being in a slightly irreverent mood (and a big fan of Monty Python's Flying Circus)

1.1.3 £Qué?

- Popular lenguaje de programación Open Source.
- Aplicación en aplicaciones completas y scripting.
- Enfocado a la productividad, y a la calidad y claridad de código.

- Lenguaje de alto nivel Sencillo de programar, funciona sin cambios en distintos sistemas.
- Interpretado, no compilado Permite uso interactivo al coste de velocidad de ejecución.
- Gestión automática de memoria
- Multi-paradigma Mezcla programación imperativa, funcional y orientada a objetos.
- Tipado dinámico Los objetos tienen tipo, las variables no
- Extensa librería estándar
- Indentación semántica en lugar de llaves

1.1.4 Zen de Python

There should be one (and preferably only one) obvious way to do it.

```
In [5]: import this

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

1.1.5 2.7 vs 3.6

- Versiones incompatibles Partes de 3.x disponibles en 2.x con el módulo **future**.
- Versión 2.7 incluida por defecto en OS X y la mayoria de distros linux.
- La versión 3.x continua actualizandose, la 2.x solo recibe bugfixes.

- Todavia existen librerías incompatibles con Python 3
- Python 3 incorpora una serie de cambios que impiden retrocompatibilidad Mejora soporte unicode, corrige inconsistencias del lenguaje…

1.1.6 Instalación de paquetes

Usando pip

pip install numpy

Usando anaconda

conda install numpy

- Múltiples instalaciones intercambiables utilizando *entornos virtuales*
- Pip y Anaconda pueden usarse al mismo tiempo, pero no interoperan
- Anaconda permite instalar dependencias del sistema

1.1.7 Formato y ejecución

- Directamente desde línea de comandos
- Mediante un REPL (comandos python / ipython)
- Ejecutando un script (ficheros .py)
- Dentro de un notebook

Línea de comandos

```
[rober@localhost ~]$ python -c "print 'hola mundo'"
hola mundo
[rober@localhost ~]$
```

REPL de Python

```
[rober@localhost ~]$ python
Python 2.7.12 (default, Sep 29 2016, 13:30:34)
[GCC 6.2.1 20160916 (Red Hat 6.2.1-2)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print 'hola mundo'
hola mundo
>>>
```

REPL de iPython / Jupyter
Desde un fichero .py
Dentro de un notebook

1.1.8 Cabeceras

Shebang - permite ejecutar el script implicitamente con el interprete seleccionado.
!/usr/bin/env python
Codificación - define la codificación de caracteres del fichero como UTF-8
Permite incluir caracteres como ñ o ó en el fuente.

```
In [4]: # -*- coding: utf-8 -*-
```

```
[rober@localhost ~]$ ipython
Python 2.7.12 (default, Sep 29 2016, 13:30:34)
Type "copyright", "credits" or "license" for more information.

IPython 5.1.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: print 'hola mundo'
hola mundo

In [2]:
```

```
><>~¤ /home/rober 80x8
  1 print "hola mundo"
~
~
~
~
~
NORMAL  test.py                                  python  100% ␤  1:  1

rober@localhost:~ 80x41
[rober@localhost ~]$ python test.py
hola mundo
[rober@localhost ~]$
```

```
In [9]:                                                    Slide Type
print "hola mundo"
hola mundo
```

1.2 Módulo 2A - Conceptos Básicos

1.2.1 Tipos básicos y valores

Tipo	Valores	
int	-2, -1, 0, 1, 2	Números enteros
float	3.1415, 1.4142, 1e10	Números decimales
str	'hola', "dos", """Python"""	Cadenas de texto
bool	True, False	Valores lógicos
NoneType	None	

Float

```
In [102]: print type(123.)
          print type(.523)
          print type(24.412)
          print type(12e16)

<type 'float'>
<type 'float'>
<type 'float'>
<type 'float'>
```

String

```
In [105]: print type('cadena " de \' caracteres')
          print type("cadena \" de ' caracteres")
          print type("""
          cadena "
              de '
          caracteres "
          """)

<type 'str'>
<type 'str'>
<type 'str'>
```

1.2.2 Comprobación de tipos

```
In [36]: print -1, type(-1)
         print 3.1415, type(3.1415)
         print "hola", type("hola")
         print True, type(True)
         print None, type(None)

-1 <type 'int'>
3.1415 <type 'float'>
hola <type 'str'>
True <type 'bool'>
None <type 'NoneType'>
```

```
In [37]: print type(1) is int
         print type(1.4142) is float
         print type("hola") is str
         print type(True) is bool
         print type(None) is NoneType
```

```
True
True
True
True
```

```
        ---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call last)

        <ipython-input-37-e41353170ceb> in <module>()
          3 print type("hola") is str
          4 print type(True) is bool
    ----> 5 print type(None) is NoneType


        NameError: name 'NoneType' is not defined
```

1.2.3 Conversión entre tipos

int

```
In [40]: print int(2.1)
         print int(True), int(False)
         print int(" 2 ")
         print int("dos")
```

```
2
1 0
2
```

```
        ---------------------------------------------------------------------------

        ValueError                                Traceback (most recent call last)

        <ipython-input-40-24b766dc7cc2> in <module>()
          2 print int(True), int(False)
          3 print int(" 2 ")
    ----> 4 print int("dos")


        ValueError: invalid literal for int() with base 10: 'dos'
```

```
In [41]: print int(None)
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-41-6842e1ef3c45> in <module>()
----> 1 print int(None)


TypeError: int() argument must be a string or a number, not 'NoneType'
```

float

```
In [21]: print float(2)
         print float(True), float(False)
         print float(" 2.5 ")
         print float("2,5")
```

```
2.0
1.0 0.0
2.5



---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-21-7ab135c75e56> in <module>()
  2 print float(True), float(False)
  3 print float(" 2.5 ")
----> 4 print float("2,5")


ValueError: invalid literal for float(): 2,5
```

```
In [42]: print float(None)
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-42-4f7c66b40a62> in <module>()
----> 1 print float(None)


TypeError: float() argument must be a string or a number
```

str

```
In [43]: print str(2)
         print str(3.1415)
         print str(True), str(False)
         print str(None)
```

```
2
3.1415
True False
None
```

bool

```
In [45]: print bool(5)
         print bool(-2)
         print bool(0)

True
True
False
```

```
In [28]: print bool(2.5)
         print bool(0.0)
         print bool(1e-100)

True
False
True
```

```
In [34]: print bool("Cualquier cadena de texto")
         print bool("")
         print bool(" ")

True
False
True
```

```
In [46]: print bool(None)

False
```

1.2.4 Operaciones aritméticas

Operador	Función
+x -x	signo
+ -	suma y resta
/ // %	multiplicación, división y resto
**	exponente

Precedencia

exponente -> signo -> multiplicación / división / resto -> suma / resta

```
In [47]: ((3 + 2) * 2) - 1

Out[47]: 9
```

```
In [48]: 3 + 2 * 2 - 1

Out[48]: 6
```

La operación suma también funciona con cadenas de caracteres

```
In [139]: frase = "unimos " + "cadenas " + "de " + "caracteres"
          print frase

unimos cadenas de caracteres
```

Precaución al introducir otros tipos de variables sin realizar las conversiones debidas

```
In [141]: print "Número " + 3 + "."


