

Introducción rápida a Python

Mario García-Valdez

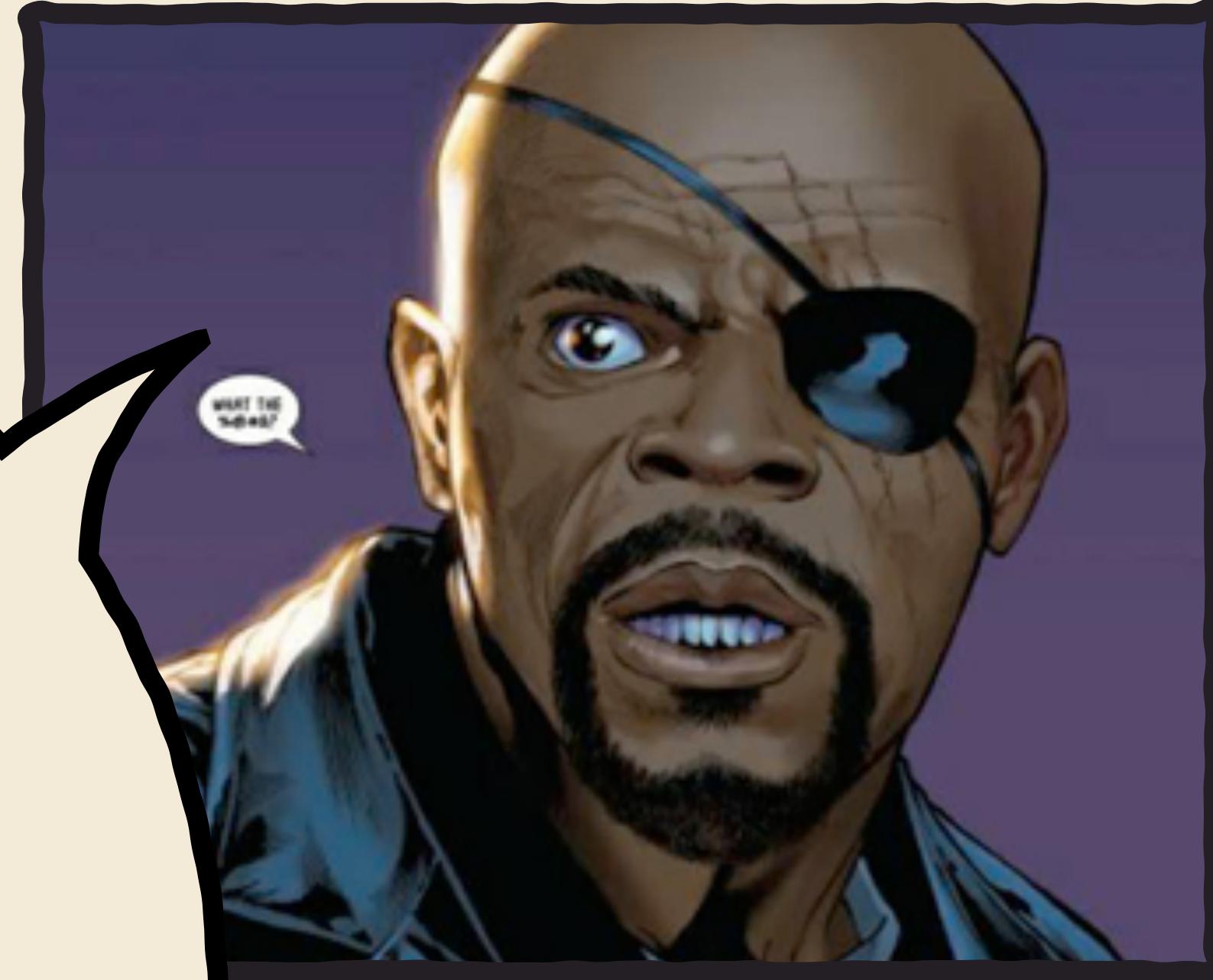
¿Python?

- Es un lenguaje dinámico e interpretado.
- De código libre.
- Amigable para principiantes.
- Disponible en:

<http://www.python.org>

```
]:
    x+5

def dotwrite(ast):
    nodename = getNodeName()
    if symbol.sym_name.get(int(ast[0]), ast[0]):
        print '%s [%label=%s] %s' % (nodename, label,
                                      ast[1])
    elif isinstance(ast[1], str):
        if ast[1].strip():
            print '= %s'; ' % ast[1]
        else:
            print ''
    else:
        print '';
        children = []
        for n, child in enumerate(ast[1:]):
            children.append(dotwrite(child))
        print '%s -> (%s %s)' % (nodename,
                                   len(children),
                                   ', '.join(children))
```





- Desarrollado por Guido van Rossum a principio de los noventas.
- El nombre es por el grupo de cómicos ingleses **Monty Python**.

Ejemplo Básico

```
x = 34 - 23          # Comentario
y = "Hola"           # Otro
z = 3.45
if z == 3.45 or y == "Hola":
    x = x + 1
    y = y + " Mundo" # Concatenación de cadenas

print(x)
print(y)
```

Entendiendo el código

La indentación es parte del lenguaje

Los tipos de las “variables” no se tienen que declarar

La asignación usa `=` y se compara con `==`

Los símbolos `+ - * / %` funcionan como siempre

Se concatena con `+`.

Se usa de forma especial `%` para derle formato a las cadenas (como el printf de C)

Los operadores lógicos son palabras (`and, or, not`)

La función para imprimir en pantalla es `print()`.

Tipos Básicos

Enteros

```
z = 5 // 2  
# El resultado es 2, división entera.
```

Flotantes

```
x = 3.456
```

Cadenas

Se puede usar “ ” o ‘ ’ para indicarlas.

“abc” ‘abc’ (Son lo mismo.)

En caso de conflicto se usan ambas.

“matt’s”

Se usan comillas triples para múltiples párrafos o para incluir comillas y apóstrofes:

“““a‘b“c””””

Espacio en Blanco

El espacio en blanco tiene significado en Python:

- En especial la indentación y los saltos de línea.
- Utiliza un salto de línea para terminar una línea de código.
- Se utiliza un \ para que el salto de línea no se considere.

No se usan llaves { } para marcar los bloques.

Se utiliza indentación consistente para esto.

- El primer renglón con más indentación inicia el bloque.
- El primer renglón con menos indentación termina el bloque.

```
if z == 3.45 or y == "Hello":  
    x = x + 1  
    y = y + " World"    # String concat.  
print x  
print y
```

Comentarios

Inician con **#** el resto de la línea se ignora.

Puede incluirse una “cadena de documentación” en la primera línea de una función o clase.

Esta documentación se usa por el ambiente de desarrollo, depuradores y otras herramientas.

Se considera de buen gusto incluir esta documentación.

```
def mi_funcion(x, y):  
    """El docstring. Esta función  
    es bien importante ya que blah blah blah."""  
    # Comentario aqui...
```



Asignación - Atado

El atado (binding) de una variable significa que se asigna a un nombre (una referencia) a cierto objeto

La asignación crea referencias, no copias.

Los nombres en Python no tienen un tipo propio.

Los objetos si.

Python determina el tipo de la referencia de forma automática, dependiendo del tipo de objeto que se asigne.

Creas el nombre la primera vez que aparece a la izquierda de una expresión.

```
x = 3
```

¿tipo? no ¿tipo? int

Asignación - Atado

Si tratas de utilizar un nombre antes de que sea creado, saldrá un error:

```
>>> x = 3
3
>>> y

Traceback (most recent call last):
  File "<pyshell#16>", line 1, in -toplevel-
    y
NameError: name 'y' is not defined
>>> y = 3
>>> y
3
```

Nombres válidos

Python distingue entre minúsculas y mayúsculas.

Los nombres no pueden empezar con número.

Pueden contener letras, números y subguiones (_).

bob Bob _bob __2_bob_ bob_2 BoB

Palabras reservadas:

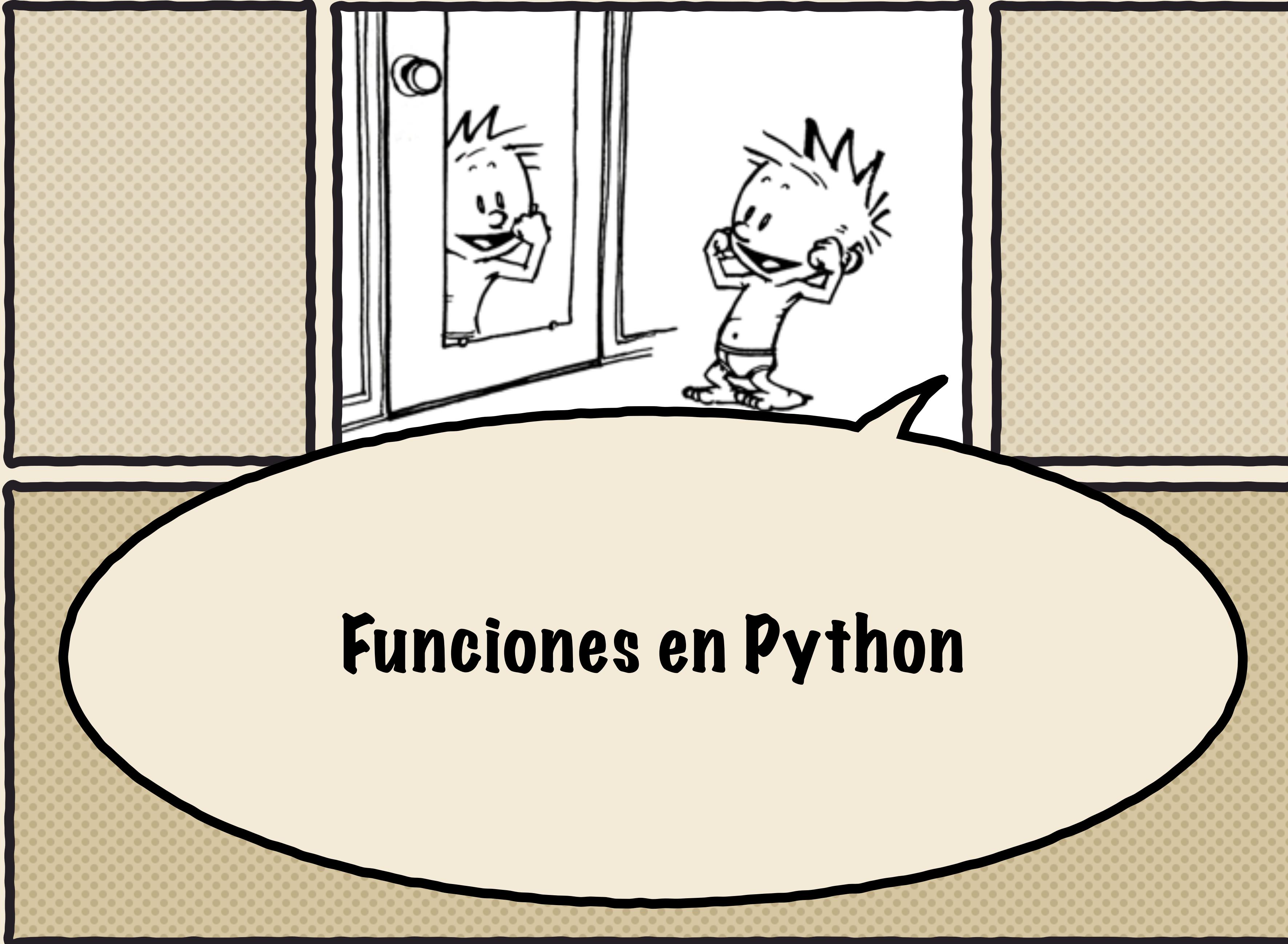
and, assert, break, class, continue, def, del, elif,
else, except, exec, finally, for, from, global, if,
import, in, is, lambda, not, or, pass, print, raise,
return, try, while

Modo Interactivo

En [modo interactivo](#), la última expresión impresa se asigna a la variable `_`.

