Introducción a Python

Grupo de Sistemas y Comunicaciones

gsyc-profes@gsyc.escet.urjc.es

Octubre 2002

Primeros pasos

Características

Python es un lenguaje:

- de alto nivel
- interpretado
- orientado a objetos (todo son objetos)
- dinámicamente tipado (frente a estáticamente tipado)
- fuertemente tipado (frente a débilmente tipado)
- sensible a mayúsculas/minúsculas

Un programa en Python

```
def buildConnectionString(params):
    """Build a connection string from a dictionary of parameters.

    Returns string."""
    return ";".join(["%s=%s" % (k, v) for k, v in params.items()])

if __name__ == "__main__":
    myParams = {"server":"mpilgrim", \
                "database":"master", \
                "uid":"sa", \
                "pwd":"secret" \
                }
    print buildConnectionString(myParams)
```

Ejecución:

```
xterm$ python odbchelper.py
server=mpilgrim;uid=sa;database=master;pwd=secret
```

Cadenas de documentación

- No son obligatorias pero sí muy recomendables (varias herramientas hacen uso de ellas).
- La cadena de documentación de un objeto es su atributo `__doc__`
- En una sola línea para objetos sencillos, en varias para el resto de los casos.
- Entre triples comillas-dobles (incluso si ocupan una línea).
- Si hay varias líneas:
  - La primera línea debe ser una resumen breve del propósito del objeto. Debe empezar con mayúscula y acabar con un punto
  - Una línea en blanco debe separar la primera línea del resto
  - Las siguientes líneas deberían empezar justo debajo de la primera comilla doble de la primera línea

De una sola línea:

```
def kos_root():
    """Return the pathname of the KOS root directory."""
    global _kos_root
    ...
```

De varias:

```
def complex(real=0.0, imag=0.0):
    """Form a complex number.

    Keyword arguments:
    real -- the real part (default 0.0)
    imag -- the imaginary part (default 0.0)

    """
    if imag == 0.0 and real == 0.0: return complex_zero
```

Objetos en Python

Todo son objetos, en sentido amplio:

- Cualquier objeto puede ser asignado a una variable o pasado como parámetro a una función
- Algunos objetos pueden no tener ni atributos ni métodos
- Algunos objetos pueden no permitir que se herede de ellos

Ejemplos de objetos Python: Strings, listas, funciones, módulos...

Todos los objetos tienen:

- Identidad:
  - Nunca cambia.
  - El operador `is` compara la identidad de dos objetos.
  - La función `id()` devuelve una representación de la identidad (actualmente, su dirección de memoria).
- Tipo:
  - Nunca cambia.
  - La función `type()` devuelve el tipo de un objeto (que es otro objeto)
- Valor:
  - Objetos inmutables: su valor no puede cambiar
  - Objetos mutables: su valor puede cambiar

Contenedores: objetos que contienen referencias a otros objetos (ej.: tuplas, listas, diccionarios).

Definición de variables

En Python no hay declaración explícita de variables:

- Las variables "nacen" cuando se les asigna un valor
- Las variables "desaparecen" cuando se sale de su ámbito

Pero Python no permite referenciar variables a las que nunca se ha asignado un valor.

Sangrado y separadores de sentencias

- ¡En Python NO hay llaves ni `begin-end` para encerrar bloques de código! Un mayor nivel de sangrado indica que comienza un bloque, y un menor nivel indica que termina un bloque.
- Las sentencias se terminan al acabarse la línea (salvo casos especiales donde la sentencia queda "abierta": en mitad de expresiones entre paréntesis, corchetes o llaves).
- El carácter \ se utiliza para extender una sentencia más allá de una linea, en los casos en que no queda "abierta".
- El carácter : se utiliza como separador en sentencias compuestas. Ej.: para separar la definición de una función de su código.
- El carácter ; se utiliza como separador de sentencias escritas en la misma línea.

El atributo __name__ de un módulo

Los módulos son objetos, con ciertos atributos predefinidos.

El atributo `__name__`:

- si el módulo es importado (con `import`), contiene el nombre del fichero, sin trayecto ni extensión
- si el módulo es un programa que se ejecuta sólo, contiene el valor `__main__`

Puede escribirse ejecución condicionada a cómo se use el módulo:

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


Importar módulos

- `import nombre-módulo`
  permite acceder a los símbolos del módulo con la sintaxis `nombre-módulo.X`
- `from nombre-módulo import a, b, c`
  incorpora los símbolos a, b, c al espacio de nombres, siendo accesibles directamente (sin cualificarlos con el nombre del módulo)
- `from nombre-módulo import *`
  incorpora los símbolos del módulo al espacio de nombres, siendo accesibles directamente (sin cualificarlos con el nombre del módulo).