        ---------------------------------------------------------------

        TypeError                       Traceback (most recent call last)

        <ipython-input-141-2b28070ce7d8> in <module>()
    ----> 1 print "Número " + 3 + "."


        TypeError: cannot concatenate 'str' and 'int' objects


In [142]: print "Número " + str(3) + "."

Número 3.
```

Las cadenas también permiten multiplicación

```
In [143]: "s" * 10

Out[143]: 'sssssssss'
```

1.2.5 División

Python 2	Funcionamiento
int / int	entero redondeado hacia abajo
int / float	decimal sin redondeo

```
In [50]: 4/3, 4/3.0

Out[50]: (1, 1.3333333333333333)
```

Python 3 -> siempre decimal sin redondeo

1.2.6 Floor division

Redondeo explicito **hacia abajo**

```
In [53]: 4 // 3, 4.0 // 3.0

Out[53]: (1, 1.0, 1.3333333333333333)
```

1.2.7 Asignación de variables

La asignación de variables se realiza mediante =, con tipado dinámico

Por convención, los nombres de variables se componen de letras en minúscula, separando palabras con _.

```
In [58]: variable_uno = 40
         variable_dos = 2
         variable_uno + variable_dos

Out[58]: 42
```

Se puede asignar múltiples valores al mismo tiempo

```
In [59]: a, b, c = 1, 5.2, 'var'

         print c, b, a

var 5.2 1


In [63]: x, y = 0, 0
         x += 2
         y -= 2
         print x, y

2 -2


In [68]: x, y, z = 3, 3, 3
         x *= 3
         y /= 2.
         z //= 2
         print x, y, z

9 1.5 1


In [73]: x, y = 3, 3
         x %= 2
         y **= 3
         print x, y

1 27
```

1.2.8 Comentarios

Comentarios de una línea con #

```
In [135]: # Esto es un comentario de una línea
          x = 12
          # Esto es otro comentario de una línea
```

Comentarios multilinea entre tres dobles comillas (“””)

```
In [138]: """
          Esto es un comentario multilinea
              Puede contener saltos de línea, ""
           tabuladores, espacios y comillas
          """
          x = 6
```

1.2.9 Pass

La expresión **pass** no tiene ningún efecto.

```
In [144]: pass
```

1.2.10 Operadores lógicos

and, or, not

```
In [75]: print True or False
         print True and False
         print not (True and False)
```

```
True
False
True
```

Preferencia sobre operadores aritméticos

```
In [78]: None or 2 + 5
```

```
Out[78]: 7
```

1.2.11 Comparaciones

Toman dos valores y devuelven un booleano.

Operador	Símbolo
mayor	>
mayor igual	>=
igual	==
no igual	!=
menor igual	<=
menor	<

```
In [110]: print 5 == 5
          print 4 < 1e10
          print 2.0 > -.1
```

```
True
True
True
```

1.2.12 Funciones

```
In [79]: def foo():
             print "foo"

         foo()
```

```
foo
```

```
In [80]: print foo
```

```
<function foo at 0x7f705406f5f0>
```

```
In [81]: otro_foo = foo
         otro_foo()
```

```
foo
```

Retorno de valores

```
In [83]: def foo():
             return 0
         print foo()
```

```
0
```

Una misma función puede tener múltiples valores de retorno.

```
In [84]: def foo():
             return 0, 5
         print foo()
```

```
(0, 5)
```

```
In [85]: a, b = foo()
         print b
```

```
5
```

1.2.13 Parámetros

- Los parámetros carecen de tipo.
- Puden pasarse por nombre.
- Pueden tener asignados valores por defecto (permite parámetros opcionales).

```
In [90]: def foo(a, b, c):
             return (a + b) * c
         print foo(1, 2, 3)
```

```
9
```

```
In [91]: foo(c=0, a=1, b=1)
```

```
Out[91]: 0
```

```
In [94]: def foo(a, b, c=1):
             return (a + b) * c

         foo(2, 3), foo(2, 3, 2)
```

```
Out[94]: (5, 10)
```

```
In [95]: def foo(a=1, b, c):
             return (a + b) * c

         foo(1, 2)
```

```
  File "<ipython-input-95-eda175a7e5a3>", line 1
    def foo(a=1, b, c):
SyntaxError: non-default argument follows default argument
```

1.2.14 Scope

- Bloques definidos mediante espacio en blanco (tabuladores o espacios).
- Por norma general, las variables pueden acceder a valores dentro de su nivel de indentación o mayor.
- Siempre dentro del mismo bloque.

```
In [100]: var_a = 5

          def foo():
              var_b = 2
              print 'a interior = ', var_a
              print 'b interior = ', var_b

          foo()
          print 'a exterior = ', var_a
          print 'b exterior = ', var_b
```

```
a interior =  5
b interior =  2
a exterior =  5
b exterior =
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)

<ipython-input-100-985510e353dd> in <module>()
  8 foo()
  9 print 'a exterior = ', var_a
---> 10 print 'b exterior = ', var_b


NameError: name 'var_b' is not defined
```

1.2.15 Condiciones

```
In [111]: def mayor(a, b):
              if a > b:
                  return a
              else:
                  return b

          mayor(6, 8)
```

```
Out[111]: 8
```

```
In [113]: def mayor(a, b, c):
              if (a >= b and a >= c):
                  return a
              elif (b >= a and b >= c):
                  return b
              else:
                  return c

          mayor(6, 9, 2)
```

```
Out[113]: 9
```

Las condiciones pueden anidarse.

```
In [116]: def mayor(a, b, c):
              if (a >= b):
                  if (a >= c):
                      return a
                  else:
                      return c
              else:
                  if (b >= c):
                      return b
                  else: return c

          mayor(6, 9, 2)
```

```
Out[116]: 9
```

1.2.16 Iteración y bucles

- Python contiene bucles for y while – iteración definida e indefinida.
- **for** se utiliza para iterar sobre secuencias de valores.
- **while** se ejecuta hasta el cumplimiento de una condición.