Esto significa que, cuando se usa Python como calculadora, se facilita continuar los cálculos, por ejemplo:

```
>>> iva = 12.5 / 100
>>> precio = 100.50
>>> precio * iva
12.5625
>>> precio + _
113.0625
>>> round(_, 2)
113.06
>>>
```



Funciones en Python

Como definir funciones

Empiezan con def

Nombre de la función

Argumentos

Dos puntos

```
def get_optimo(filename, path):
```

“Cadena de Documentación”

linea1

linea2

```
return mejor
```

Regresa el control al código
que llamó la función

Este nivel de indentación está
dentro del bloque de la función

La función termina al encontrarse otra línea
con nivel de indentación menor o fin de archivo

No se indica tipo que regresa, ni tipo de argumentos

Llamando a las funciones

La sintaxis para llamar una función es:

```
>>> def producto(x, y):  
        return x * y  
  
>>> producto(3, 4)  
12
```

Los parámetros en Python se llaman “Call By-Sharing”

Los parámetros son referencias a las variables enviadas.

- Si la variable enviada es inmutable, se llama “por valor”
- Si la variable enviada es mutable se llama “por referencia”
- Si a la variable se le asigna otra referencia, por ejemplo a nueva lista, la referencia original no se pierde, funciona como llamado “por valor”

Pase de Parámetros

```
>>> def foo(x):
...     x.append(2)
...     x = [1,2]
...
>>> r = [1,2]
>>> foo(r)
>>> r
[3, 3, 2]
```

Parámetros *

Las funciones pueden recibir un número arbitrario de argumentos

```
>>> def print_args(*args):  
    print args  
  
>>> print_args(3, 4, 10, 'hey')  
(3, 4, 10, 'hey')
```

Como vemos los argumentos se reciben en tuplas.

Si se requiere, se pueden indicar argumentos posicionales.
Deben preceder a los parámetros arbitrarios.

```
>>> def print_args(pos1,pos2,*args):  
    print (pos1, pos2, args)
```

* en el llamado de funciones

Al llamar funciones, indicamos con un asterisco que la secuencia son parámetros enviados por posición.

```
>>> def print_args(a,b,c):  
        print (a,b,c)  
  
>>> print_args(3, 4, 10)  
(3, 4, 10)  
  
>>> a = (3, 4, 10)  
  
>>> print_args(*a)  
(3, 4, 10)
```

Parámetros por nombre arbitrarios **

El mecanismo para recibir un número arbitrario de parámetros con nombre utiliza ahora doble asterisco.

```
>>> def print_args(**kwargs) :  
    print(kwargs)  
  
>>> print_args(a=3, b=4, c=10)  
{'a'=3, 'b'=4, 'c'=10}
```

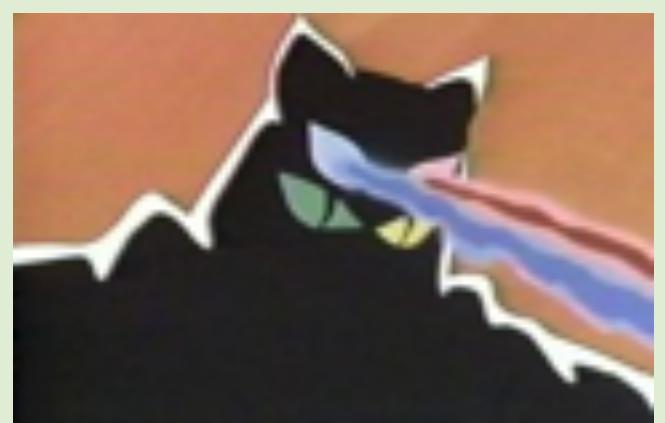
Al igual que con los parámetros por posición, también se puede usar ** al llamar la función, pasando ahora un diccionario.

Funciones sin `return`

Todas las funciones en Python tienen un valor de regreso.

incluso si no hay una línea con `return` dentro de ellas.

Estas funciones regresan un valor `None`



`None` es una constante especial del lenguaje.

`None` es similar a `NULL`, `void`, o `nil` en otros lenguajes.

`None` es también equivalente a Falso.

El intérprete no imprime `None`.

¿Sobrecargado de Funciones? **No**

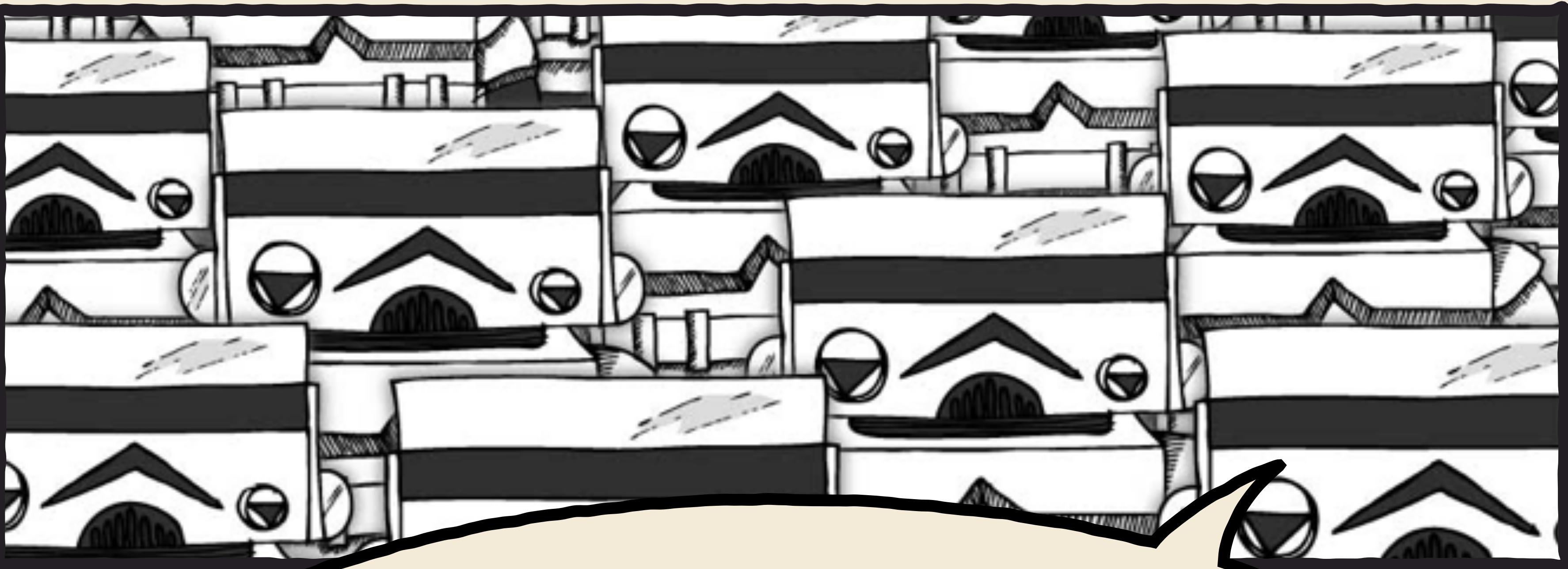
No hay sobrecargado de funciones en Python.

Dos funciones no pueden tener el mismo nombre, incluso si tienen diferentes argumentos.

Funciones “de Fábrica”

El intérprete de Python incluye varias funciones “incluidas de fábrica” las cuales están siempre disponibles:

Built-in Functions				
<code>abs()</code>	<code>divmod()</code>	<code>input()</code>	<code>open()</code>	<code>staticmethod()</code>
<code>all()</code>	<code>enumerate()</code>	<code>int()</code>	<code>ord()</code>	<code>str()</code>
<code>any()</code>	<code>eval()</code>	<code>isinstance()</code>	<code>pow()</code>	<code>sum()</code>
<code>basestring()</code>	<code>execfile()</code>	<code>issubclass()</code>	<code>print()</code>	<code>super()</code>
<code>bin()</code>	<code>file()</code>	<code>iter()</code>	<code>property()</code>	<code>tuple()</code>
<code>bool()</code>	<code>filter()</code>	<code>len()</code>	<code>range()</code>	<code>type()</code>
<code>bytearray()</code>	<code>float()</code>	<code>list()</code>	<code>raw_input()</code>	<code>unichr()</code>
<code>callable()</code>	<code>format()</code>	<code>locals()</code>	<code>reduce()</code>	<code>unicode()</code>
<code>chr()</code>	<code>frozenset()</code>	<code>long()</code>	<code>reload()</code>	<code>vars()</code>
<code>classmethod()</code>	<code>getattr()</code>	<code>map()</code>	<code>repr()</code>	<code>xrange()</code>
<code>cmp()</code>	<code>globals()</code>	<code>max()</code>	<code>reversed()</code>	<code>zip()</code>
<code>compile()</code>	<code>hasattr()</code>	<code>memoryview()</code>	<code>round()</code>	<code>__import__()</code>
<code>complex()</code>	<code>hash()</code>	<code>min()</code>	<code>set()</code>	<code>apply()</code>
<code>delattr()</code>	<code>help()</code>	<code>next()</code>	<code>setattr()</code>	<code>buffer()</code>
<code>dict()</code>	<code>hex()</code>	<code>object()</code>	<code>slice()</code>	<code>coerce()</code>
<code>dir()</code>	<code>id()</code>	<code>oct()</code>	<code>sorted()</code>	<code>intern()</code>

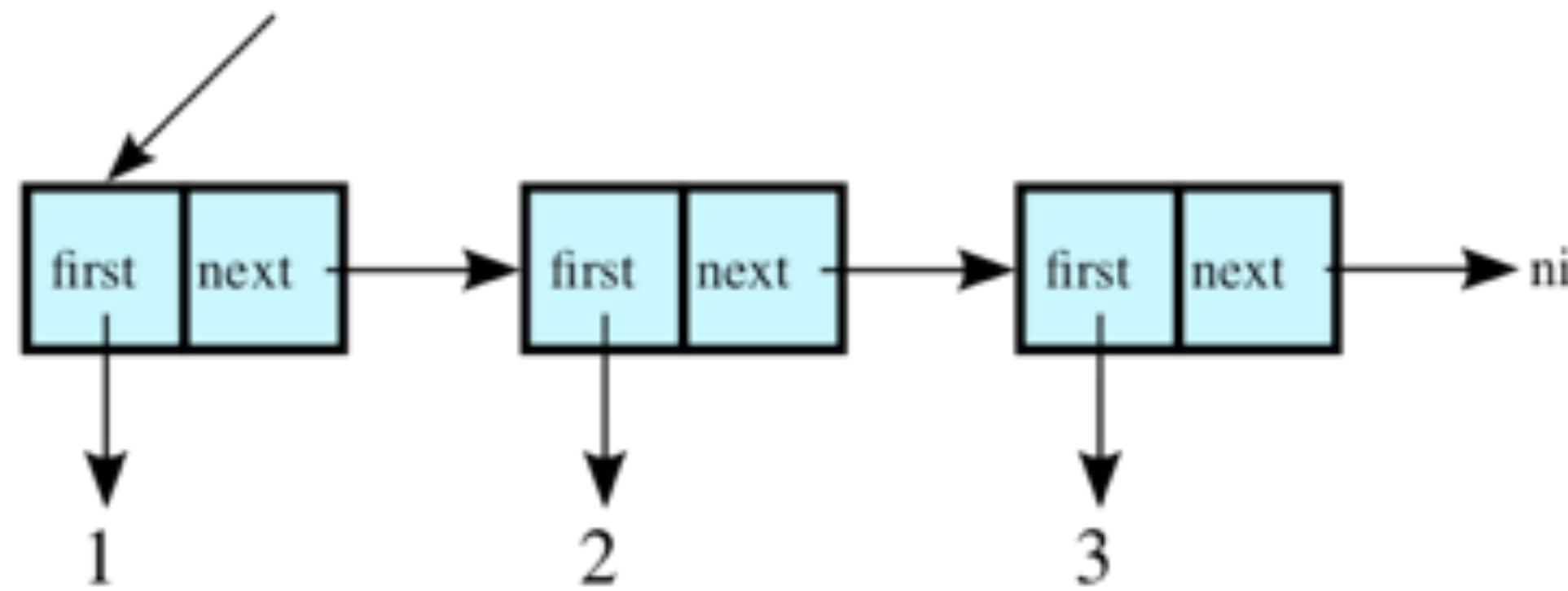


Objetos Tipo Colección

Python utiliza varios tipos de datos compuestos, que se utilizan para agrupar otros valores.

