Lenguajes de Scripting
Python

Mauricio Araya Jorge Valencia

Agenda

Índice

1. Lenguajes de Scripting

Características de lenguajes de scripting

- *"Script"* es *guión*
  - Invocar programas automáticamente
  - Lenguajes interpretados
  - Muy alto nivel
- Características generales
  - Manejo de strings y expresiones regulares
  - Sin tipos, variables no se declaran
  - Administración automática de memoria
- Características adicionales comunes
  - **Empotrable**: Puede usarse desde dentro de programas
  - **Extensible**: Pueden agregarse nuevas funcionalidades

Algunos lenguajes

- Python: Por *Monty Python*
- Perl: Oficialmente, abreviatura de *Practical Extraction and Report Language*
- PHP: Abreviatura de *PHP: Hypertext Processor*
- Ruby: La piedra preciosa (si hay perlas...)
- TCL: Es *Tool Control Language*, creado con la idea de ser un lenguaje de extensión/configuración
- expect: Una extensión de TCL, orientada a controlar programas interactivos
- AWK: Lenguaje inventado por Aho, Weinberger, Kernighan.

2. Introducción al lenguaje Python

Python

- Lenguaje de Programación Dinámico
- Pensado como Lenguaje Multiparadigma
- Diseñado para ser usado Intuitivamente
- Python incluye baterías!
  - Biblioteca estándar muy completa
- Python se lleva bien con sus amigos
  - Interoperabilidad con Java, .NET, CORBA, C++, Lisp, etc.
- Python no se aprende, se usa!
  - Orientación a objetos intuitiva, expresiones procedurales naturales, sintaxis minimal

Oficialmente...

Guido van Rossum afirma que Python es:

- Simple, fácil de aprender, libre y open source
- De alto nivel, portable e interpretado
- Orientado a objetos, extensible y embebible

Además aclara que Python:

- NO es un animal que estrangula su presa hasta matarla
- NO es un lenguaje que ofrece libertad
- NO es un lenguaje de solo escritura

Modo Interactivo

```
maray@foobar # python
>>> print "Hello World"
Hello World
>>> 3 * 5 + 10/2
20
>>> la_respuesta_universal=42
>>> if la_respuesta_universal==42:
...     print "Conoces el significado de la vida"
... else:
...     print "Aun te falta mucho por aprender"
...
Conoces el significado de la vida
```

Modo Línea de comandos

El tradicional programa en Python puede escribirse:

```
python -c 'print "Hello, world!"'
```

Acá `-c` indica que el argumento siguiente ha de considerarse como una línea del script a ser procesado por `Python`. Para evitar que el shell interprete el contenido del script (que comúnmente contendrá caracteres que son especiales para él) el script mismo se encierra entre apóstrofes.

Así, `Python` es *embebible* en la shell del sistema, y puede ocuparse para tareas típicas del sistema.

Modo Script

Al igual que otros lenguajes de scripting se ocupa el sha-bang (`#!`) para hacer el script ejecutable.

```
#!/usr/bin/env python
la_respuesta_universal=42
if la_respuesta_universal==42:
    print "Conoces el significado de la vida"
else:
    print "Aun te falta mucho por aprender"
```

Y después...

```
maray@foobar # python script.py
Conoces el significado de la vida
maray@foobar # chmod +x script.py
maray@foobar # ./script.py
Conoces el significado de la vida
```

2.1. Una mordida de Python

Ejemplo ODBC

Python asegura ser intuitivo, veamos que dice el público.

```
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)
```

Ejemplo Simple

Iteraciones Simples

```
str = "foo"; lst = ["monty", 42, "python"]
for char in str:
    print char
for elem in lst:
    print elem
```

Ejemplo Simple 2

Funciones e Iteraciones

```
def iterquad ():
    for i in range(5):
        yield (i*i)

for j in iterquad():
    print j
```

Ejemplo Simple 3

Parametros y Retorno Múltiple

```
def quadcube (x):
    return x**2, x**3

a, b = quadcube(3)
print a
print b
```

Conclusión

- Python es intuitivo para ejemplos simples
- Se torna desordenado para aplicaciones complejas (ODBC)
- No es la cura a todos los males, es solo otro paradigma
- Ideal para prototipos y aplicaciones pequeñas
- No recomendable para aplicaciones demasiado complejas (SIGA)
- No recomendable para trabajo a bajo nivel (Driver)