```
In [118]: range(10)
```

```
Out[118]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
In [120]: for i in range(10):
              print i, i * i
```

```
0 0
1 1
2 4
```

```
3 9
4 16
5 25
6 36
7 49
8 64
9 81
```

```
In [121]: a = 0
          for i in range(10):
              a += i
          print a
```

```
45
```

```
In [122]: a = 2
          while a < 1000:
              a *= a
          print a
```

```
65536
```

Los bucles pueden anidarse unos dentro de otros.

```
In [133]: for i in range(4):
              for j in range(3):
                  print i + j,
              print ''
```

```
0 1 2
1 2 3
2 3 4
3 4 5
```

1.2.17 Break y Continue

- Break interrumpe la iteración
- Continue salta al siguiente ciclo

```
In [124]: a = 0
          for i in range(10):
              a = i
              if i > 3: break

          print a
```

```
4
```

```
In [126]: for i in range(10):
              if i % 2: continue
              print i
```

```
0
2
4
6
8
```

1.3 Módulo 2B - Conceptos Básicos

1.3.1 Import

- Añade uno o más objetos (variables, clases, métodos...) al *namespace** actual.

* Conjunto de objetos a los que podemos referirnos por nombre

- Los objetos añadidos provienen de código residente en otro script del mismo proyecto o en una librería en el *path*.

Importar los elementos de una librería añade el nombre de la librería al namespace, a partir del cual usamos sus contenidos.

Podemos elegir el nombre con el que se importa la librería.

```
In [3]: import math

        print math.pi, math.sin(1), math.cos(1)

3.14159265359 0.841470984808 0.540302305868


In [4]: import math as m

        print m.pi, m.sin(1), m.cos(1)

3.14159265359 0.841470984808 0.540302305868
```

También es posible importar partes concretas de una librería, directamente al namespace del script.

```
In [5]: from math import pi, sin

        print pi, sin(1), cos(1)

3.14159265359 0.841470984808


        ---------------------------------------------------------------

        NameError                       Traceback (most recent call last)

        <ipython-input-5-f965523473a4> in <module>()
          1 from math import pi, sin
          2
    ----> 3 print pi, sin(1), cos(1)


        NameError: name 'cos' is not defined
```

Incluyendo nombre

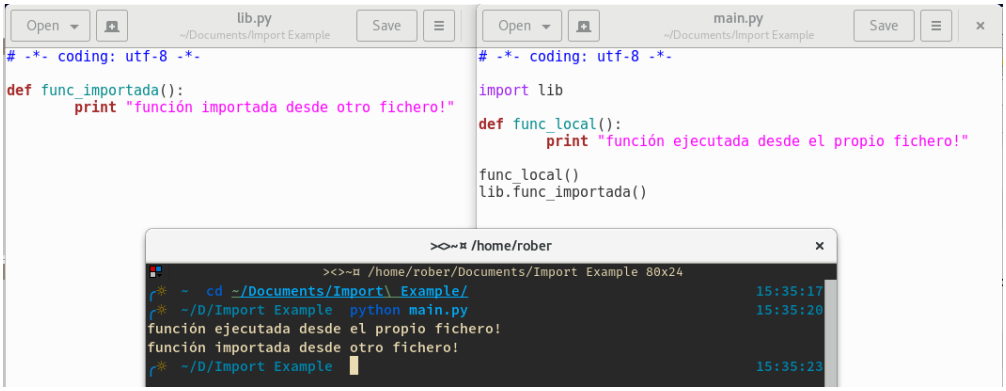
```
In [8]: from math import pi as p, sin as s

        print p, s(1)

3.14159265359 0.841470984808
```

O podemos volcar una librería completa sobre el namespace actual.

Desaconsejado - Facilidad de provocar problemas, particularmente colisión de nombres. Posible ineficiencia si hay muchos objetos. No documenta explicitamente el origen de los objetos.

```
In [6]: from math import *

        print pi, sin(1), cos(1)

 3.14159265359 0.841470984808 0.540302305868
```

También pueden realizarse imports entre ficheros



1.3.2 Excepciones

Incluso las sentencias sintácticamente correctas pueden producir errors cuando se intenta ejecutarlas.

Los errores de tiempo de ejecución se denominan *exceptions*.

Una excepción no tiene por que significar el fin del programa, siempre y cuando se manejen adecuadamente.

```
In [12]: 0 / 0

         ---------------------------------------------------------------------

         ZeroDivisionError                Traceback (most recent call last)

         <ipython-input-12-b761d17a0499> in <module>()
        ----> 1 0 / 0


         ZeroDivisionError: integer division or modulo by zero
```

Las excepciones pertenecen a un tipo dado, incluido en la stack traceback (ej. ZeroDivisionError). Python contiene una serie de tipos básicos, y permite añadir nuevos.

```
In [14]: try:
             print "Producimos un error, y abortamos la ejecución de este bloque"
             0 / 0
             print "Este código nunca se ejecuta"
         except ZeroDivisionError:
             print "Se ha producido un error, pero el programa finaliza correctamente"
```

```
Producimos un error, y abortamos la ejecución de este bloque
Se ha producido un error, pero el programa finaliza correctamente
```

Podemos omitir el tipo de la excepción para capturar cualquiera que se produzca.

```
In [16]: try:
             open('fichero_no_existente.txt')
         except ZeroDivisionError:
             print "Aseguremonos que no divide por cero."
         except:
             print "Producido error al abrir fichero."
```

```
Producido error al abrir fichero.
```

Puede añadirse un **else** adicional, que se ejecuta **cuando la excepción no se produce.**

```
In [ ]: try:
            f = open('fichero')
        except:
            print 'Error al abrir fichero'
        else:
            contenido = leer(f)
            f.close()
```

Se utiliza un else en lugar de código adicional en el try cuando se quiere evitar capturar otras excepciones accidentalmente.

1.3.3 Strings

Python provee una selección de funciones que facilitan el tratamiento de texto.

- Convertir entre mayusculas y minúsculas.
- Comprobar si un texto está en mayúsculas o minúsculas.

```
In [7]: print 'aAaAaAaA'.upper()
        print 'aAaAaAaA'.lower()
```

```
AAAAAAAA
aaaaaaaa
```

```
In [8]: print 'AAAAAAAA'.isupper()
        print 'AAAAAAAA'.islower()
```

```
True
False
```

- Comprobar si el comienzo o final de un texto corresponde con una cadena dada.