Objetos tipo Secuencia

Tuplas, Listas, Cadenas



Objetos Tipo Secuencia

Tupla

Una secuencia ordenada e **inmutable** de elementos.

Los elementos pueden ser de diferentes tipos, incluyendo otras colecciones.

Cadenas (Strings)

Inmutables

Conceptualmente iguales a las Tuplas.

Listas

Una secuencia **mutable** de elementos de diferentes tipos.

Las secuencias usan una sintaxis similar

Los tres tipos de secuencias comparten la misma funcionalidad y sintaxis.

Diferencia Clave:

Tuplas y cadenas son *inmutables*.

Las listas son *mutables*.

Las operaciones que veremos a continuación, son aplicables a todas las colecciones tipo secuencia.

Colecciones tipo secuencia

- Las tuplas se definen como una lista de valores (elementos) separados por comas, entre **paréntesis** .

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
```

- Las listas igual, solo que entre **corchetes**.

```
>>> li = ["abc", 34, 4.34, 23]
```

- Las cadenas se escriben entre comillas (" , ' , """).

```
>>> st = "Hello World"  
>>> st = 'Hello World'  
>>> st = """Esta es una multi-línea  
que utiliza triple comillas dobles."""
```

Notación tipo arreglo

- Podemos acceder a los elementos individuales de una tupla, lista o cadenas utilizando la notación de corchetes con índices. Como los arreglos clásicos de C#, Java, C.

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')  
>>> tu[1]      # Segundo elemento de la tupla.  
'abc'  
  
>>> li = ["abc", 34, 4.34, 23]  
>>> li[1]      # Segundo elemento de la lista.  
34  
  
>>> st = "Hello World"  
>>> st[1]      # Segundo elemento de la cadena.  
'e'
```

índices positivos y negativos

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Índice positivo: se cuenta de izquierda a derecha empezando en 0.

```
>>> t[1]  
'abc'
```

Índice negativo: se cuenta de derecha a izquierda, iniciando en -1.

```
>>> t[-3]  
4.56
```

Cortes (Slicing)

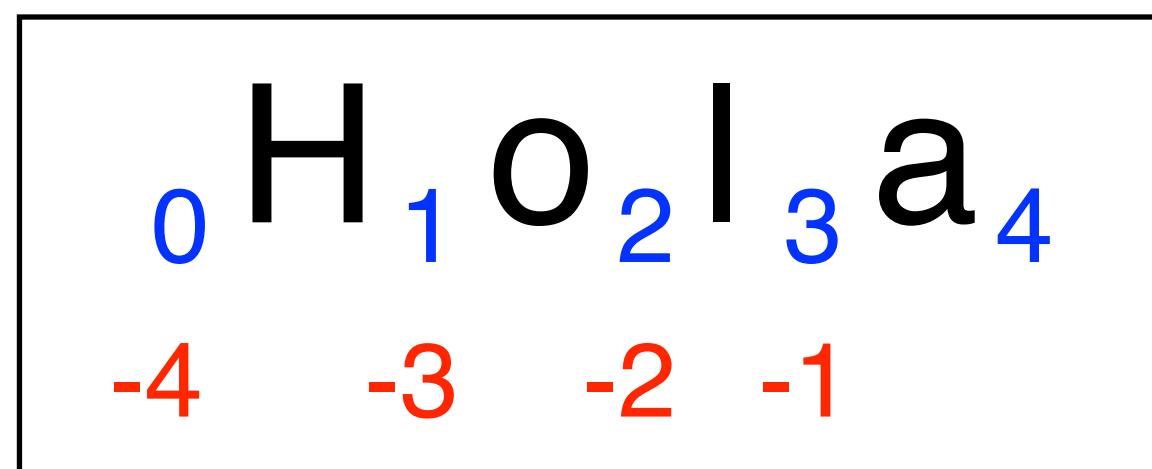
```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Regresa una copia del contenedor con un subconjunto de los miembros originales. Empieza a copiar desde el primer índice y se detiene antes del segundo.

```
>>> t[1:4]  
('abc', 4.56, (2,3))
```

También se pueden usar índices negativos.

```
>>> t[1:-1]  
('abc', 4.56, (2,3))
```



Copiando toda la secuencia

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Para regresar una *copia* de toda la secuencia, puedes usar `[:]`.

```
>>> t[:]  
(23, 'abc', 4.56, (2,3), 'def')
```

La diferencia entre estas dos líneas es muy importante:

```
>>> list2 = list1 # Los nombres hacen referencia a la misma lista  
          # Si cambiamos una se cambian ambas
```

```
>>> list2 = list1[:] # Dos copias diferentes, dos referencias
```

El operador 'in'

Una prueba booleana para ver si un elemento está en la secuencia:

```
>>> t = (1, 2, 4, 5)
>>> 3 in t
False
>>> 4 in t
True
>>> 4 not in t
False
```

En las cadenas, prueba si una subcadena está en la secuencia

```
>>> a = 'abcde'
>>> 'c' in a
True
>>> 'cd' in a
True
>>> 'ac' in a
False
```

El operador +

El operador **+** produce una *nueva* tupla, lista o cadena cuyos valores son la concatenación de los argumentos.

```
>>> (1, 2, 3) + (4, 5, 6)  
(1, 2, 3, 4, 5, 6)
```

```
>>> [1, 2, 3] + [4, 5, 6]  
[1, 2, 3, 4, 5, 6]
```

```
>>> "Hello" + " " + "World"  
'Hello World'
```

El operador *

El operador * produce una *nueva* tupla, lista o cadena cuyos valores son la repetición de los argumentos.

```
>>> (1, 2, 3) * 3  
(1, 2, 3, 1, 2, 3, 1, 2, 3)
```

```
>>> [1, 2, 3] * 3  
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

```
>>> "Hello" * 3  
'HelloHelloHello'
```



Mutabilidad

Tuplas vs Listas

Listas: Mutables

```
>>> li = ['abc', 23, 4.34, 23]  
>>> li[1] = 45  
>>> li  
['abc', 45, 4.34, 23]
```

- Podemos modificar la lista *in place*.
- El nombre *li* sigue apuntando a la misma dirección de memoria, después de la actualización.

Tuplas: Inmutables

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')  
>>> t[2] = 3.14
```

```
Traceback (most recent call last):  
  File "<pyshell#75>", line 1, in -toplevel-  
    tu[2] = 3.14
```

TypeError: object doesn't support item assignment

No puedes cambiar una tupla.

Se puede crear una nueva tupla y asignar la nueva referencia al nombre original.

```
>>> t = (23, 'abc', 3.14, (2,3), 'def')
```

Gracias a esto las tuplas son más eficientes que las listas.

Operaciones solo para Listas (1)

```
>>> li = [1, 11, 3, 4, 5]
```

append()

```
>>> li.append('a')      # Se utiliza un método de la lista  
>>> li  
[1, 11, 3, 4, 5, 'a']
```

insert()

```
>>> li.insert(2, 'i')  
>>> li  
[1, 11, 'i', 3, 4, 5, 'a']
```

Operaciones solo para Listas (2)

extend()

El operador + crea una nueva lista (con una nueva referencia)
extend altera la lista li in place.

```
>>> li.extend([9, 8, 7])  
>>> li  
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7]
```

Cuidado:

extend recibe una lista.

append recibe un solo elemento.

```
>>> li.append([10, 11, 12])  
>>> li  
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7, [10, 11, 12]]
```

Operaciones solo para Listas (3)

```
>>> li = ['a', 'b', 'c', 'b']
```

index()

```
>>> li.index('b')      # indice de primera ocurrencia*
```

1

*existen otras formas

count()

```
>>> li.count('b')      # numero de ocurrencias
```

2

remove()

```
>>> li.remove('b')    # remueve la primera ocurrencia
```

```
>>> li
['a', 'c', 'b']
```

Listas

```
>>> li = [5, 2, 6, 8]
```

reverse()

```
>>> li.reverse()      # ordena en reversa la lista *in place*
>>> li
[8, 6, 2, 5]
```

sort()

```
>>> li.sort()        # ordena la lista *in place*
>>> li
[2, 5, 6, 8]
```

```
>>> li.sort(alguna_funcion)
# se ordena utilizando la función recibida
```

Listas y Tuplas

Las listas son mas lentas pero poderosas que las tuplas.

Las listas se pueden modificar y permiten mas operaciones.

Las tuplas son mas rápidas pero son inmutables.

Para convertir entre ellas utiliza las funciones list() y tuple():

```
li = list(tu)
```

```
tu = tuple(li)
```

zip y map

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

>>> lista2 = ['Ana', 'Tom', 'Zoe']

>>> list(zip(lista,lista2))
[('1', 'Ana'), ('2', 'Tom'), ('3', 'Zoe')]

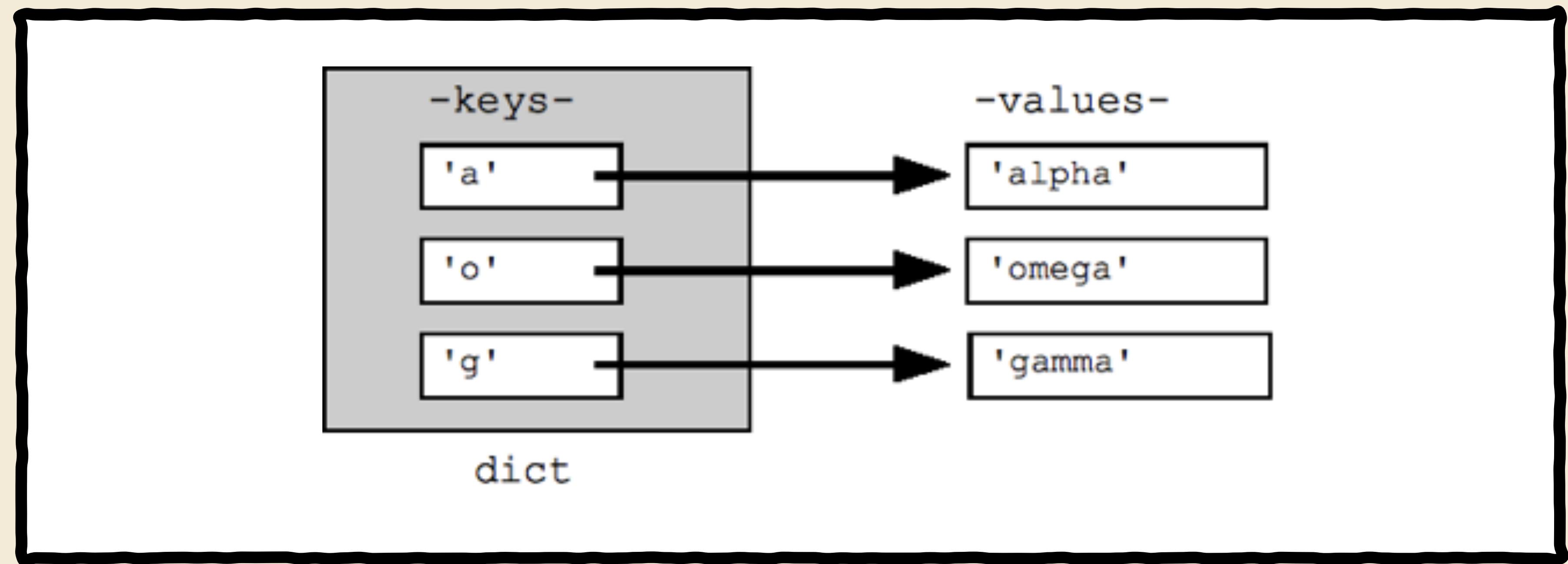
>>> map(int,lista)
[1, 2, 3]
```

DEMOCRACIA (del griego *dēmos*, pueblo, y *krateō*, gobernar) Gobierno en el que el pueblo gobierna la

Una colección de
asociaciones:

Los Diccionarios





Los diccionarios contienen un conjunto desordenado de parejas **clave:valor**.