Diccionarios

- Tipo de datos predefinido en Python, equivalente al "hash" de Perl.
- Es un conjunto desordenado de elementos que se escriben entre llaves.
- Cada elemento del diccionario es un par clave-valor.
- Se pueden obtener valores a partir de la clave, pero no al revés.

```
>>> d = {"server":"mpilgrim", "database":"master"}
>>> d
{'server': 'mpilgrim', 'database': 'master'}
>>> d["server"]
'mpilgrim'
>>> d["mpilgrim"]
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
KeyError: mpilgrim
```


- Asignar valor a una clave existente reemplaza el antiguo
- Una clave de tipo cadena es sensible a mayúsculas/minúsculas
- Pueden añadirse entradas nuevas al diccionario
- Los diccionarios se mantienen desordenados

```
>>> d["database"] = "pubs"
>>> d
{'server': 'mpilgrim', 'database': 'pubs'}
>>> d["uid"] = "sa"
>>> d
{'server': 'mpilgrim', 'uid': 'sa', 'database': 'pubs'}
```

- Los valores de un diccionario pueden ser de cualquier tipo
- Las claves pueden ser enteros, cadenas y algún otro tipo
- Pueden borrarse un elemento del diccionario con `del`
- Pueden borrarse todos los elementos del diccionario con `clear()`

```
>>> d["retrycount"] = 3
>>> d[42] = "douglas"
>>> del d["uid"]
>>> d
{'server': 'mpilgrim', 42: 'douglas', 'database': 'master', 'retrycount': 3}
>>> d.clear()
>>> d
{}
```


Otras operaciones con diccionarios:

- `len(d)` devuelve el número de elementos de `d`
- `d.has_key(k)` devuelve 1 si existe la clave `k` en `d`, 0 en caso contrario
- `k in d` equivale a: `d.has_key(k)`
- `d.items()` devuelve la lista de elementos de `d`
- `d.keys()` devuelve la lista de claves de `d`
- `d1.update(d2)` equivale a: `for k in d2.keys(): d1[k] = d2[k]`
- `d.get(k,v)` devuelve el valor de clave `k` si existe, `v` en caso contrario

Listas

- Tipo de datos predefinido en Python, va mucho más allá de los arrays
- Es un conjunto indexado de elementos que se escriben entre corchetes
- Cada elemento se separa del anterior por un carácter ,
- El primer elemento tiene índice 0.
- Un índice negativo accede a los elementos empezando por el final de la lista. El último elemento tiene índice -1.

```
>>> li = ["a", "b", "mpilgrim", "z", "example"]
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li[0]
'a'
>>> li[4]
'example'
>>> li[-1]
'example'
```


- Pueden referirse rodajas (*slices*) de listas escribiendo dos índices entre el carácter :
- La rodaja va desde el primero, incluido, al último, excluido.
- Si no aparece el primero, se entiende que empieza en el primer elemento (0)
- Si no aparece el segundo, se entiende que termina en el último elemento (incluido).

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li[:3]
['a', 'b', 'mpilgrim']
>>> li[3:]
['z', 'example']
>>> li[:]
['a', 'b', 'mpilgrim', 'z', 'example']
```

- `append()` añade un elemento al final de la lista
- `insert()` inserta un elemento en la posición indicada

```
>>> li
['a', 'b', 'mpilgrim', 'z', 'example']
>>> li.append("new")
>>> li
['a', 'b', 'mpilgrim', 'z', 'example', 'new']
>>> li.insert(2, "new")
>>> li
['a', 'b', 'new', 'mpilgrim', 'z', 'example', 'new']
```


- `index()` busca en la lista un elemento y devuelve el índice de la primera aparición del elemento en la lista. Si no aparece se eleva una excepción.
- El operador `in` devuelve 1 si un elemento aparece en la lista, y 0 en caso contrario.

```
>>> li
['a', 'b', 'new', 'mpilgrim', 'z', 'example', 'new']
>>> li.index("example")
5
>>> li.index("new")
2
>>> li.index("c")
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: list.index(x): x not in list
>>> "c" in li
0
```

- `remove()` elimina la primera aparición de un elemento en la lista. Si no aparece, eleva una excepción.
- `pop()` devuelve el último elemento de la lista, y lo elimina.