```
In [14]: print 'En un lugar de la mancha'.startswith('En')
         print 'En un lugar de la mancha'.endswith('mancha')

True
True
```

```
In [16]: print 'En un lugar de la mancha'.endswith('MANCHA'.lower())

True
```

- Comprobar si una cadena dada se encuentra en el texto.
- Comprobar donde se encuentra dicha cadena.

```
In [24]: print 'mancha' in 'En un lugar de la mancha, de cuyo nombre no quiero...'
         print 'mancha' not in 'En un lugar de la mancha, de cuyo nombre no quiero...'
         print not 'mancha' in 'En un lugar de la mancha, de cuyo nombre no quiero...'

True
False
False
```

```
In [26]: print 'En un lugar de la mancha, de cuyo nombre no quiero...'.find('mancha')
         print 'En un lugar de la mancha, de cuyo nombre no quiero...'.find('Quijote')

18
-1
```

Podemos usar un bloque for para iterar sobre los caracteres de un string

```
In [27]: for c in 'abracadabra':
             print c

a
b
r
a
c
a
d
a
b
r
a
```

- Podemos dividir una string en partes (convertirla en una lista)
- Podemos convertir una lista de strings en una sola

En ambos casos utilizamos separadores

```
In [29]: print 'En un lugar de la mancha'.split()
         print 'Enmiauunmiaulugarmiaudemiaulamiaumancha'.split('miau')
```

```
['En', 'un', 'lugar', 'de', 'la', 'mancha']
['En', 'un', 'lugar', 'de', 'la', 'mancha']
```

```
In [30]: separador, miau = ' ', 'miau'
         print separador.join(['En', 'un', 'lugar', 'de', 'la', 'mancha'])
         print miau.join(['En', 'un', 'lugar', 'de', 'la', 'mancha'])
```

```
En un lugar de la mancha
Enmiauunmiaulugarmiaudemiaulamiaumancha
```

Formateo de cadenas

Dos métodos comunmente utilizados: * mediante el operador % * mediante el método .format()

- Introducir un solo valor

```
In [45]: 'Numero %d.' % 5
```

```
Out[45]: 'Numero 5.'
```

- Múltiples valores

```
In [46]: '%s %i.' % ('Numero', 5)
```

```
Out[46]: 'Numero 5.'
```

Permite añadir padding, especificar el número de decimales, incluir el signo de los números...
No requiere especificar el tipo de las variables.

- Sin especificar orden.

```
In [49]: 'Los numeros {} y {}'.format(5, 'seis')
```

```
Out[49]: 'Los numeros 5 y seis'
```

- Espeficicando el orden.

```
In [50]: 'Los numeros {1} y {0}'.format(5, 'seis')
```

```
Out[50]: 'Los numeros seis y 5'
```

Información adicional

1.3.4 Ficheros

La apertura de archivos se realiza mediante la función open(), que toma dos argumentos: * una ruta hasta el fichero a leer. * un modo de apertura, según las operaciones que queramos realizar

modo	r	r+	w	w+	a	a+
leer	x	x		x		x
escribir		x	x	x	x	x
crear			x	x	x	x
truncar	x	x	x	x		
inicio	x	x	x	x		
fin					x	x

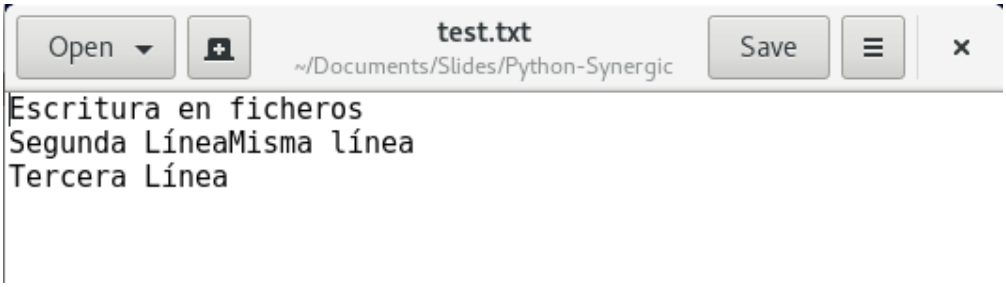
- añadimos una b para leer / escribir en binario (ej. rb / wb)
- **importante** cerrar los ficheros una vez terminamos con la función close

Lectura y escritura

La función **write** escribe cadenas de texto sobre el fichero. La función **writelines** escribe una colección de cadenas de texto.

Los saltos de línea no se incluyen automáticamente.

```
In [54]: f = open('test.txt', 'w')

         f.write('Escritura en ficheros\n')
         f.write('Segunda Línea')
         f.write('Misma línea\n')
         f.write('Tercera Línea')

         f.close()
```



La función **read** devuelve una cadena de texto con el contenido completo del fichero. La función **readlines** devuelve una colección de cadenas de texto, una por línea.

```
In [57]: f = open('test.txt', 'r')
         print f.read()
         f.close()

         f = open('test.txt', 'r')
         print f.readlines()
         f.close()
```

```
Escritura en ficheros
Segunda LíneaMisma línea
Tercera Línea
['Escritura en ficheros\n', 'Segunda L\xc3\xadneaMisma l\xc3\xadnea\n', 'Tercera L\xc3\xadnea']
```

Los bloques with permiten abrir ficheros, y los cierran de manera automática al terminar.

```
In [59]: with open('test.txt', 'r') as infile:
             print infile.read()
```

```
Escritura en ficheros
Segunda LíneaMisma línea
Tercera Línea
```

Permiten utilizar más de un fichero en el mismo bloque sin necesidad de anidarlos.

```
In [61]: with open('test.txt', 'r') as infile, open('test2.txt', 'w') as outfile:
             outfile.write(infile.read())

         with open('test2.txt', 'r') as f:
             print f.read()
```

```
Escritura en ficheros
Segunda LíneaMisma línea
Tercera Línea
```

1.4 Módulo 3A - Colecciones

1.4.1 Listas

Secuencia ordenada mutable heterogenea

- Secuencia: contiene una serie de datos uno tras otro
- Ordenada: los contenidos tienen un orden definido
- Mutable: los contenidos pueden ser modificados
- Heterogenea: los contenidos no tienen por que tener un solo tipo

```
In [65]: lista = [1, 2, 3, "lista"]
         print lista
```

```
[1, 2, 3, 'lista']
```

```
In [66]: for elemento in lista: print elemento
```

```
1
2
3
lista
```

```
In [67]: print type(lista)
```

```
<type 'list'>
```

Acceso a elementos

- Se utiliza un indice que comienza en cero.
- Podemos contar desde el principio o el final (con números negativos)

```
In [70]: print lista[0]
         print lista[2]
```

```
1
3
```

```
In [72]: print lista[-1]
         print lista[-3]
```

```
lista
2


In [74]: print len(lista)
         print lista[len(lista)]

4



        ---------------------------------------------------------------------------

        IndexError                                Traceback (most recent call last)

        <ipython-input-74-6d23ce90aa7d> in <module>()
          1 print len(lista)
    ----> 2 print lista[len(lista)]


        IndexError: list index out of range
```

Slices

Podemos obtener slices (sub-secuencias) de iterables, especificando el inicio y el fin (incluido y excluido respectivamente).