Las claves pueden ser cualquier tipo de dato inmutable.

Los valores pueden ser de cualquier tipo.

Un diccionario puede almacenar diferentes tipos de datos.

Definir y recuperar en diccionarios

```
>>> d = { 'usuario' : 'bozo' , 'pswd' :1234 }  
  
>>> d[ 'usuario' ]  
  
'bozo'  
  
>>> d[ 'pswd' ]  
  
1234  
  
>>> d[ 'bozo' ]  
  
Traceback (innermost last):  
  File '<interactive input>' line 1, in ?  
    KeyError: bozo
```

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> tel.keys()
['guido', 'irv', 'jack']
>>> tel.has_key('guido')
True
```



Actualizando Diccionarios

```
>>> d = {'user':'bozo', 'pswd':1234}
```

```
>>> d['user'] = 'clown'
```

```
>>> d
```

```
{'user':'clown', 'pswd':1234}
```

Las claves deben de ser únicas.

Al asignar a una clave existente, remplaza el valor.

```
>>> d['id'] = 45
```

```
>>> d
```

```
{'user':'clown', 'id':45, 'pswd':1234}
```

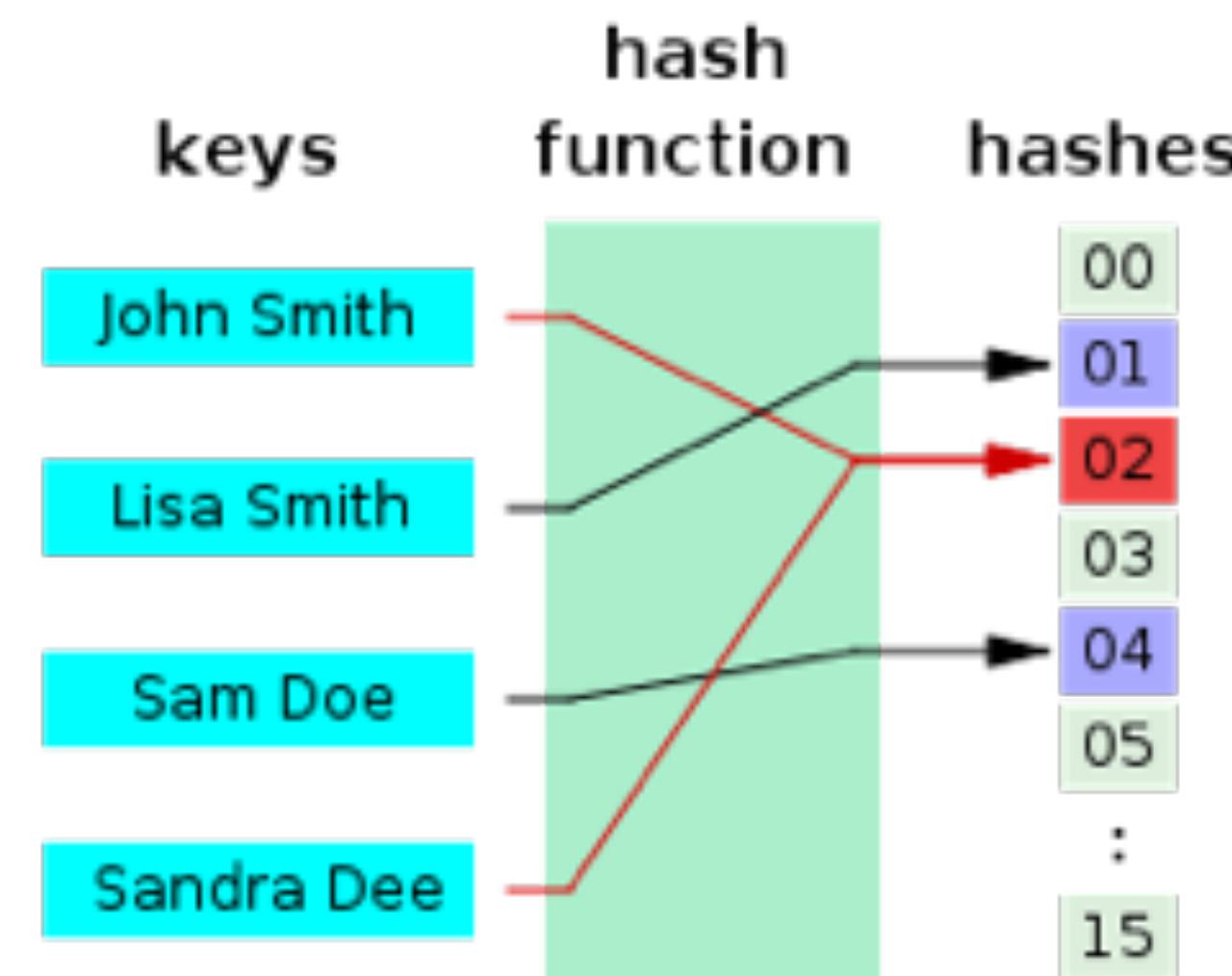
Hashing en Diccionarios

Los diccionarios no tienen orden.

Esto es porque se implementan utilizando *hashing*



hashing



Removiendo elementos de los diccionarios

```
>>> d = {'user':'bozo', 'p':1234, 'i':34}

del

>>> del d['user']          # Remueve un par clave-valor.
>>> d
{'p':1234, 'i':34}

clear()

>>> d.clear()             # Remueve todos los pares.
>>> d
{}

>>> a=[1,2]
>>> del a[1]               # (del también se usa en listas)
>>> a
[1]
```

Métodos útiles para acceder a diccionarios

```
>>> d = {'user':'bozo', 'p':1234, 'i':34}
```

keys()

```
>>> d.keys()          # Lista de claves  
['user', 'p', 'i']
```

values()

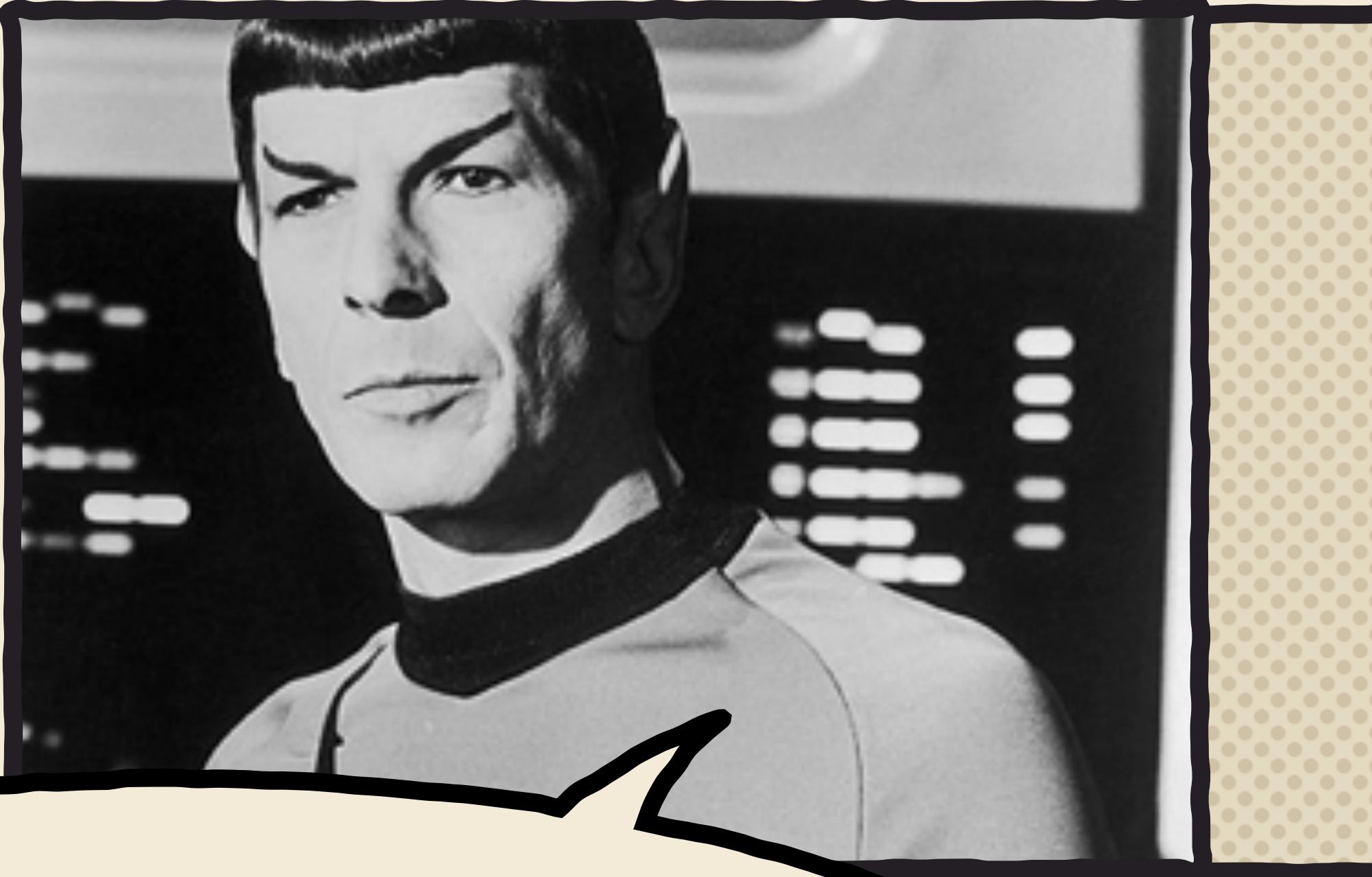
```
>>> d.values()        # Lista de valores.  
['bozo', 1234, 34]
```

items()

```
>>> d.items()         # Diccionario como una lista de tuplas.  
[('user','bozo'), ('p',1234), ('i',34)]
```

in

```
>>> if 'z' in dict: print dict['z'] # No Key Error
```



Expresiones Lógicas

True y False

True y False son constantes en Python.

Otros valores equivalentes a **True** o **False**:

False: cero, **None**, contenedores u objetos vacíos.

True: números distintos a cero, objetos no vacíos.

Operadores de comparación: ==, !=, <, <=, etc.

X y Y tienen el mismo valor: **x == y**

Si comparas **x is y**:

X y Y son dos variables que hacen referencia al mismo objeto.

Se pueden combinar expresiones booleanas.

Utilizando **and**, **or** y **not**.

Para evitar ambigüedad se necesitan paréntesis.

Propiedades de **and** y **or**

Realmente **and** y **or** no regresan **True** o **False**.

Lo que regresan es el valor de una de sus sub-expresiones
(el cual podría no ser booleano).

X **and** Y **and** Z

Si todas son verdaderas, regresa el valor de Z.

De otro modo, regresa el valor de la primera expresión falsa.

X **or** Y **or** Z

Si todas son falsas, regresa el valor de la expresión Z.

De otro modo, regresa el valor de la primera expresión verdadera.

and y **or** utilizan evaluación **lazy**, así que no se siguen
evaluando las siguientes sub-expresiones, al resolver la
expresión lógica.



Expresiones condicionales

```
x = valor_verdadero if condición else valor_falso
```

Utiliza también evaluación [Lazy](#):

Primero, se evalúa **condición**

Si regresa **True**, valor_verdadero **se evalúa y regresa**.

Si regresa **False**, valor_falso **se evalúa y regresa**.



Control de Flujo

Condiciones if

```
if x == 3:  
    print "x vale 3."  
elif x == 2:  
    print "x vale 2."  
else:  
    print "x vale otra cosa."  
print "Esto ya está fuera del 'if'."
```

Fíjate:

El uso de bloques indentados.

Dos puntos (:) de la expresión booleana.

Ciclos while

```
>>> x = 3
>>> while x < 5:
    print x, "dentro del loop"
    x = x + 1
3 dentro del loop
4 dentro del loop
>>> x = 6
>>> while x < 5:
    print x, "dentro del loop"

>>>
```

break y **continue**

Puedes utilizar la palabra reservada **break** para salir del ciclo **while** completamente.

Puedes utilizar la palabra reservada **continue** dentro de un ciclo, para detener el procesamiento de la iteración actual para ir inmediatamente a la siguiente.

ciclos for

Un ciclo for recorre cada uno de los elementos de una colección, o cualquier objeto “iterable”

```
for <elemento> in <colección>:  
    <sentencias>
```

Si <colección> es una lista o tupla, el for recorre cada elemento de la colección.

Si <colección> es una cadena, entonces el ciclo recorre cada carácter de la cadena.