```
 >>> li
['a', 'b', 'new', 'mpilgrim', 'z', 'example', 'new', 'two', 'elements']
>>> li.remove("new")
>>> li
['a', 'b', 'mpilgrim', 'z', 'example', 'new', 'two', 'elements']
>>> li.remove("c")
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
ValueError: list.remove(x): x not in list
>>> li.pop()
'elements'
>>> li
['a', 'b', 'mpilgrim', 'z', 'example', 'new', 'two']
```


- El operador + concatena dos listas, devolviendo una nueva lista
- El operador * concatena repetitivamente una lista a sí misma

```
>>> li = ['a', 'b', 'mpilgrim']
>>> li = li + ['example', 'new']
>>> li
['a', 'b', 'mpilgrim', 'example', 'new']
>>> li += ['two']
>>> li
['a', 'b', 'mpilgrim', 'example', 'new', 'two']
>>> li = [1, 2] * 3
>>> li
[1, 2, 1, 2, 1, 2]
```

- `sort()` ordena una lista. Puede recibir opcionalmente un argumento especificando una función de comparación, lo que enlentece notable su funcionamiento
- `reverse()` invierte las posiciones de los elementos en una lista.

Ninguno de estos métodos devuelve nada, simplemente alteran la lista sobre la que se aplican.

```
>>> li = ['a', 'b', 'new', 'mpilgrim', 'z', 'example', 'new', 'two', 'elements']
>>> li.sort()
>>> li
['a', 'b', 'elements', 'example', 'mpilgrim', 'new', 'new', 'two', 'z']
>>> li.reverse()
>>> li
['z', 'two', 'new', 'new', 'mpilgrim', 'example', 'elements', 'b', 'a']
```


Tuplas

Tipo predefinido de Python para una lista inmutable.

Se define de la misma manera, pero con los elementos entre paréntesis.

Las tuplas no tienen métodos: no se pueden añadir elementos, ni cambiarlos, ni buscar con `index()`.

Sí puede comprobarse la existencia con el operador `in`.

```
>>> t = ("a", "b", "mpilgrim", "z", "example")
>>> t[0]
'a'
>>> 'a' in t
1
>>> t[0] =  "b"
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
```

Utilidad de las tuplas:

- Son más rápidas que las listas
- Pueden ser una clave de un diccionario (no así las listas)
- Se usan en el formateo de cadenas

`tuple(li)` devuelve una tupla con los elementos de la lista `li`

`list(t)` devuelve una lista con los elementos de la tupla `t`

Asignaciones múltiples y rangos

- Pueden hacerse también tuplas de variables:

```
>>> v = ('a', 'b', 'e')
>>> (x, y, z) = v
>>> x
'a'
```

- La función `range()` permite generar listas al vuelo:

```
>>> range(7)
[0, 1, 2, 3, 4, 5, 6]
>>> (MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
... FRIDAY, SATURDAY, SUNDAY) = range(7)
>>> MONDAY
0
>>> SUNDAY
6
```

Mapeo de listas

- Se puede mapear una lista en otra, aplicando una función a cada elemento de la lista:

```
>>> li = [1, 9, 8, 4]
>>> [elem*2 for elem in li]
[2, 18, 16, 8]
>>> li
[1, 9, 8, 4]
>>> li = [elem*2 for elem in li]
>>> li
[2, 18, 16, 8]
```


Filtrado de listas

- Sintaxis:

```
[expresión-mapeo for elemento in lista-origen if condición-filtrado]
```

- Ejemplos:

```
>>> li = ["a", "mpilgrim", "foo", "b", "c", "b", "d", "d"]
>>> [elem for elem in li if len(elem) > 1]       1
['mpilgrim', 'foo']
```

Cadenas de caracteres

- Las cadenas pueden encerrarse entre comillas simples (') o dobles (")
- Si una cadena se expande más de una línea, puede usarse el carácter \, o encerrarse entre una triple comilla simple (''') o doble (""")
- La triple comilla interpreta el formateo tal cual aparece en el código, incluyendo saltos de línea. En el resto de casos hay que usar \n
- El operador + concatena cadenas, y el * las repite un número entero de veces
- Se puede acceder a los caracteres de cadenas mediante índices y rodajas como en las listas