3. Lenguaje Python

3.1. Componentes Básicos

Literales

- **Números:**
  - enteros (`42`)
  - punto flotante (`1.66`)
  - notación científica (`1.2E10`)
  - complejos (`-2.1 + 0.9j`)
- **Strings:**
  - entre apostrofes (`'Hola Mundo'`)
  - entre comillas (`"maray's bar"`)
  - multilinea (triple comilla o apostrofe)

    ```
    '''Una linea de texto
       Aqui no se interpreta la indentacion'''
    ```
  - Los Strings son inmutables
  - Python no tiene tipo `char`

Identificadores

- Nombres de identificadores se componen igual que en C
- **Variables**: cualquier identificador es una variable a no ser que se demuestre lo contrario
- **Tipos de Datos**: clases, strings y números
- **Funciones:** los subprogramas definidos por el operador `def`
- **Objetos:** todo lo anterior son objetos, ¡incluyendo strings y números!

Indentación

En Python la indentación es parte de la *sintaxis* y *semántica* del programa, a diferencia de C o Java

- Los bloques están definidos por la indentación
- No existen literales de bloque como `begin/end` o `{ }`
- Es muy importante ser consistente en la indentación
- Un error en la indentación genera un error en el programa

```
i = 5
 print 'El valor es', i # ERROR
print 'Repito, el valor es', i
```

```
maray@foobar # python indent.py
  File "indent.py", line 2
      print 'El valor es', i # ERROR
      ^
SyntaxError: invalid syntax
```

Operadores

- Operadores matemáticos, bit-a-bit y relacionales: igual que C y Java
- Tiene operadores matemáticos extendidos a strings (i.e. `'a' + 'b' --> 'ab'` o `'a' * 3 --> 'aaa'`)
- Nuevos operadores matemáticos: power (`**`) y floor divition (`//`)
- Operadores lógicos: `not`, `and` y `or`

Operadores

- Operador funcional: `lambda`
- Operadores de membresía e identidad: `in`, `not in`, `is not is`
- Operadores de subíndice, rango, tupla, lista, diccionario, etc: `x[]`, `x[:]`, `(e1,e2...)`, `[e1,e2...]`, `{key:dat,...}`, etc
- Precedencia: lambda, lógicos, membresía/identidad, relacionales, bit-a-bit, matemáticos, subíndice, rango, tupla, lista, diccionario, etc.

Control de Flujo

- Sacados del paradigma imperativo
- Definidos por el bloque de indentación y el operador `:`
- `if` se puede utilizar con `elif` y `else`
- `while` se puede utilizar con `else`
- `for` funciona sobre *sequencias*, como listas, strings, diccionarios, etc. Es similar a un foreach.
- También existen los saltos incondicionales implícitos mediante `break` y `continue`

Ejemplo `while`, e `if`

```
number = 23
running = True
while running:
        guess = int(raw_input('Ingrese un numero: '))

        if guess == number:
                print 'Correcto!'
                running = False # this causes the while loop to stop
        elif guess < number:
                print 'Incorrecto, es mayor'
        else:
                print 'Incorrecto, es menor'
else:
        print 'Termino el Loop.'
        # Do anything else you want to do here
print 'Termino el programa'
```

Ejemplo `while` con Salto Incondicional

```
while True:
        s = raw_input('Ingrese String :')
        if s == 'quit':
                break
        if len(s) > 3:
                continue
        print 'Tiene que ser mas largo'
```

Ejemplo `for`

```
for i in range(1, 5):
        print i
else:
        print 'Se termino'

for i in ['pato', 'peto', 'pito', 1.3E04]:
        print i
else:
        print 'NEXT!'
complex = {'uno': 10.23+2j , 'dos':1.32+43j }
for i,j in complex.items():
        print i, ' = ', j
```

3.2. Subprogramas

Funciones `def`

- Una definición de función es una sentencia ejecutable.
- Objetos de primera clase que se ligan en el espacio de nombres del archivo
- Definidos por el bloque de indentación y el operador **:**
- Los parametros son variables y pueden tener valores por defecto

```
<funcdef> ::=
   def <funcname> ( [<par_list>] ) : <staments>
<par_list> :==
   <parameter>[=<value>]| <par_list>,<par_list>
```