```
In [77]: print lista[1:3]

[2, 3]
```

El inicio y fin de una sub-secuencia es por defecto el de la secuencia original.

```
In [79]: print lista[2:]
         print lista[:2]
         print lista[:]

[3, 'lista']
[1, 2]
[1, 2, 3, 'lista']
```

Nuevamente podemos usar valores negativos.

```
In [80]: print lista[1:-1]

[2, 3]
```

Las listas son mutables, por lo cual podemos reasignar elementos, insertarlos y eliminarlos.

```
In [81]: print lista
         lista[1] = 'nuevo elemento'
         print lista
```

```
[1, 2, 3, 'lista']
[1, 'nuevo elemento', 3, 'lista']
```

```
In [82]: lista.append('append') # añadir al final
         print lista

[1, 'nuevo elemento', 3, 'lista', 'append']
```

```
In [87]: lista.insert(0, True) # añadir en cualquier indice
         print lista

[True, 1, 'nuevo elemento', 3, 'lista', 'append']
```

```
In [88]: del lista[0] # eliminar por indice
         lista.remove('append') # eliminar por valor
         print lista

[1, 'nuevo elemento', 3, 'lista']
```

Ordenación

- Existe dos métodos de ordenación: **sorted** y **sort**.
  - **sorted** devuelve una copia ordenada de la secuencia.
  - **sort** ordena la lista y devuelve None.
- La función **reverse** permite invertir la ordenación de una lista.

```
In [111]: lista = [3, 1, 5, 6, 2, 3, 1]

          print sorted(lista), lista

[1, 1, 2, 3, 3, 5, 6] [3, 1, 5, 6, 2, 3, 1]
```

```
In [112]: print lista.sort(), lista

None [1, 1, 2, 3, 3, 5, 6]
```

```
In [113]: print lista.reverse(), lista

None [6, 5, 3, 3, 2, 1, 1]
```

1.4.2 Tuplas

Secuencia ordenada inmutable heterogenea

- Secuencia: contiene una serie de datos uno tras otro
- Ordenada: los contenidos tienen un orden definido
- Mutable: los contenidos **NO** pueden ser modificados
- Heterogenea: los contenidos no tienen por que tener un solo tipo

```
In [93]: tupla = (1, 2, 'hola')
         print tupla

(1, 2, 'hola')


In [90]: for elemento in tupla: print elemento

1
2
hola


In [92]: print type(tupla)

<type 'tuple'>
```

El acceso a elementos funciona igual que en las listas, con la excepción de que no podemos añadir, reasignar o eliminar elementos.

```
In [96]: print tupla[1:]

(2, 'hola')


In [100]: tupla[0] = 5


        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-100-ccca35294a64> in <module>()
    ----> 1 tupla[0] = 5


        TypeError: 'tuple' object does not support item assignment


In [102]: tupla.append(3)


        ---------------------------------------------------------------------------

        AttributeError                            Traceback (most recent call last)

        <ipython-input-102-b95351daa86b> in <module>()
    ----> 1 tupla.append(3)


        AttributeError: 'tuple' object has no attribute 'append'


In [103]: del tuple[0]
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-103-a30ca5af21ca> in <module>()
----> 1 del tuple[0]


TypeError: 'type' object does not support item deletion
```

Pueden crearse tuplas de un solo elemento.

```
In [99]: tupla_incorrecta = (1)
         tupla_correcta = (1,)

         print type(tupla_incorrecta), tupla_incorrecta
         print type(tupla_correcta), tupla_correcta
```

```
<type 'int'> 1
<type 'tuple'> (1,)
```

1.4.3 Strings como secuencias

Las cadenas de caracteres pueden tratarse como secuencias, como vimos al iterar.
Podemos aplicar sobre ellas tecnicas como las slices, medir su longitud...
Notese que las strings son inmutables.

```
In [105]: string = 'cadena de caracteres'
          print len(string)
```

```
20
```

```
In [108]: print string[1:-1]
```

```
adena de caractere
```

```
In [110]: print sorted(string)
```

```
[' ', ' ', 'a', 'a', 'a', 'a', 'c', 'c', 'c', 'd', 'd', 'e', 'e', 'e', 'e', 'n', 'r', 'r', 's', 't']
```

Strip

El método strip elimina todos los caracteres de espacio en blanco (espacio, tabulador, salto de línea...) **del comienzo y final** de una cadena.

```
In [5]: cadena = """
                    esta cadena contiene espacios en blanco\ty tabuladores\t\n y
        saltos de linea
        """

        print cadena.strip(), '.'
```

```
esta cadena contiene espacios en blanco        y tabuladores
 y
saltos de linea .
```

Los caracteres a eliminar pueden especificarse por parametro.

```
In [6]: cadena = '12222112cadena22211'
        print cadena.strip('12')

cadena
```

Los métodos lstrip y rstrip cumplen la misma función pero solo en el comienzo o final de la cadena respectivamente.

```
In [7]: cadena = '12222112cadena22211'
        print cadena.lstrip('12')
        print cadena.rstrip('12')

cadena22211
12222112cadena
```

1.5 Módulo 3B - Colecciones

1.5.1 Sets

Colección no ordenada mutable heterogenea de elementos únicos

- Colección no ordenada: los contenidos no tienen un orden definido, carece de ciertas operaciones asociadas a secuencias.
- Mutable: los contenidos pueden ser modificados
- Heterogenea: los contenidos no tienen por que tener un solo tipo
- Elementos únicos: contiene o no un elemento, nunca múltiples

```
In [10]: conjunto = {1, 2, 3}
         print conjunto

set([1, 2, 3])
```

```
In [11]: for elemento in conjunto: print elemento

1
2
3
```

```
In [12]: print type(conjunto)

<type 'set'>
```

No permite el acceso a elementos independientes (**no ordenado**).