```
>>> 'doesn\'t'
"doesn't"
>>> "doesn't"
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> word = 'Help' + 'A'
>>> word
'HelpA'
>>> '<' + word*5 + '>'
'<HelpAHelpAHelpAHelpAHelpA>'
>>> word[4]
'A'
>>> word[0:2]
'He'
>>> word[-2:]
'pA'
```

- El operador % permite hacer formateo de cadenas (al estilo de `sprintf` en C) apoyándose en tuplas:

```
>>> uid = "sa"
>>> pwd = "secret"
>>> print pwd + " is not a good password for " + uid
secret is not a good password for sa
>>> print "%s is not a good password for %s" % (pwd, uid)
secret is not a good password for sa
>>> userCount = 6
>>> print "Users connected: %d" % (userCount, )
Users connected: 6
>>> print "Users connected: " + userCount
Traceback (innermost last):
  File "<interactive input>", line 1, in ?
TypeError: cannot add type "int" to string
```


`join()` devuelve una cadena que engloba a todos los elementos de la lista.

`split()` devuelve una lista dividiendo una cadena. Volviendo al ejemplo inicial:

```
>>> params = {"server":"mpilgrim", "database":"master",
... "uid":"sa", "pwd":"secret"}
>>> ["%s=%s" % (k, v) for k, v in params.items()]
['server=mpilgrim', 'uid=sa', 'database=master', 'pwd=secret']
>>> s = ";".join(["%s=%s" % (k, v) for k, v in params.items()])
>>> s
'server=mpilgrim;uid=sa;database=master;pwd=secret'
>>> s.split(";")
['server=mpilgrim', 'uid=sa', 'database=master', 'pwd=secret']
```

Completando sintaxis

Sentencia `if`:

```
>>> x = int(raw_input("Please enter an integer: "))
>>> if x < 0:
...      x = 0
...      print 'Negative changed to zero'
... elif x == 0:
...      print 'Zero'
... elif x == 1:
...      print 'Single'
... else:
...      print 'More'
```

Nótese como el carácter : introduce cada bloque de sentencias.

Sentencia `for`:

```
>>> a = ['cat', 'window', 'defenestrate']
>>> for x in a:
...     print x, len(x)
...
cat 3
window 6
defenestrate 12
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print i, a[i]
...
0 Mary
1 had
2 a
3 little
4 lamb
```

Sentencia `while`:

```
>>> a, b = 0, 1
>>> while b < 1000:
...     print b,
...     a, b = b, a+b
...
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

Nótese el efecto de un carácter `,` al final de un `print`

Nótese otro modelo de asignación múltiple

Sentencia nula: `pass`

Valor nulo: `None`

Introspección

Más código a estudiar

```
def help(object, spacing=10, collapse=1):
    """Print methods and doc strings.

    Takes module, class, list, dictionary, or string."""
    methodList = [method for method in dir(object)
                  if callable(getattr(object, method))]
    processFunc = collapse and (lambda s: " ".join(s.split())) or (lambda s: s)
    print "\n".join(["%s %s" %
                     (method.ljust(spacing),
                      processFunc(str(getattr(object, method).__doc__)))
                    for method in methodList])

if __name__ == "__main__":
    print help.__doc__
```

Ejemplo de uso:

```
>>> from apihelper import help
>>> li = []
>>> help(li)
append     L.append(object) -- append object to end
count      L.count(value) -> integer -- return number of occurrences of value
extend     L.extend(list) -- extend list by appending list elements
index      L.index(value) -> integer -- return index of first occurrence of value
insert     L.insert(index, object) -- insert object before index
pop        L.pop([index]) -> item -- remove and return item at index (default last)
remove     L.remove(value) -- remove first occurrence of value
reverse    L.reverse() -- reverse *IN PLACE*
sort       L.sort([cmpfunc]) -- sort *IN PLACE*; if given, cmpfunc(x, y) -> -1, 0, 1
```


Parámetros nombrados y opcionales

- Los parámetros con valores por defecto son opcionales
- En la llamada, pueden especificarse los parámetros por nombre

```
def help(object, spacing=10, collapse=1):
```

Llamadas válidas:

```
help(odbchelper)
help(odbchelper, 12)
help(odbchelper, collapse=0)
help(spacing=15, object=odbchelper)
```

Funciones predefinidas para introspección

Todas estas funciones están en el módulo `__builtin__`. Python siempre tiene implícito un `from __builtin__ import *`

- `type()` devuelve el tipo de su argumento

```
>>> type(1)
<type 'int'>
>>> li = []
>>> type(li)
<type 'list'>
>>> import odbchelper
>>> type(odbchelper)
<type 'module'>
>>> import types
>>> type(odbchelper) == types.ModuleType
1
```