Ejemplo de Funciones

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

# Now call the function we just defined:
fib(2000)
#1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

Ejemplo de Valores por Defecto

```
def ask_ok(prompt, retries=4, complaint='Yes or no, please!'):
    while True:
        ok = raw_input(prompt)
        if ok in ('y', 'ye', 'yes'): return True
        if ok in ('n', 'no', 'nop', 'nope'): return False
        retries = retries - 1
        if retries < 0: raise IOError, 'refusenik user'
        print complaint

ask_ok('Hola: ')
ask_ok('Chao: ',complaint='en gringo yes/no')
ask_ok('Prompt ',1,"nada")
```

Funciones Anónimas

- Python soporta funciones anónimas usando **lambda**
- Pueden ser ligadas a nombres de forma dinámica
- Se pueden ocupar como funciones parametrizables
- Simulan paradigma funcional

```
#!/usr/bin/env python
g = lambda x: x**2
print g(8)
# 64

def crear_sumador(n): return lambda x: x + n
f = crear_sumador(2)
g = crear_sumador(6)
print f(42), g(42)
#44 48
print crear_sumador(22)(33)
#55
```

Documentación de Funciones

- Despues del **def** se utiliza un string multilinea
- La primer linea es la descripción corta
- Despues viene una linea en blanco
- Por último la descripcion larga

```
#!/usr/bin/env python
def mi_funcion():
    """Funcion inutil.

    De verdad esta funcion no hace nada
    """
    pass

help(mi_funcion)
```

3.3. Estructuras de Datos

Listas

- Listas son estructuras de datos compuestas
- Se pueden ocupar como arreglos
- No son inmutables, y son anidables

```
#!/usr/bin/env python
a = ['john', 'locke', 42, 101]
b = ['a','b','c',a]
b # ['a', 'b', 'c', ['john', 'locke', 42, 101]]
a[0] # john
a[1:3] + ['Boo!'] # ['locke', 42, 'Boo!']
2*a[:3] # ['john', 'locke', 42, 'john', 'locke', 42]
a[2]=a[2] + 1
```

Funciones para Listas

```
len(list)
del list[x:y]
list.extend(l)
list.insert(i,x)
list.remove(x)
list.count(x)
list.sort()
list.reverse()
list.append(x) # Pila y Cola
list.pop()     # Pila
list.pop(0)    # Cola
```

Tuplas

- Como Python es multiparadigma incluye tuplas
- Muy similares a las listas, pero inmutables
- Se utilizan con parentesis, en vez de corchetes
- Muy utilizadas para retornos múltiples

```
#!/usr/bin/env python
t = 12345, 54321, 'hello!'
t[0] # 12345
t #(12345, 54321, 'hello!')
u = t, (1, 2, 3, 4, 5)
u # ((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
def func():
   return 3,4
x,y = func()
print x,y
# 3 4
```

Conjuntos

- Collección no ordenada de elementos sin duplicados
- Un Conjunto es un objeto que soporta operaciones matemáticas
- Operadores:

Code	Descripción	Fórmula
A=**set**(**list**)	Crea conjunto	$A = \{l_1, l_2, ..., l_n\}$
obj **in** A	Membresia	$obj \in A$
A | B	Unión	$A \cup B$
A & B	Conjunción	$A \cap B$
A − B	En A pero no en B	$A - (A \cap B)$
A ^ B	En A o B pero no en ambos	$(A \cup B) - (A \cap B)$

Diccionarios

- Tambien llamados arreglos asociativos
- Colección indexada por llaves (keys)
- Las *keys* son de cualquier tipo inmutable
- Se definen como $\{k_1 : v_1, k_2 : v_2, ..., k_n : v_n\}$
- No tienen orden implícito, se puede ordenar con sort().
- Operador var[key] permite recuperar y/o asignar
- **dict**() crea diccionarios desde otras estructuras de datos
- **del** hace lo suyo!

Ejemplo de diccionarios

```
#!/usr/bin/env python
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
'guido' in tel # True
dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
# {'sape': 4139, 'jack': 4098, 'guido': 4127}
dict([(x, x**2) for x in (2, 4, 6)])
# {2: 4, 4: 16, 6: 36}
```

3.4. Expresiones Regulares

Expresiones Regulares

- Patrones de búsqueda y reemplazo para strings
- Forma simple y resumida de tratar con texto
- Pequeño *lenguaje* embebido en Python

- El uso indiscriminado puede llevar a poca legibilidad
- Una expresión regular se compone de:
  - **Caracteres:** se representan a sí mismos
  - **Metaracteres:** representan *reglas*
- Similar a la idea de un BNF pero más sencillo!