```
In [13]: print conjunto[0]
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-13-d2b158a85c99> in <module>()
----> 1 print conjunto[0]


TypeError: 'set' object does not support indexing
```

```
In [14]: print conjunto[:-1]
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-14-c8a893983b3c> in <module>()
----> 1 print conjunto[:-1]


TypeError: 'set' object has no attribute '__getitem__'
```

Añadir valores

- de uno en uno mediante *add*
- en bloque con *update,* pasando un iterable como argumento

```
In [25]: conjunto = {1, 2, 3, 4}
         conjunto.add(5)
         print conjunto
```

```
set([1, 2, 3, 4, 5])
```

```
In [26]: conjunto.update([6, 7, 8])
         print conjunto
```

```
set([1, 2, 3, 4, 5, 6, 7, 8])
```

Contiene valores **únicos**. Añadir valores adicionales no produce ningún resultado.
Como contar el número de valores únicos de una lista:

```
In [35]: lista = [1, 5, 4, 1, 4, 6, 3, 5, 2, 1, 6, 2, 3, 6, 8, 2, 1, 5, 6]
         conjunto = set(lista)
         print len(lista), len(conjunto)
```

```
19 7
```

```
In [36]: lista = lista + [8, 8, 8, 8, 8, 8, 8, 8, 8, 8]
         conjunto = set(lista)
         print len(lista), len(conjunto)
```

```
29 7
```

No podemos introducir elementos mutables dentro de un set.

```
In [37]: set([[1], [2], [3]])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)

<ipython-input-37-97c6e4bc6fef> in <module>()
----> 1 set([[1], [2], [3]])


TypeError: unhashable type: 'list'
```

Existen varias maneras de eliminar elementos de un set:

- **remove:** elimina el elemento dado y lanza KeyError si no lo encuentra
- **discard:** elimina el elemento dado si está presente
- **pop:** elimina y devuelve un elemento al azar, devuelve KeyError en el set vacio
- **clear:** vacía el set

```
In [50]: conjunto = {1, 2, 3, 4, 5}

         conjunto.remove(4)
         print conjunto

         conjunto.discard(3)
         print conjunto

         print conjunto.pop()

         conjunto.clear()
         print conjunto
```

```
set([1, 2, 3, 5])
set([1, 2, 5])
1
set([])
```

Eficiente a la hora de realizar operaciones de pertenencia.

```
In [15]: lista = range(10000000)
         conjunto = set(lista)

In [19]: %time 999999 in lista
```

```
CPU times: user 20.5 ms, sys: 0 ns, total: 20.5 ms
Wall time: 20.6 ms
```

```
Out[19]: True
```

```
In [20]: %time 999999 in conjunto

CPU times: user 12 µs, sys: 0 ns, total: 12 µs
Wall time: 15 µs


Out[20]: True
```

Operaciones con sets

- es subconjunto / superconjunto (<=, >=)
- unión (|)
- intersección (&)
- diferencia y diferencia simétrica (-, ^)

- **Subconjunto:** todo elemento del primero está en el segundo
- **Superconjunto:** todo elemento del segundo está en el primero

```
In [39]: print {1, 2} <= {1, 2, 3, 4, 5}
         print {1, 2} <= {1, 3, 4, 5}

True
False


In [40]: print {1, 2, 3, 4, 5} >= {1, 2}
         print {1, 3, 4, 5} >= {1, 2}

True
False
```

- **Unión:** nuevo set con los elementos de ambos
- **Intersección:** nuevo set con los elementos comunes

```
In [41]: print {1, 2, 3, 4, 5} | {4, 5, 6, 7, 8}

set([1, 2, 3, 4, 5, 6, 7, 8])


In [42]: print {1, 2, 3, 4, 5} & {4, 5, 6, 7, 8}

set([4, 5])
```

- **Diferencia:** conjunto con los elementos en el primero pero no en el segundo
- **Diferencia simétrica:** conjunto con los elementos que están en un conjunto, pero no en ambos

```
In [44]: print {1, 2, 3, 4} - {3, 4}

set([1, 2])


In [46]: print {1, 2, 3, 4} ^ {3, 4, 5}

set([1, 2, 5])
```

Frozen sets

Colección no ordenada inmutable heterogenea de elementos únicos

- Colección no ordenada: los contenidos no tienen un orden definido, carece de ciertas operaciones asociadas a secuencias.
- Inmutable: los contenidos no pueden ser modificados
- Heterogenea: los contenidos no tienen por que tener un solo tipo
- Elementos únicos: contiene o no un elemento, nunca múltiples

```
In [56]: conjunto = frozenset([1, 2, 3, 4])

         print conjunto & {3, 4, 5, 6}

         conjunto.add(9)
```

```
frozenset([3, 4])
```

```
        ---------------------------------------------------------------------------
        AttributeError                            Traceback (most recent call last)

        <ipython-input-56-b75f9b0e34fb> in <module>()
          3 print conjunto & {3, 4, 5, 6}
          4
    ----> 5 conjunto.add(9)


        AttributeError: 'frozenset' object has no attribute 'add'
```

1.5.2 Diccionarios

Colección no ordenada mutable heterogenea de pares clave / valor

- Colección no ordenada: los contenidos no tienen un orden definido, carece de ciertas operaciones asociadas a secuencias.
- Mutable: los contenidos pueden ser modificados
- Heterogenea: los contenidos no tienen por que tener un solo tipo
- Pares clave valor: formado por valores indexados en función a una clave **inmutable**

```
In [61]: diccionario = {'a': 1, 'b': 2, 'c': 3}
         print diccionario
```

```
{'a': 1, 'c': 3, 'b': 2}
```

```
In [62]: for clave in diccionario: print clave, diccionario[clave]
```

```
a 1
c 3
b 2
```

```
In [60]: print type(diccionario)
```

```
<type 'dict'>
```

Se **introducen** y **modifican** valores mediante asignación.

```
In [3]: empresa = {
            'nombre': 'Synergic Partners',
            'empleados': 100
        }

        empresa['sedes'] = ['Madrid', 'Terrassa', 'Munich'] # añadimos nueva clave/valor
        print empresa

        empresa['sedes'] = ['Madrid', 'Terrassa'] # editamos un valor existente
        print empresa
```

```
{'nombre': 'Synergic Partners', 'sedes': ['Madrid', 'Terrassa', 'Munich'], 'empleados': 100}
{'nombre': 'Synergic Partners', 'sedes': ['Madrid', 'Terrassa'], 'empleados': 100}
```

Solo **un valor por clave** (se permiten colecciones).