- `str()` convierte su argumento en una cadena

```
>>> str(1)
'1'
>>> horsemen = ['war', 'pestilence', 'famine']
>>> horsemen.append('Powerbuilder')
>>> str(horsemen)
"['war', 'pestilence', 'famine', 'Powerbuilder']"
>>> str(odbchelper)
"<module 'odbchelper' from 'c:\\docbook\\dip\\py\\odbchelper.py'>"
>>> str(None)
'None'
```

- `dir()` devuelve la lista de atributos y métodos de su argumento:

```
>>> li = []
>>> dir(li)          1
['append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
>>> d = {}
>>> dir(d)           2
['clear', 'copy', 'get', 'has_key', 'items', 'keys', 'setdefault', 'update', 'values']
>>> import odbchelper
>>> dir(odbchelper)  3
['__builtins__', '__doc__', '__file__', '__name__', 'buildConnectionString']
```


- `callable()` devuelve 1 si su argumento puede ser invocado (es una función, método o clase), y 0 en caso contrario.

```
>>> import odbchelper
>>> callable(odbchelper.__doc__)
0
>>> callable(odbchelper.buildConnectionString)
1
```

- `getattr()` devuelve cualquier atributo de cualquier objeto

```
>>> li = ["Larry", "Curly"]
>>> getattr(li, "pop")           2
<built-in method pop of list object at 010DF884>
>>> getattr(li, "append")("Moe") 3
>>> li
["Larry", "Curly", "Moe"]
```


Los extravagantes and, or

- En una contexto booleano:
  - `0, (), [], {}, None` devuelven «falso»
  - cualquier otra cosa devuelve «verdadero»
- Si todos los elementos son verdaderos, `and` devuelve el último, en caso contrario devuelve el primer valor falso.
- Si algún elemento es verdadero, `or` lo devuelve inmediatamente, en caso contrario devuelve el último

```
>>> 'a' and 'b'
'b'
>>> '' and 'b'
''
>>> 'a' or 'b'
'a'
>>> '' or 'b'
'b'
>>> '' or [] or {}
{}
```

El «truco and-or» permite emular la sintaxis de C: `cond ? xxx : yyy`

```
>>> a = "first"
>>> b = "second"
>>> 1 and a or b
'first'
>>> 0 and a or b
'second'
```

No funciona si a puede ser evaluado como falso. Por si acaso, debería usarse:

```
>>> (1 and [a] or [b])[0]
```


Funciones lambda

Una función `lambda` permite una declaración «en línea», abreviada y anónima de una función.

- Puede tener cualquier número de parámetro (incluso opcionales)
- Devuelve el valor de una única expresión

```
>>> def f(x):
...     return x*2
...
>>> f(3)
6
>>> g = lambda x: x*2  1
>>> g(3)
6
>>> (lambda x: x*2)(3) 2
6
```

Orientación a Objetos

Fundamentos de Orientación a Objetos

- Tipos Abstractos de Datos: Un clase de objetos liga una estructura de datos con el código que trabaja directamente sobre ella: atributos y métodos
- Herencia: Es posible definir nuevas clases «heredando» de otras ya existentes:
  - están disponibles los atributos y métodos de la clase «padre»
  - pueden escribirse nuevos atributos y métodos
  - pueden redefinirse métodos de la clase padre para que se comporten de manera distinta para la clase hija

- Polimorfismo: Es posible que una variable referencia a cualquier instancia de cualquier clase de una jerarquía de herencia.
- Ligadura dinámica o despacho dinámico: Si en el código aparece una invocación de un método a través de una variable, hasta el tiempo de ejecución no sé decide qué metodo en concreto será llamado, en función de la instancia concreta de su jerarquía a la que haga referencia la variable.

Clases en Python

```
class MyClass:
    "A simple example class"
    def f(self):
        return 'hello world'
```

- Las clases también tienen documentación
- Los métodos siempre reciben como primer parámetro una referencia a la instancia sobre la que se invocan. Por convenio siempre se llama `self` a este parámetro.
- Cuando se invoca un método desde fuera, no hay que incluir ningún parámeto para `self`

- Creación de instancias

```
>>> x = MyClass()
>>> x.f()
'hello world'
```


El método __init__

- Es llamado automáticamente al crear una instancia. No debe llamarse explícitamente
- Puede tener cualquier número y tipo de argumento (el primero, `self`).
- Al crear una instancia, se pasan los parámetros que requiere `__init__` (excepto `self`).
- Nunca devuelve un valor

```
>>> class MyClass:
...     def __init__(self, value):
...             print value
...
>>> x = MyClass()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: __init__() takes exactly 2 arguments (1 given)
>>> x = MyClass(34)
34
>>> y = MyClass("casa")
casa
```