Expresiones Regulares

- Cada construcción calza con un patrón dado, en una posición específica. A las construcciones que sólo registran una posición (no calzan con caracteres) les llaman *anchor*.
- Las construcciones en general son voraces: Calzan lo más que pueden. Pero también son poco previsoras: Calzan en la primera oportunidad, aunque hayan calces "mejores" más adelante.

Metacaracteres

.	calza cualquier caracter menos \n
^	calza el principio del string
\$	calza el final del string
*	calza 0 o más repeticiones de un caracter/regla
+	calza 1 o más repeticiones de un caracter/regla
?	calza 0 o 1 caracteres
*?,+?,??	calces no voraces (i.e. `<h1>title</h1>`)
{m}	calza `m` veces la ER anterior
{m,n}	calza `m` a `n` veces (no voraz con `?`)
\	escapa los metacaracteres (i.e. `\*`)
[]	conjunto de caracteres. Caracter − para rangos.
|	`A|B` calza `A` o `B`
()	calza la expresión entre parentesis completa (agrupa)

Otros Modificadores

\number	calza `number` veces lo que le precede
\A	calza **solo** al inicio del string
\b	calza el string vacio al principio/final de una palabra
\B	calza el string vacio no al principio/final de una palabra
\d	calza caracteres decimales (`[0-9]`)
\D	calza todo caracter no decimal (`[^0-9]`)
\s	calza todo caracter *blank* (`[ \t\n\r\f\v]`)
\S	calza todo caracter no *blank* (`[^ \t\n\r\f\v]`)
\w	calza todo caracter alfanumérico (`[a-zA-Z0-9_]`)
\W}	calza todo caracter no alfanumérico (`[^a-zA-Z0-9_]`)
\Z	calza **solo** al final del string

Algunas ER

```
(.*)blog(.*)\.com$
# dominios .com que contengan blog
pagina\s{1,8}(\d+)
# palabra pagina, multiples espacios y un numero
0x[0-9ABCDEF]*
# numero hexadecimal
```

```
linux+
# 'linu' y muchas 'x'
^begin (.*) end$
# linea que parta con 'begin ' y termine con ' end'
```

Uso de Expresiones Regulares

- Las ER al ser parte de un mini-lenguaje (lenguaje regular), debe ser compilado o interpretado para utilizarlo
- Perl interpreta las expresiones regulares para su facil uso
- Python (como C y otros) deben compilar la expresión
- Hay que importar la biblioteca `re`
- Se usa **compile**() para crear el objeto ER

```
import re
p = re.compile('ab*')
print p
# <_sre.SRE_Pattern object at 0xb7e013e0>
```

Funciones para Expresiones Regulares

- search(pat,**str**) : busca en un string por el patrón, retorna un *MatchObject*
- match(pat,**str**) : intenta calzar el patrón partiendo del principio del string. Retorna un *MatchObject*.
- split (pat,**str**) : divide el string dependiendo del patrón. Retorna una lista de strings.
- findall (pat,**str**) : retorna una lista de todos los calces.
- sub(pat,rep,**str**) : reemplaza lo calzado con un string `rep`, o con una función `rep`. En este último caso, la función recibe un parámetro *MatchObject*.
- subn(pat,rep,**str**) : igual que el anterior pero

Se puede utilizar la versión orientada a objetos mediante médotos del objeto `re`. Corresponden a las mismas funciones sin el parametro `pat`. Es más eficiente.

Ejemplo de uso de ER

```
import re
def checkMatch(myMatch):
    if myMatch: print "Si calza!"
    else: print "No calza!"
match1=re.search('^From','From John Lock:')
match2=re.compile('^From').search("To Jacob")
match3=re.search('From',"Desde From From")
match4=re.match('From',"Desde From From")
checkMatch(match1);checkMatch(match2)
checkMatch(match3);checkMatch(match4)
str1=re.split('\W+',"4 8 15  16 23  42")
str2,n=re.subn('-'," ","Archivo-de-prueba-2")
print str1,'\n',str2," (",n," subs )"
```

`MatchObject`

- Objeto que representa un calce.
- Usado para tratamiento avanzado de calces.
- Algunos Métodos:

```
m.start() # donde empieza el calce
m.end()   # donde termina
m.span()  # tupla (m.start(),m.end())
```

- Algunos Atributos:

```
.re     # objeto ER
.string # string original
.pos    # posicion inicial de busqueda
.endpos # posicion final de busqueda
```

4. Python en Serio

4.1. Módulos

Intérprete y código fuente

- Problema: cuando cierras el intérprete se pierde todo el código escrito.
- Solución: Escribir archivos de código fuente, AKA *scripts* (*guiones*).
- Problemas: ¿qué pasa si yo quiero reutilizar sólo las funciones definidas en un guión? ¿cómo hago para dividir un guión en multiples archivos?
- Solución: *Módulos*.