```
for carácter in "Hello World":  
    print carácter
```

ciclos for

```
for <elemento> in <colección>:  
    <sentencias>
```

<elemento> puede ser complejo.

Cuando los elementos de una **<colección>** son a su vez secuencias, entonces **<elemento>** puede ser una “plantilla” de la estructura de los elementos.

Esto facilita el acceso a los elementos internos.

```
for (x, y) in [(a,1), (b,2), (c,3), (d,4)]:  
    print x
```

La función `range()`

Normalmente queremos iterar sobre una secuencia de enteros que inician en cero.

La función `range()` toma un entero como parámetro y regresa una lista de números del cero a uno antes del número que recibió.

`range(5)` regresa [0,1,2,3,4]

Para imprimir los números uno a uno:

```
for x in range(5) :  
    print x
```

Equivalente al clásico:

```
for(int i=0; i<=5; i++)
```

Diccionarios y `for`

```
>>> edades = { "Sam" :4, "Mary" :3, "Bill" :2 }
>>> edades
{'Bill': 2, 'Mary': 3, 'Sam': 4}
>>> for nombre in edades.keys():
    print nombre, edades[nombre]
```

Bill 2

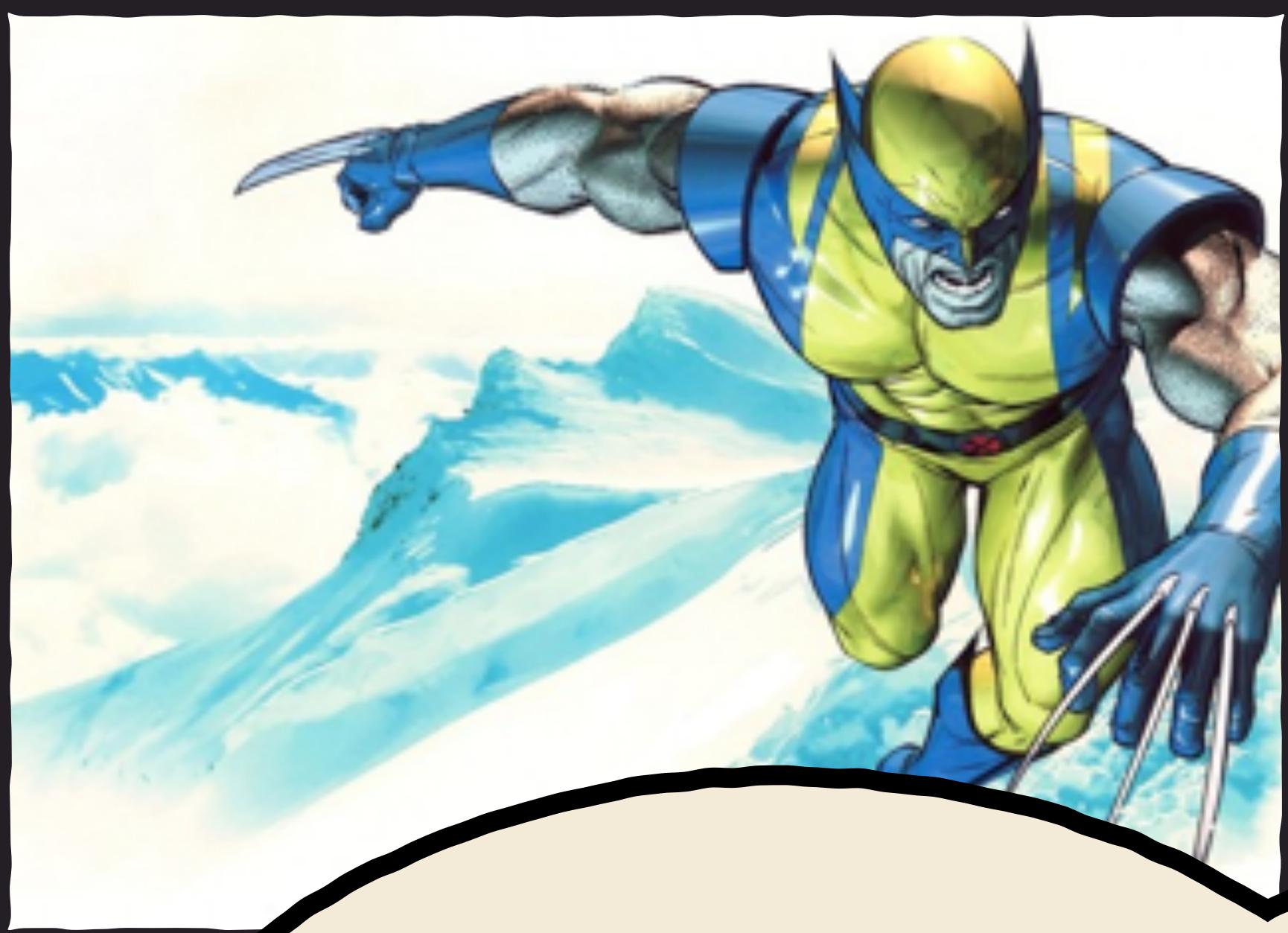
Mary 3

Sam 4

```
>>> for key in sorted(edades.keys()):
    print key, edades[key]
```

```
>>> for key in edades:
    print key, edades[key]
```

Generación de Listas por Comprensión



Listas por comprensión

Una característica poderosa del lenguaje.

Generas una nueva lista aplicando una función a cada elemento de la lista original.

[expresión for nombre in lista]

La expresión es alguna operación sobre nombre.

Cada elemento de la lista, se asigna a nombre, y se calcula el nuevo elemento utilizando la expresión.

Los elementos resultantes se van colectando en una nueva lista la cual se regresa como resultado de comprensión.

```
>>> li = [3, 6, 2, 7]
>>> [elem*2 for elem in li]
[6, 12, 4, 14]
```

Listas por comprensión

[expresión for nombre in lista]

Si la lista contiene elementos de distintos tipos, entonces la expresión debe ser capaz de operar correctamente con todos los elementos de la lista.

Si los elementos de la lista son a su vez contenedores, entonces el nombre puede consistir de patrones de nombres que “empaten” con los elementos de la lista.

```
>>> li = [ ('a', 1), ('b', 2), ('c', 7)]
>>> [ n * 3 for (x, n) in li]
[3, 6, 21]
```

Listas por comprensión

[expresión for nombre in lista]

La expresión puede contener funciones.

```
>>> def subtract(a, b):  
    return a - b  
  
>>> oplist = [(6, 3), (1, 7), (5, 5)]  
>>> [subtract(y, x) for (x, y) in oplist]  
[-3, 6, 0]
```

Listas por comprensión: Filtros

```
[ expresión for nombre in lista if filtro ]
```

La expresión booleana filtro determina si el elemento se evaluará o no por la expresión.

Si filtro es False entonces el elemento se omite de la lista antes de que se evalúe la comprensión.

Con filtros y anidadas

[expresión for nombre in lista if filtro]

```
>>> li = [3, 6, 2, 7, 1, 9]
>>> [elem * 2 for elem in li if elem > 4]
[12, 14, 18]
```

Como la operación toma una lista como entrada y produce una lista como salida, éstas se pueden anidar fácilmente:

```
>>> li = [3, 2, 4, 1]
>>> [elem*2 for elem in
      [item+1 for item in li] ]
[8, 6, 10, 4]
```

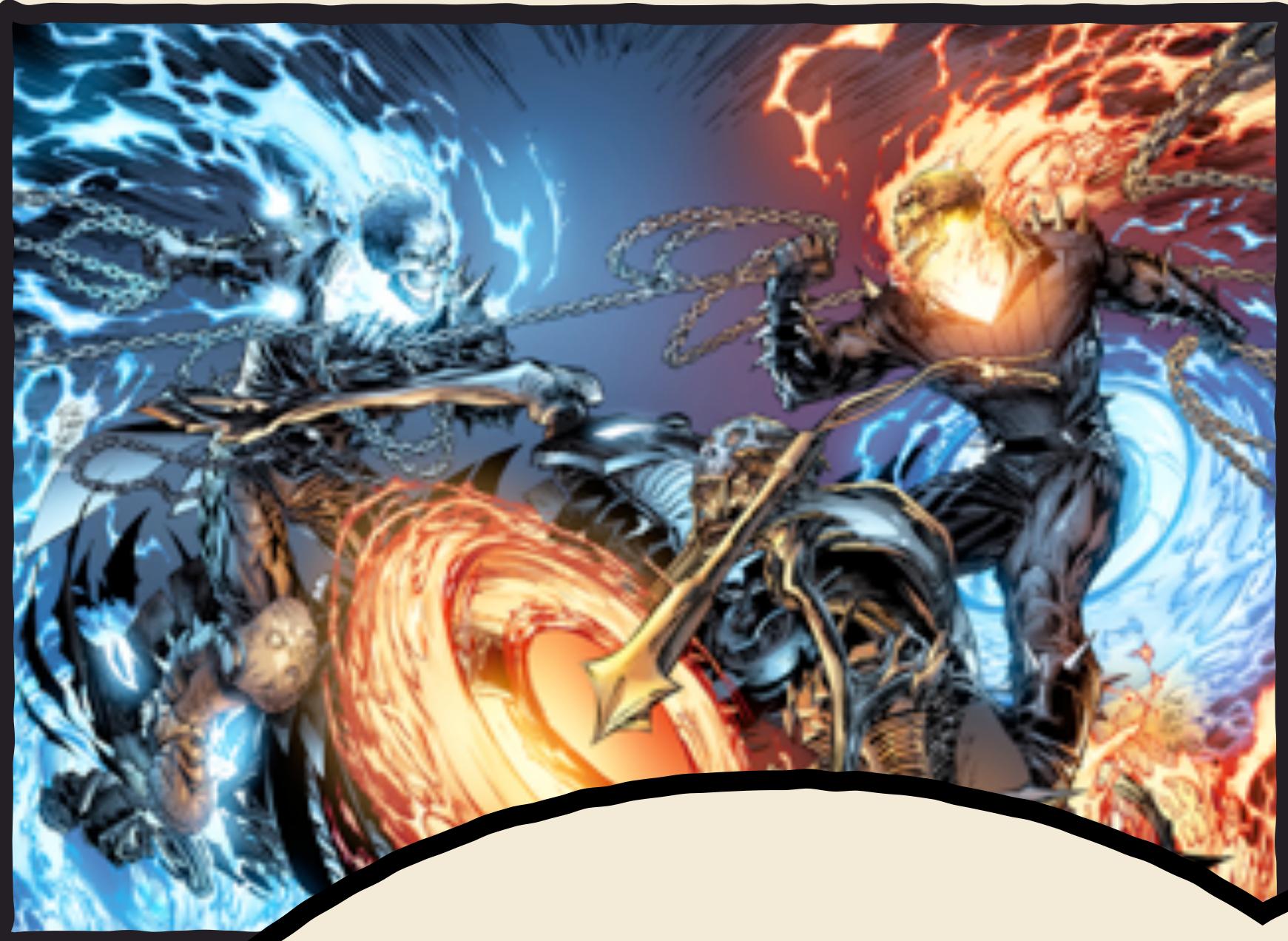


Como leer archivos y algunas otras funciones Built-In

Como leer archivos de texto

```
archivo = open('nombre_de_archivo', 'rU')
una_cadena = archivo.read()
for line in archivo:
    print line
archivo.close()
```

Operaciones con Cadenas



Operaciones con cadenas

La clase `string` tiene varios métodos que son muy útiles para dar formato a las cadenas de texto.