Atributos de instancia

- Los atributos de instancia suelen definirse en el `__init__`
- Para acceder a los atributos de instancia dentro de la clase, se usa la notación `self.atributo`

```
>>> class MyClass:
...     def __init__(self):
...             self.i = 1
...
>>> x = MyClass()
>>> y = MyClass()
>>> x.i
1
>>> x.i += 1
>>> x.i
2
>>> y.i
1
```

- Como cualquier variable, los atributos de instancia pueden «surgir» al vuelo en cualquier momento, lo que puede llevar a que diferentes instancias de la misma clase tengan distintos atributos, en un momento dado.
- Los atributos de instancia pueden eliminarse con `del`
- La función predefinida `vars()` permite ver la tabla de símbolos de una clase o una instancia. Nótese como los métodos están en la tabla de símbolos de la clase. También puede usarse `dir()`


```
>>> x.j = 5
>>> x.j
5
>>> y.j
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: MyClass instance has no attribute 'j'
>>> vars(MyClass)
{'__init__': <function __init__ at 0x810f8dc>, '__doc__': None,
'__module__': '__main__'}
>>> del x.i
>>> vars(x)
{'j': 5}
>>> vars(y)
{'i': 1}
```

Atributos de clase

- Los atributos de clase suelen definirse al principio de la definición de la clase, antes de los métodos.
- Para acceder a los atributos de clase dentro de la clase, se usa la notación `clase.atributo`
- Para acceder a los atributos de clase fuera de la clase, se usa la notación `clase.atributo` o `instancia.atributo`.
- Una asignación a `instancia.atributo` crea un nuevo atributo de instancia, ocultando el atributo de clase que sólo queda accesible a través de `clase.atributo`


```
>>> class MyClass:
...     i = 1
...     def __init__(self):
...             MyClass.i = MyClass.i + 1
...             self.j = 10
...
>>> x = MyClass()
>>> y = MyClass()
>>> x.i
3
>>> y.i
3
>>> MyClass.i
3
>>> x.i = 1000
>>> x.i
1000
>>> y.i
3
>>> MyClass.i
3
>>> del x.i
>>> x.i
3
```

Herencia

- Al definir la clase puede incluirse entre paréntesis uno o más ancestros (¡hay herencia múltiple!)
- Las referencias a atributos, si no se encuentran en la clase derivada, se buscan en la clase base.
- Si se escribe un método `__init__` para una clase derivada, hay que asegurarse de que llame al `__init__` de la clase base

```
class FileInfo(UserDict):
    def __init__(self, filename=None):
        UserDict.__init__(self)
        self["name"] = filename
```


- El atributo `__class__` permite comprobar a qué clase de una jerarquía pertenece una instancia dada en un instante dado.

```
def copy(self):
        if self.__class__ is UserDict:
            return UserDict(self.data)
```

Métodos predefinidos de las clases

Hay muchos. Algunos son:

- `__del__()` se invoca automáticamente cuando un objeto deja de existir
- `__repr__()` devuelve un string con la representación «oficial» del objeto
- `__cmp__()` se invoca por las operaciones de comparación

Restricción de acceso

- En Python, por defecto, todos los métodos y atributos son públicos.
- Para que un método o atributo sea privado, su nombre debe empezar por `__`, y NO terminar por `__`

```
>>> class MyClass:
...     def __init__(self):
...             self.i = 10
...             self.__j = 20
...
>>> x = MyClass()
>>> x.i
10
>>> x.__j
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: MyClass instance has no attribute '__j'
```

- ...¡Psst! La transparencia anterior no es cierta del todo:

```
>>> vars(x)
{'_MyClass__j': 20, 'i': 10}
>>> x._MyClass__j = 25
>>> vars(x)
{'_MyClass__j': 25, 'i': 10}
```

¡PERO NO SE TE OCURRA USAR ESTO NUNCA!

Referencias

- Mark Pilgrim, *Dive Into Python*:
  `http://diveintopython.org/`
- Guido van Rossum, *Python Tutorial*:
  `http://www.python.org/doc/current/tut/tut.html`
- Guido van Rossum, *Python Library Reference*:
  `http://www.python.org/doc/current/lib/`