Módulos

- Básicamente, un módulo es un archivo de código fuente Python.
- El nombre de este archivo **debe** ser `nombre_modulo.py`.
- Dentro del código de un módulo, el nombre de este puede accederse por medio de la variable `__name__`.
- Un archivo usado como programa siempre lleva el nombre de módulo `__main__`.
- Cualquier módulo puede hacer uso de las definiciones en otro módulo mediante `import`.

Ejemplo básico

```
# modulos.py
print __name__;

#modulos-programa.py

import modulos

print __name__
```

```
jorjazo@bodoque:~$ python modulos-programa.py
modulos
__main__
jorjazo@bodoque:~$ python modulos.py
__main__
```

Ejemplo: Fibonacci

```
# modulo de numeros Fibonacci
def fib(n):    # escribe la serie de Fibonacci hasta n
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b
    print

def fib2(n): # retorna la serie de Fibonacci hasta n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result

if __name__ == "__main__" :
        fib(100)
        serie = fib2(100)
        for numero in serie :
                print numero
```

Ejemplo: Fibonacci

```
jorjazo@bodoque:~$ python fibo.py
1 1 2 3 5 8 13 21 34 55 89
1
1
2
3
5
8
13
21
34
55
89
```

Ejemplo: Fibonacci

```
jorjazo@bodoque:~$ python
>>> import fibo
>>> fibo.fib(100)
1 1 2 3 5 8 13 21 34 55 89
>>> fib2(100)
NameError: name 'fib2' is not defined
>>> from fibo import fib2
>>> fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fib(100)
NameError: name 'fib' is not defined
```

```
>>> from fibo import *
>>> fib(100)
1 1 2 3 5 8 13 21 34 55 89
```

¿Dónde se buscan los módulos?

1. variable `sys.path`, automáticamente inicializada al directorio actual (¡no es lo mismo que el directorio del script!)
2. en las ubicaciones descritas en la variable de entorno `PYTHONPATH`
3. ubicación dependiente de la instalación del intérprete (en sistemas operativos Unix esta es, usualmente, `.:/usr/local/lib/py`

Paquetes

- Son espacios de nombres para módulos.
- Los módulos deben organizarse en una estructura de directorios que represente la estructura de paquetes (similar a Java)
- Cada directorio a ser tratado como paquete debe contener un archivo llamado `__init__.py`.
- Este archivo puede estar en blanco o tener código de inicialización del paquete.
- En él se puede definir una variable especial `__all__` con los nombres de los módulos que contiene el paquete.

4.2. Excepciones

Errores y excepciones

- Se identifican dos cursos excepcionales de ejecución en Python: Errores de sintaxis y excepciones.
- Los errores de sintaxis ocurren (de modo obvio) cuando el intérprete detecta problemas en la sintaxis del código.
- Las excepciones ocurren debido a algún problema generado por la ejecución del código.
- Ejemplos típicos: división por cero (`ZeroDivisionError`), nombres no definidos (`NameError`), errores de conversión de tipos (`TypeError`), etc.
- Se tratan de un modo similar a Java.

Excepciones

- Se pueden generar con `raise NombreDeError`.
- Sintácticamente se tratan con cláusulas `try`.
- Se capturan con bloques `except`.
- Acepta una cláusula final `else` para especificar un bloque de código que deba ejecutarse después del `try`, con o sin excepciones.

Cláusulas `except`

- `except NombreDeError`: para especificar qué hacer en caso de un error tipo `TipoDeError`.
- `except NombreDeError, (errno, strerror)`: igual que la anterior, pero dentro del bloque se puede utilizar las variables `errno` y `strerror` que contienen un número de error y un mensaje de error, respectivamente.
- `except (ErrorUno, ErrorDos, ErrorTres)`: permite camputrar error de tipo `ErrorUno`, `ErrorDos` y `ErrorTres` en un mismo bloque.
- `except`: sólo se puede usar como última cláusula `except` en una sentencia `try`, y sirve de comodín la excepciones no capturadas anteriormente.