```
In [8]: empresa = {
            'nombre': 'Synergic Partners',
            'empleados': 100
        }
        print empresa

        empresa['empleados'] = ['Carme', 'Jaume']
        print empresa
```

```
{'nombre': 'Synergic Partners', 'empleados': 100}
{'nombre': 'Synergic Partners', 'empleados': ['Carme', 'Jaume']}
```

```
In [9]: empresa = {
            'nombre': 'Synergic Partners',
            'empleados': []
        }

        empresa['empleados'].append('Carme')
        empresa['empleados'].append('Jaume')
        print empresa
```

```
{'nombre': 'Synergic Partners', 'empleados': ['Carme', 'Jaume']}
```

Eliminación de contenido

- eliminar un par clave/valor con del
- eliminar todo el contenido de un diccionario

```
In [7]: diccionario = {1:2, 3:4, 5:6}
        print diccionario

        del diccionario[1]
```

```
        print diccionario

        diccionario.clear()
        print diccionario
```

```
{1: 2, 3: 4, 5: 6}
{3: 4, 5: 6}
{}
```

Iterar por un diccionario

- Iterar sobre claves
- Iterar sobre pares clave / valor

```
In [11]: diccionario = {'a': 1, 'b': 2}

         for k in diccionario: print k, diccionario[k]
```

```
a 1
b 2
```

```
In [13]: for k, v in diccionario.items(): print k, v
```

```
a 1
b 2
```

Valor por defecto
Podemos referenciar una clave sin preocuparnos si existe en el diccionario o no.

- si la clave existe, obtenemos su valor correspondiente
- si no obtenemos None (o un valor especificado por parámetro) en lugar de una excepción.

```
In [18]: diccionario = {1:2, 3:4, 5:6}

         print diccionario.get(1)
         print diccionario.get(18)
         print diccionario.get(18, -1)
```

```
2
None
-1
```

El operador de pertenencia comprueba si una clave se encuentra en el diccionario.

```
In [20]: print 1 in diccionario
         print 2 in diccionario
         print 2 in diccionario.values()
```

```
True
False
True
```

Podemos **concatenar** dos diccionarios (las claves repetidas se sobreescriben).

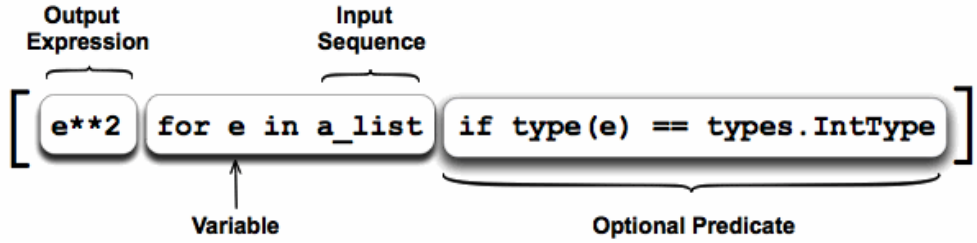
```
In [1]: a = {1:2, 3:4, 5:6}
        b = {7:8, 5:'repetido'}

        a.update(b)
        print a
```

```
{1: 2, 3: 4, 5: 'repetido', 7: 8}
```

1.5.3 Comprehension

Sentencias que permiten construir secuencias a partir de otras secuencias.

Consta de las siguientes partes: * una secuencia de entrada * una variable que representa cada elemento de la entrada iterativamente * una expresión de salida que produce elementos de la secuencia de salida a partir de elementos de la secuencia de entrada * un predicado condicional (opcional)



(source)

Una comprehension combina operaciones de mapeo y filtrado sobre una colección existente para producir una nueva

Generación de una lista de números que crecen cubicamente.

Sin comprehension

```
In [15]: def lista_cubo():
             lista = []
             for i in range(10):
                 lista.append(i ** 2)
             return lista

         print lista_cubo()
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Con comprehension

```
In [4]: [i ** 2 for i in range(10)]
```

```
Out[4]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Como hemos visto, las comprehension pueden incluir una clausula condicional, actuando como un filtro.

```
In [18]: [x for x in range(10) if x % 2]
```

```
Out[18]: [1, 3, 5, 7, 9]
```

Podemos anidar comprehensions para construir colecciones anidadas dentro de otras colecciones.

```
In [21]: [[i + j for i in range(3)]for j in range(3)]
```

```
Out[21]: [[0, 1, 2], [1, 2, 3], [2, 3, 4]]
```

Y encadenar comprehensions entre si (ojo con el orden!)

```
In [22]: [str(i) + " " + str(j) for i in range(3) for j in range(2)]

Out[22]: ['0 0', '0 1', '1 0', '1 1', '2 0', '2 1']
```

Las comprehension no están limitadas a listas.
Podemos extender la misma mecánica a sets:

```
In [29]: a = {x + 1 for x in range(3)}

         print a, type(a)

set([1, 2, 3]) <type 'set'>
```

Y a diccionarios:

```
In [30]: d = {x: [] for x in range(4)}

         print d, type(d)

{0: [], 1: [], 2: [], 3: []} <type 'dict'>
```

1.5.4 Generators

Un generador es un tipo de función que no termina en el sentido normal, sino que produce un valor y espera volver a ser llamada para producir el siguiente, tantas veces como fuera necesario.

```
In [32]: def contra_ejemplo():
             return 1
             return 2
             return 3

         print contra_ejemplo()
         print contra_ejemplo()
         print contra_ejemplo()

1
1
1
```

```
In [36]: def ejemplo():
             yield 1; yield 2; yield 3

         generador = ejemplo()
         print next(generador)
         print next(generador)
         print next(generador)

1
2
3
```

El uso de generadores permite generar grandes cantidades de valores (listas infinitas) sin mantener más que un elemento en memoria.

```
In [46]: def fibonacci_gen(a, b):
             yield a
             yield b
             while True:
                 c = a + b
                 a = b
                 b = c
                 yield c

         g = fibonacci_gen(1, 1)
         for i in range(100): print next(g),
```

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657 46368 75025 121393
```

1.6 Módulo 4A - Conceptos Avanzados

1.6.1 Programación orientada a objetos

> Paradigma de programación basado en el concepto de "objetos": estructuras que pueden contener tantos datos como código capaz de interactuar con dichos datos; agrupaciones de estado y comportamiento relacionados.

- Los atributos (datos / variables) encapsulan el estado o las características de un objeto.
- Los métodos (código / funciones) encapsulan el comportamiento de dicho objeto.

Clases e instancias

- Una clase es la plantilla que describe los detalles de un objeto, a partir de las cuales se crean los mismos. Está compuesta por un nombre y una serie de atributos y operaciones. **Crear una clase no crea un objeto**.
- Una instancia es cada objeto individual, creado a partir de la plantilla provista por la clase, y con valores y estado propios. **Cada instancia de una clase es un objeto**.

```
In [3]: class Test(object): pass

        t1, t2 = Test(), Test()

        print t1
        print t2
```

```
<__main__.Test object at 0x7f0e13b5b2d0>
<__main__.Test object at 0x7f0e13b5b310>
```

- Python 2 requiere que las clases "hereden" explicitamente de object o de otra clase - de ahí la notación *Clase(object)*
- Python 3 hace que ese requerimiento sea opcional, pudiendo omitirlo

Por convención, los nombres de las clases utilizan nombres en CamelCase: sin separación entre palabras, todas comienzan por mayuscula.