```
>>> "hello".upper()  
'HELLO'
```

En la documentación podrás encontrar muchas más.

Nota: usa `<string>.strip()` para eliminar los saltos de línea de los archivos de texto.

Operador % para formato de cadenas

El operador **%** te permite construir cadenas a partir de diferentes tipos de datos a manera de “llena lo espacios”.

Por ejemplo podemos indicar cuantos decimales pueden imprimirse.

Muy parecido al comando `sprintf` de C.

```
>>> x = "abc"  
>>> y = 34  
>>> "%s xyz %d" % (x, y)  
'abc xyz 34'
```

La tupla después del operador **%** se utiliza para llenar los espacios marcados por **%s** y **%d**.

Debes revisar la documentación para ver otros códigos de formato.

Imprimiendo cadenas

Puedes imprimir una cadena a la pantalla utilizando `print`.

```
>>> print "%s xyz %d" % ("abc", 34)
abc xyz 34
```

`print` automáticamente añade una nueva línea al final de la cadena.

Si incluyes una lista de objetos, los concatenará con un espacio entre ellos.

```
>>> print "abc"      >>> print "abc", "def"
abc                  abc def
```

Tip útil:

```
>>> print "abc",
```

No añade una nueva línea, sólo agrega un espacio.

De cadena a lista a cadena

join pasa una lista de cadenas a una sola cadena.

<cadena_separadora>.join(<lista>)

```
>>> ";" .join( ["abc", "def", "ghi"] )  
"abc ; def ; ghi"
```

split convierte una cadena a una lista de cadenas.

<alguna_cadena>.split(<cadena_separadora>)

```
>>> "abc ; def ; ghi" .split( ";" )  
["abc", "def", "ghi"]
```

Fíjate en el cambio de responsable de la operación

split, join y listas por comprensión

split y join se utilizan a veces en conjunto con comprensión de listas :

```
>>> " ".join( [s.capitalize() for s in "this is a test ".split(" ")] )  
'This Is A Test'
```

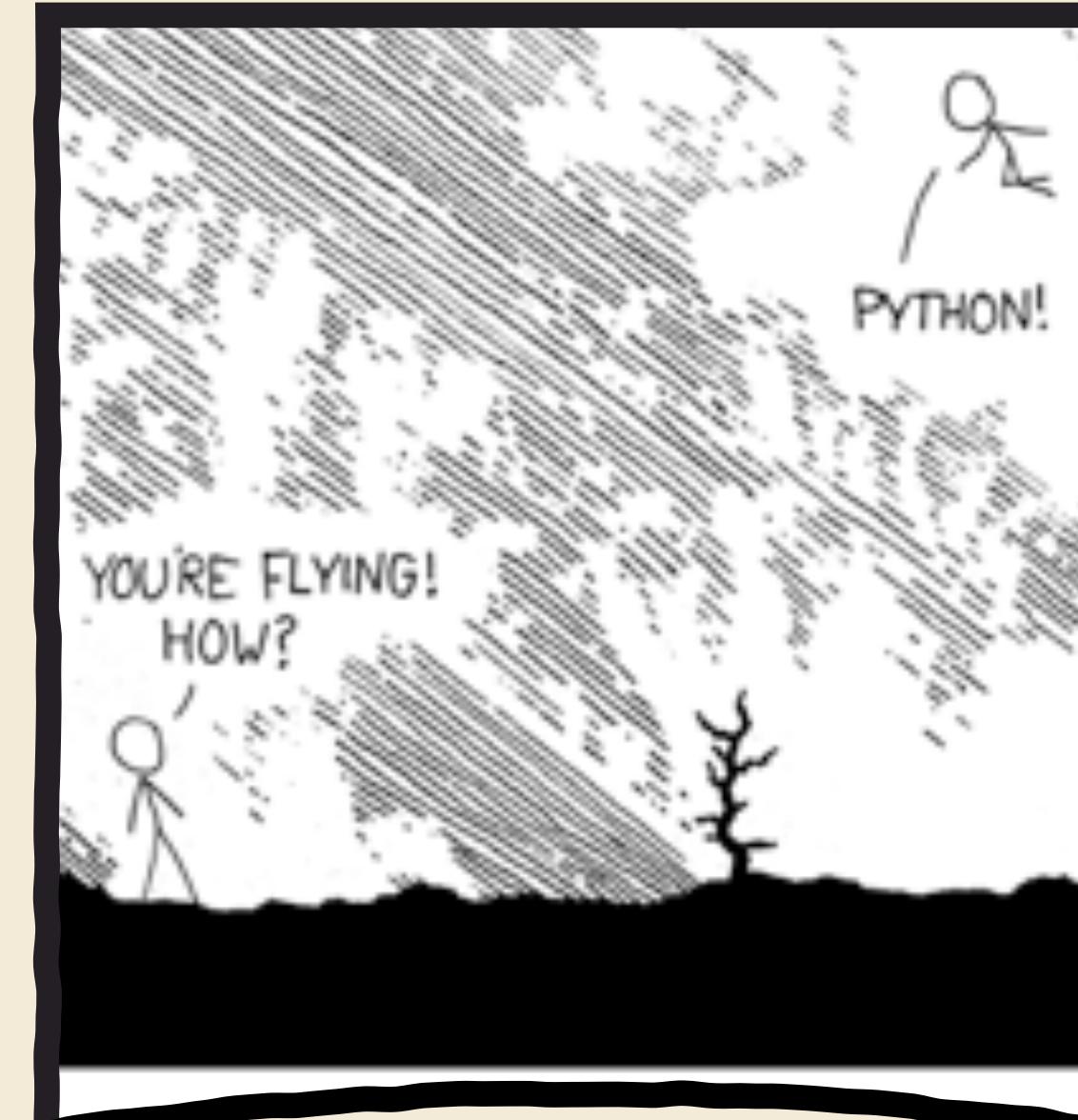
```
>>> # Por partes:  
>>> "this is a test" .split(" ")  
['this', 'is', 'a', 'test']  
>>> [s.capitalize() for s in "this is a test" .split(" ")]  
['This', 'Is', 'A', 'Test']  
>>>
```

str()

La función built-in str() puede convertir cualquier tipo de dato a una cadena.

Puedes definir como será este comportamiento para los tipos de datos definidos por el usuario, o redefinir el de muchos tipos.

```
>>> "Hello " + str(2)  
"Hello 2"
```



Otras monerías

Notación Lambda

Pueden definirse funciones anónimas.
Esto se usa cuando queremos pasar una función sencilla como argumento a otra función..

```
>>> applier(lambda z: z * 4, 7)  
28
```

El primer argumento a nuestra función `applier()` es una función anónima que regresa la entrada multiplicada por cuatro.

Nota: Al utilizar la notación lambda solamente podemos hacer funciones de una expresión.

Valores por defecto para argumentos

Puedes especificar valores por defecto, es decir valores que se tomarán en caso de que no se envíen al llamar la función.

Al incluir valores por defecto, los argumentos se hace opcionales.

```
>>> def myfun(b, c=3, d="hello") :  
        return b + c  
>>> myfun(5,3,"hello")  
>>> myfun(5,3)  
>>> myfun(5)
```

Todas las funciones anteriores regresan 8.

Cuidado con Valores por defecto

Los valores por defecto se inicializan una sola vez.

Por ejemplo:

```
def f(a, L=[ ]):
    L.append(a)
    return L

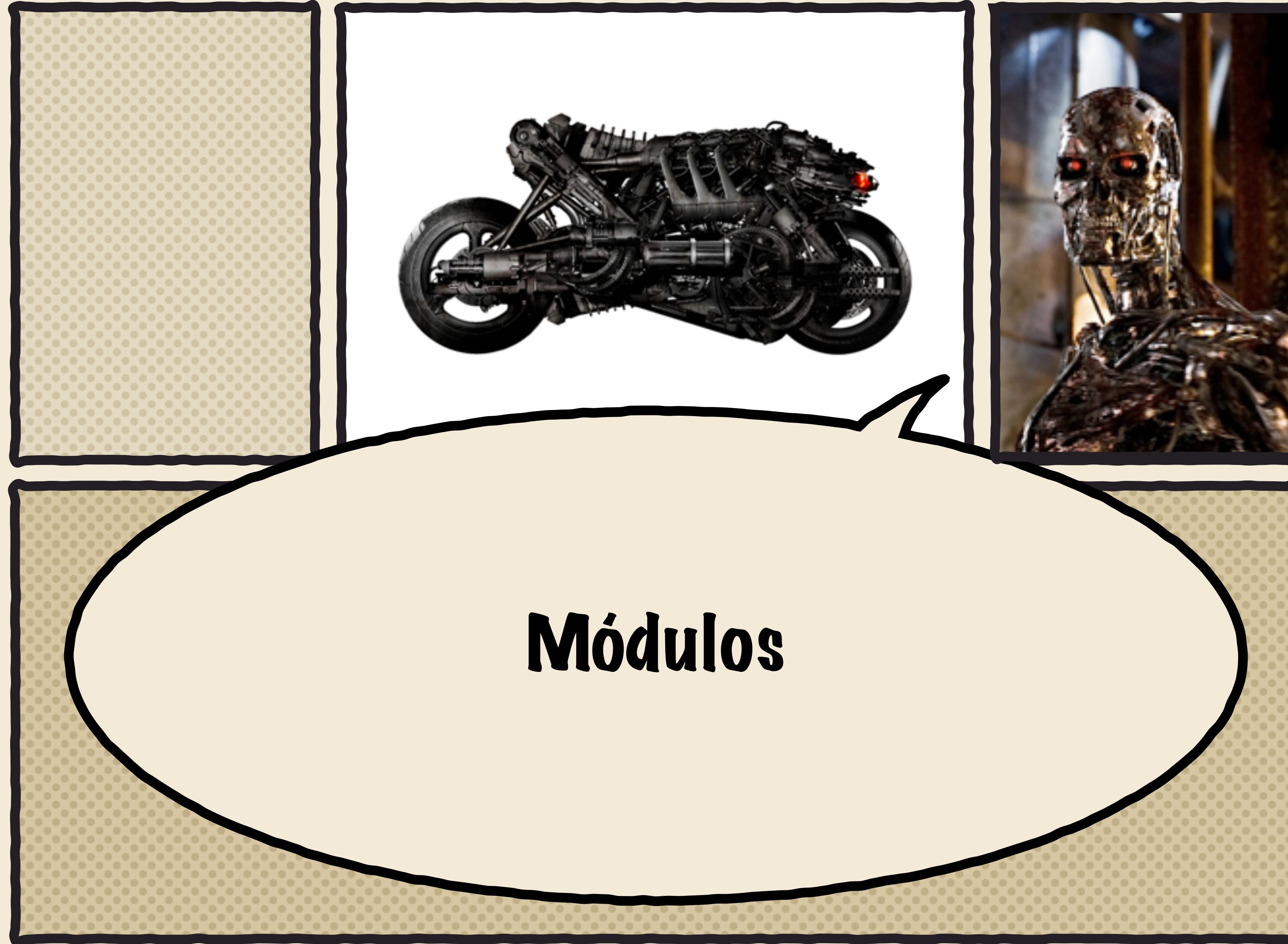
print f(1)
print f(2)
print f(3)
```

Imprimiría:

```
[1]
[1, 2]
[1, 2, 3]
```

Podemos corregirlo así:

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```



import y módulos

Sirve para utilizar clases y funciones definidas en otros archivos.

Un módulo en Python es un archivo del mismo nombre con extensión **.py**.

Como en Java **import**, C# **using** y en C++ **include**.

Tres maneras de utilizar el comando:

```
import algun_archivo  
from algun_archivo import *  
from algun_archivo import className
```

¿Cuál es la diferencia?

Que se importa del archivo y que nombre tienen las referencias después de ser importadas.

import

```
import algun_archivo
```

Todo lo que se encuentra en archivo.py es importado.

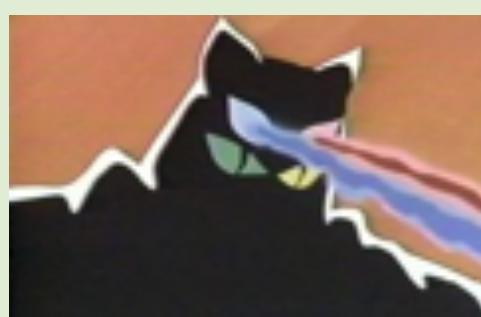
Para referirse a los elementos importados se debe agregar el nombre del módulo antes del nombre:

```
algun_archivo.className.method("abc")  
algun_archivo.myFunction(34)
```

```
from archivo import *
```

También se importa todo, pero ahora no es necesario agregar el nombre del módulo antes ya que todo se importó al espacio de nombres actual.

```
className.method("abc")  
myFunction(34)
```



¡Cuidado! Esto puede redefinir funciones o clases que se llamen igual en tu programa y en el módulo.

import

```
from algun_archivo import className
```

Solo el elemento `className` de `algun_archivo.py` es importado.

Después de importar `className`, se puede utilizar sin necesidad de agregar el prefijo del módulo, ya que se trajo al espacio de nombres actual.

```
className.method("abc")      #Esto se importó
```

```
myFunction(34)                #Esta función no
```

Podemos importar la función también:

```
from algun_archivo import className, myFunction
```

Ejecutando módulos como scripts

```
# Fibonacci numbers module

def fib(n):      # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b

def fib2(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

```
> python fibo.py <arguments>
```

Agregando:

```
if __name__ == "__main__":
```

Al final de un módulo, nos permite usarlo también como script.

El código en el bloque se ejecutará, cuando el módulo se llame desde la línea de comandos.

Nota el `import sys`

Rutas de búsqueda de módulos

Cuando importamos un modulo por ejemplo `spam`, el intérprete inicia una búsqueda:

Primero busca si hay un built-in con ese nombre.

Si no se encuentra busca un archivo llamado `spam.py` en una lista de directorios dada por la variable `sys.path`.

`sys.path` se inicializa desde con estas localidades:

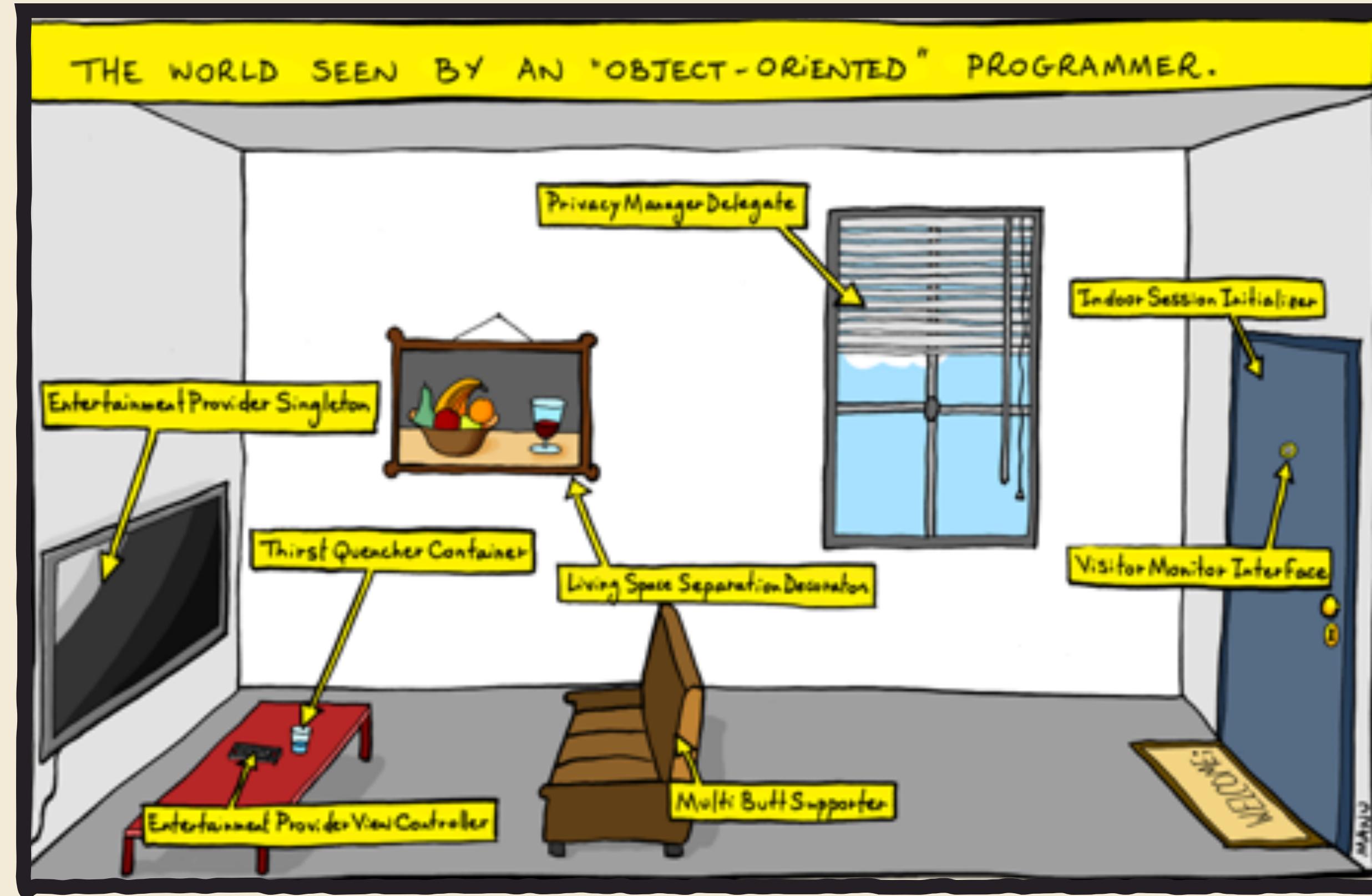
- El directorio que contiene el script de entrada (el directorio actual).
- La variable de entorno `PYTHONPATH` (una lista de directorios, utiliza la misma sintaxis que la variable `PATH`).
- Después de la inicialización, los programas de Python pueden modificar la variable `sys.path`.

El directorio actual es insertado al principio de las rutas, adelante de la ruta de librerías estándar.

Método `dir()`

```
>> import fibo, sys  
>>> dir(fibo)  
['__name__', 'fib', 'fib2']  
>>> dir(sys)  
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__stderr__',  
'__stdin__', '__stdout__', '_getframe', 'api_version', 'argv',  
'builtin_module_names', 'byteorder', 'callstats', 'copyright',  
'displayhook', 'exc_clear', 'exc_info', 'exc_type', 'excepthook',  
'exec_prefix', 'executable', 'exit', 'getdefaultencoding', 'getdlopenflags',  
'getrecursionlimit', 'getrefcount', 'hexversion', 'maxint', 'maxunicode',  
'meta_path', 'modules', 'path', 'path_hooks', 'path_importer_cache',  
'platform', 'prefix', 'ps1', 'ps2', 'setcheckinterval', 'setdlopenflags',  
'setprofile', 'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout',  
'version', 'version_info', 'warnoptions']
```

La función built-in `dir()` nos da una lista de los nombres definidos en un módulo.



Programación Orientada a Objetos

¡Son todos objetos!

Todo en Python es realmente un objeto.

Hemos visto pistas de esto antes...

```
"hello".upper()  
list3.append('a')  
dict2.keys()
```

Estos se ven como llamadas a métodos en Java o C++.

Podemos definir fácilmente nuevas clases de objetos, para complementar a los built-in.

De hecho la programación en Python, normalmente se hace con el paradigma orientado a objetos.

Definiendo una Clase

Una clase (`class`) es un tipo de dato especial que define como construir cierto tipo de objetos.

`class` también almacena ciertos datos que se comparten por todas las instancias de la clase.

`Instancias` son aquellos objetos que han sido creados siguiendo la definición especificada dentro de una clase en particular.

```
class estudiante:  
    """Representa a un estudiante."""  
    def __init__(self,n,e):  
        self.nombre = n  
        self.edad = e  
    def get_edad(self):  
        return self.edad
```

Definiendo una Clase

```
class estudiante:  
    """Representa a un estudiante."""  
    def __init__(self,n,e):  
        self.nombre = n  
        self.edad = e  
    def get_edad(self):  
        return self.edad
```

Se definen los **métodos** de una **clase** incluyendo definiciones de funciones dentro del bloque de la clase.

Los métodos para las instancias deben llevar un argumento llamado **self** el cual está atado a la instancia que ejecutará el método.

El método constructor se llama **__init__**.

Instanciando objetos

No existe la palabra reservada **new** como en otros lenguajes.

Se llama simplemente al nombre de la clase con la notación () y se asigna el resultado a un nombre.

Si se pasan argumentos, estos se pasan al método **__init__()**.

Aquí, el método **__init__()** para la clase estudiante recibe “Ana” y 21 y la nueva instancia se ata al nombre **b**:

```
b = estudiante("Ana", 21)
```

self

El primer argumento de cada método es una referencia a la instancia que lo llama.

Por convención el nombre del argumento es *self*.

En `__init__`, *self* es una referencia al objeto que está siendo creado;

***self* es similar a la palabra reservada *this* en Java o C++.**

self

Cuando defines un método debes especificar el argumento *self* explícitamente.

No así cuando llamas al método.

Python lo pasa por ti automáticamente.

Al definir el método:
(este código está dentro del
bloque de la clase.)

```
def set_edad(self, num):  
    self.edad = num
```

Al llamar al método:

```
>>> x.set_edad(23)
```

Borrado de Instancias

Cuando termines de utilizar un objeto, no es necesario que lo borres explícitamente.

Python recolección automática de basura.

Python automáticamente detecta cuando las referencias a un espacio de memoria han salido del ámbito y libera esa memoria.

Normalmente funciona muy bien, muy pocas fugas de memoria.

No hay métodos destructores.

La clase estudiante

```
class estudiante:  
    """Representa a un estudiante."""  
    def __init__(self,n,e):  
        self.nombre = n  
        self.edad = e  
    def get_edad(self):  
        return self.edad
```

Sintaxis tradicional de acceso

```
>>> f = estudiante("Bob Smith", 23)

>>> f.nombre      # Access an attribute.
"Bob Smith"

>>> f.get_edad()      # Access a method.
23
```

Acceso a miembros desconocidos

Problema: En ocasiones el nombre de un atributo no se conoce si no hasta el tiempo de ejecución.