Ejemplo

```
import sys
try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except IOError, (errno, strerror):
    print "I/O error(%s): %s" %(errno, strerror)
except ValueError:
    print "Could not convert data to an integer."
except:
    print "Unexpected error:", sys.exc_info()[0]
    raise
```

4.3. Clases y Objetos

Sintaxis básica

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

Sintaxis básica

- No requieren estar en un archivo en especial.
- Su definición **debe** ser ejecutada antes de su uso.
- Se pueden definir en cualquier parte: dentro de un `if`, función, etc.

Características de Clases

- Para Python, estas son básicamente un espacio de nombres.
- En su definición se incluyen, más que nada, definiciones de funciones (métodos).
- Luego de ejecutarse la definición de una clase, el lenguaje crea un *objeto de clase*.

Métodos

- Los métodos de una clase son considerados atributos cuyo tipo es método.
- Métodos son una especie particular de función: son funciones que reciben, como primer argumento, una referencia al objeto a tratar. A la hora de invocación no es necesario pasar este argumento, pues Python entiende que subyace en llamadas como `obj.metodo()`. Sin embargo, este argumento sí se debe incluir en la declaración del método.

Clases: ejemplo básico

```
class MyClass:
    "A simple example class"
    i = 12345
    def f(self):
        return 'hello world'
```

Clases: ejemplo básico

- Tanto `MyClass.i` como `MyClass.f` son atributos válidos.
- `MyClass.__doc__` también lo es, y contiene la documentación de la clase
- `f` es de tipo método... `MyClass.f()` es una invocación válida, aunque parezca que le falta un parámetro.
- Métodos con nombre `__init__` corresponden a constructores.

Objetos

- Como es sabido, un objeto es una instancia de una clase.
- Además de los objetos instaciados comunmente, existen *objetos de clase*, que son los que representan a una clase en particular (e.g. `MyClass`).
- Instanciación se realiza mediante notación de funciones (i.e. `NombreDeClase([args])`).

Objetos: ejemplo

```
>>> class Complex:
...     "Representa numeros complejos."
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
...     def real(self):
...         return self.r
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
>>> realx = x.real
>>> realx()
3.0
>>> x.r = 1
>>> realx()
1
>>> Complex.__doc__
'Representa numeros complejos'
>>> x.__doc__
'Representa numeros complejos'
```

Herencia

Python permite herencia simple:
class DerivedClassName(BaseClassName)
...y herencia múltiple

Herencia múltiple

- Colisión de nombres de resuelve con *depth-first*, de izquierda a derecha.
- Esto es:
  1. La misma clase (`DerivedClassName`).
  2. La primera superclase, hacia la izquierda (`Base1`).
  3. Recursivamente sobre las superclases de la superclase.
  4. La segunda superclase ...

5. Entrada/Salida

Funciones simples

- **print**() : Imprime tuplas y las separa por espacios
- **file** .write() : Función para imprimir sobre un descriptor.

```
import sys
print "hola " + "mundo"
print "hola", "mundo"
print ("hola", "mundo")
print ["hola", "mundo"]
print sys.stdout
sys.stdout.write("hola mundo\n")
```

Formateo de Salida

- Formateo de Strings
  - Operadores de String (como +,*,etc)
  - Funciones Adicionales
    - **str**() : Genera strings en formato 'humano'
    - **repr**() : Genera strings en formato 'maquina'
- Formateo de Salida al estilo C (%)

Funciones Adicionales

```
import math
s = 'Hello world.'
str(s) # 'Hello world.'
repr(s) # "'Hello world.'"
str(0.1) # '0.1'
repr(0.1) # '0.10000000000000001'
for x in range(1, 11):
    print repr(x).rjust(2), repr(x*x).rjust(3),
    print repr(x*x*x).rjust(4)
'12'.zfill(5)
for x in range(1,11):
    print '%2d %3d %4d' %(x, x*x, x*x*x)
print 'Pi es app %5.3f.' %math.pi
table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 86378}
print '%(Jack)d; %(Sjoerd)d; %(Dcab)d' %table
```

Tratamiento de archivos

- Objeto file se crea con la función f=**open**(filename,mode)
- f.read() lee el archivo completo
- f.readline() y f.readlines() leen lineas
- f.write(s) escribe el string s en el archivo
- f.seek(bytes) cambia la posición en el archivo
- f.close() cierra el descriptor
- Nota: el modulo `pickle` se utiliza para la serialización de objetos (objetos persistentes).

EOF