1.6.2 Atributos

- Los atributos son las variables de clase que contienen el estado del objeto.
- Los atributos de una instancia determinada pueden ser modificados desde fuera.

```
In [9]: class Empresa(object):
            nombre = "Synergic Partners"
            empleados = 100

        sp = Empresa()
        print "{} tiene {} empleados.".format(sp.nombre, sp.empleados)

        sp.empleados += 8
        print "{} tiene {} empleados.".format(sp.nombre, sp.empleados)
```

```
Synergic Partners tiene 100 empleados.
Synergic Partners tiene 108 empleados.
```

1.6.3 Métodos

- Los métodos son funciones que actuan sobre los atributos de la propia clase.
- La palabra clave **self** se utiliza para referirse a la instancia actual de la clase.
- El parámetro self se provee automáticamente (toma como valor la referencia a la instancia sobre la que se llama).

```
In [10]: class Empresa(object):
             nombre = "Synergic Partners"
             empleados = 100

             def contratar(self, num):
                 self.empleados += num

         sp = Empresa()
         print "{} tiene {} empleados.".format(sp.nombre, sp.empleados)

         sp.contratar(5)
         print "{} tiene {} empleados.".format(sp.nombre, sp.empleados)
```

```
Synergic Partners tiene 100 empleados.
Synergic Partners tiene 105 empleados.
```

Los atributos de una clase pueden inicializarse mediante la implementación del método **init**.
Este método debe contener toda la lógica que queremos ejecutar antes de empezar a utilizar la instancia.

```
In [14]: class Empresa(object):
             def __init__(self, nombre, empleados=0):
                 self.nombre = nombre
                 self.empleados = empleados
                 self.cafes_diarios = empleados * 5

             def contratar(self, num):
                 self.empleados += num
```

```
        sp = Empresa("Synergic Partners", 100)
        print "{} tiene {} empleados.".format(sp.nombre, sp.empleados)

        st = Empresa("Telefónica", 200000000)
        print "{} tiene {} empleados.".format(st.nombre, st.empleados)
```

```
Synergic Partners tiene 100 empleados.
Telefónica tiene 200000000 empleados.
```

Al igual que **init**, existen otras operaciones nativas que podemos re-implementar para un objeto concreto.

Por ejemplo, **str** contiene la representación textual del objeto que aparece al llamar *print* sobre el.

```
In [16]: class Empresa(object):
             def __init__(self, nombre, empleados=0):
                 self.nombre = nombre
                 self.empleados = empleados
                 self.cafes_diarios = empleados * 5

             def __str__(self):
                 return "[{}] - empleados: {}, cafés diarios: {}".format(
                     self.nombre, self.empleados, self.cafes_diarios)

         sp = Empresa("Synergic Partners", 100)
         print sp
```

```
[Synergic Partners] - empleados: 100, cafés diarios: 500
```

1.6.4 Lambdas

Una lambda es una función anonima creada en tiempo de ejecución y que contiene una única expresión, cuyo resultado constituye el valor de retorno.

Una lambda puede encontrarse en cualquier punto en que se requiera una función, y no es necesario asignarla a una variable.

```
In [19]: def doble(x): return x * 2

         doble_lambda = lambda x: x * 2

         print doble(2), doble_lambda(2)
```

```
4 4
```

```
In [22]: def incrementador(n): return lambda x: x + n

         mas_dos, mas_tres = incrementador(2), incrementador(3)

         print mas_dos(7), mas_tres(7)
         print incrementador(10)(7)
```

```
9 10
17
```

1.6.5 Filter

Se utiliza para seleccionar una serie de valores de una lista de acuerdo a una función.
Se forma una nueva lista con aquellos valores de la original para los cuales la función devuelve True.

```
In [24]: lista = [2, 18, 9, 22, 17, 24, 8, 12, 27, 31, 56, 72, 9]

         print filter(lambda x: x % 2, lista)
```

```
[9, 17, 27, 31, 9]
```

1.6.6 Map

Se utiliza para aplicar una función a cada uno de los valores de una lista, y forma una nueva lista con los resultados.

```
In [26]: palabras  = "en un lugar de la mancha de cuyo nombre no quiero acordarme".split()

         map(lambda x: len(x), palabras)
```

```
Out[26]: [2, 2, 5, 2, 2, 6, 2, 4, 6, 2, 6, 9]
```

1.6.7 Por qué no usar filter / map / reduce

- Las funciones de map y filter pueden replicarse mediante comprehensions.
- No es necesario aprender funciones adicionales.
- Las comprehensions resultan ligeramente más rápidas.
- Las comprehensions son mucho más sencillas de componer

```
In [31]: lista = range(10000000)

         %time filter(lambda x: x % 2, lista)
         %time [x for x in lista if x % 2]
         print
```

```
CPU times: user 1.12 s, sys: 41.5 ms, total: 1.16 s
Wall time: 1.17 s
```

```
CPU times: user 973 ms, sys: 15.2 ms, total: 989 ms
Wall time: 991 ms
```

```
In [32]: lista = range(10000000)

         %time map(lambda x: x ** 2, lista)
         %time [x ** 2 for x in lista]
         print
```

```
CPU times: user 1.69 s, sys: 47.2 ms, total: 1.73 s
Wall time: 1.71 s
CPU times: user 1.25 s, sys: 29.1 ms, total: 1.28 s
Wall time: 1.27 s
```

```
In [35]: lista = range(10)

         print map(lambda x: x ** 2, filter(lambda x: x % 2, lista))
         print [x ** 2 for x in lista if x % 2]
```

```
[1, 9, 25, 49, 81]
[1, 9, 25, 49, 81]
```

1.6.8 Reduce

Se utiliza para combinar todos los valores de la lista y producir un único valor final.

Requiere una función que tome dos argumentos, que representan un acumulador y el elemento actual (desde el segundo en adelante).

(functools.reduce en python 3)

```
In [27]: palabras  = "en un lugar de la mancha de cuyo nombre no quiero acordarme".split()

         reduce(lambda x, y: x + "_" + y, palabras)
```

```
Out[27]: 'en_un_lugar_de_la_mancha_de_cuyo_nombre_no_quiero_acordarme'
```