Solución: `getattr(object_instance, string)`

`object_instance` es la referencia al objeto que accederemos.

`string` es una cadena que contiene el nombre del método o atributo de la instancia.

`getattr(object_instance, string)` regresa una referencia al método o atributo.

Acceso a miembros desconocidos

```
>>> f = estudiante("Bob Smith", 23)

>>> getattr(f, "nombre")
"Bob Smith"

>>> getattr(f, "get_edad")
<method get_age of class studentClass at 010B3C2>

>>> getattr(f, "get_edad")()      # llamamos a la referencia.
23

>>> getattr(f, "get_birthday") # no existe el método
```

hasattr(object_instance,string)

```
>>> f = estudiante("Bob Smith", 23)

>>> hasattr(f, "nombre")
True

>>> hasattr(f, "get_edad")
True

>>> hasattr(f, "get_birthday")
False
```

Dos tipos de atributos

Las clases tiene métodos y atributos; Los atributos son datos con los que representamos las propiedades de los objetos.

Atributos tipo dato

Son variables que pertenecen a cierta **instancia particular** de la clase.
Cada instancia tiene su propio valor.
Son los atributos más comunes.

Atributos de la clase

Pertenece a toda la **clase**.

Todas las instancias de la clase comparten el mismo valor.

Son las propiedades **static** en otros lenguajes.

Sirven para

constantes para todas las instancias.

contadores de cuantas instancias se han creado.

Atributos tipo dato

Los atributos tipo dato son creados e inicializados por el método `__init__()`.

Los atributos se crean al asignarles referencias a los nombres.

Dentro de la clase los atributos tipo dato se refieren con `self` por ejemplo, `self.nombre`

```
class profesor:  
    "Representa a los profesores"  
    def __init__(self,n):  
        self.nombre = n  
    def print_nombre(self):  
        print self.nombre
```

Atributos de la clase

Como todas las instancias de la clase, comparten la misma referencia:

cuando *cualquier* instancia cambia el valor, este cambia para *todas* las instancias.

Los Atributos de la clase se definen:

dentro de la definición de la clase.
fuerá de cualquier método.

Como solo hay uno de estos atributos por cada clase y no por instancia, estos se accesan utilizando una notacion diferente:

Se accesa a los atributos de la clase de la siguiente manera:

`self.__class__.name`

```
class sample:  
    x = 23  
    def increment(self):  
        self.__class__.x += 1
```

```
>>> a = sample()  
>>> a.increment()  
>>> a.__class__.x  
24
```

Atributos de datos y de la clase

```
class counter:  
    overall_total = 0  
        # class attribute  
    def __init__(self):  
        self.my_total = 0  
            # data attribute  
    def increment(self):  
        counter.overall_total = \  
        counter.overall_total + 1  
        self.my_total = \  
        self.my_total + 1
```

```
>>> a = counter()  
>>> b = counter()  
>>> a.increment()  
>>> b.increment()  
>>> b.increment()  
>>> a.my_total  
1  
>>> a.__class__.overall_total  
3  
>>> b.my_total  
2  
>>> b.__class__.overall_total  
3
```

Herencia

Una clase puede extender la definición de otra clase.

Permite el uso o extensión de métodos y atributos que ya han sido definidos en la clase base.

Nueva clase : *subclase*. Original: *padre*, *clase base* o *superclase*

Para definir una subclase, se indica con paréntesis el nombre de la superclase.

`class estudiante_ia(estudiante) :`

Python permite herencia múltiple.

Redefinición de Métodos

Para *redefinir un método* de la clase padre, solo se incluye una nueva definición en la subclase utilizando el mismo nombre.

El código anterior no se ejecutará.

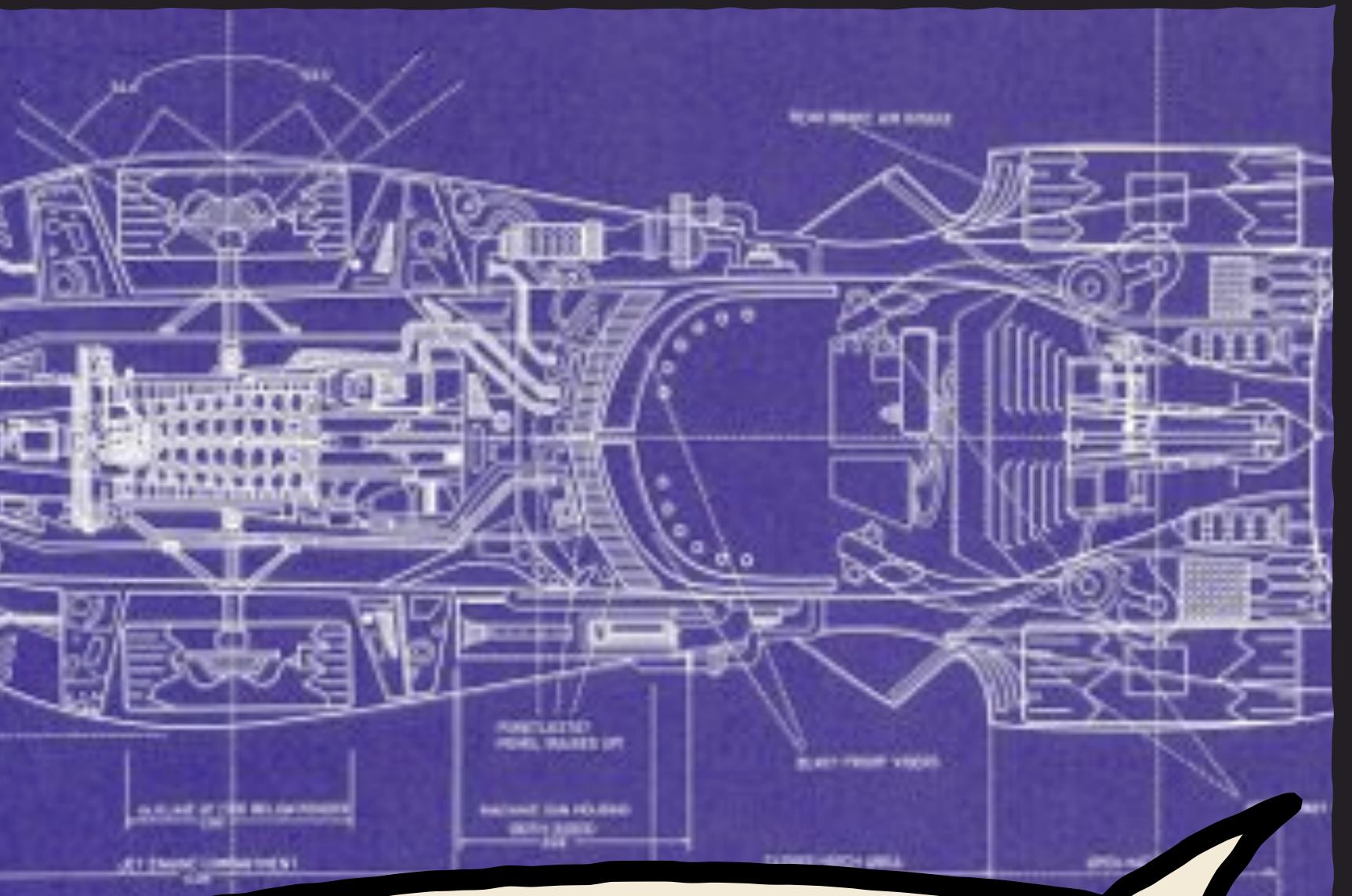
Para ejecutar el método de la clase padre *junto con* la nueva definición se debe llamar explícitamente la versión del padre.

```
clasePadre.metodoPadre(self, a, b, c)
```

En este caso si se debe agregar el argumento **self** explícitamente.

Ejemplo Herencia

```
class estudiante:  
    "Clase estudiante."  
  
    def __init__(self,n,e):  
        self.nombre = n  
        self.edad = e  
  
    def get_edad(self):  
        return self.edad  
  
class estudiante_ia(estudiante):  
    "Estudiante de IA."  
  
    def __init__(self,n,e,esp):  
        #Se llama al __init__ de la clase padre  
        estudiante.__init__(self,n,a)  
        self.especialidad = esp  
  
    def get_edad():    #Se redefine el método completamente  
        print "Age: " + str(self.edad)
```



Métodos y Atributos Built-in

Métodos Built-in de las clases

Las clases contienen muchos métodos y atributos que están incluidos por Python aunque uno no los haya definido explícitamente.

La mayoría de estos métodos definen funcionalidad automática que se dispara por operadores especiales o el uso de la clase.

Los atributos built-in definen información que debe ser almacenada para todas las clases.

Todos los miembros built-in tienen subguiones dobles encerrando a sus nombres por ejemplo:

`__init__` `__doc__`

Métodos Built-in de las clases

Por ejemplo el método `__repr__` existe para todas las clases, y este puede ser redifinido.

Este método especifica como se convierte una instancia de la clase a una cadena.

`print f` llamaría a `f.__repr__()` para producir una cadena que represente al objeto `f`.

Si tecleas `f` en el prompt y presionas ENTER, esto llamaría también al método `__repr__` para determinar lo que se mostraría al usuario. Si no se redefine sería el nombre de la clase a la que pertenece `f`.

Métodos Built-in de las clases: Ejemplo

```
class student:  
    ...  
    def __repr__(self):  
        return "I'm named " + self.full_name  
    ...  
  
>>> f = student("Bob Smith", 23)  
>>> print f  
I'm named Bob Smith  
>>> f  
"I'm named Bob Smith"
```

Métodos Built-in de las clases

Más ejemplos de métodos que se pueden redefinir:

- `__init__` : Constructor de la clase.
- `__cmp__` : Define como funciona `==` para la clase.
- `__len__` : Define como funciona `len(obj)`.
- `__copy__` : Define como se copian las instancias.

Otros métodos built-in te permiten utilizar la notación de corchetes [] como los arreglos o paréntesis como () el llamado de una función .

Atributos Built-in

Estos atributos existen para todas las clases:

- `__doc__`** : Variable que almacena la cadena de documentación de la clase.
- `__class__`** : Variable que tiene una referencia a la clase desde cualquier instancia de ella.
- `__module__`** : Variable que tiene una referencia al modulo en el que se define determinada clase.
- `__dict__`** : El diccionario que es realmente el espacio de nombres para la clase (pero no sus superclases).

Método built-in muy útil en estos casos:

`dir(x)` regresa una lista de todos los métodos y atributos definidos para el objeto x.

Atributos Built-in: Ejemplo

```
>>> f = student("Bob Smith", 23)

>>> print f.__doc__
A class representing a student.

>>> f.__class__
< class studentClass at 010B4C6 >

>>> g = f.__class__("Tom Jones", 34)
```

Datos privados

Todo atributo que tiene dos subguiones antes del nombre (pero ninguno al final) se considera privado.

Nota:

Doble subguión al incio y al final está reservado para métodos y atributos built-in de la clase.

Note:

No hay una especificación ‘protected’ en Python; así que las subclases no podrían acceder a estos miembros tampoco.

Esta presentación se basa en tres archivos disponibles en línea para el curso: **CIS 391 Introduction to Artificial Intelligence Fall 2008 (Slides)** de la universidad de Penn State ya no están en el URL original.

La documentación oficial de python.org.

Google's Python class:

<http://code.google.com/intl/es-419/edu/languages/google-python-class/index.html>

Otros:

Distribución de Python usada en clase:

<http://www.enthought.com/>

IDE:

<http://www.jetbrains.com/pycharm/>

La plantilla comic standard disponible gratuitamente en:

<http://www.keynotezone.com/themes/comics/index.html>

Imágenes de encontradas con Google, sus URLs están en las páginas individuales. La mayoría Creative Commons.

Correcciones y sugerencias a: mariosky@gmail.com

Disponible como archivo keynote para editar, en github.com/mariosky/presentaciones