Programación en Python

Jesús Espino García

Grupo de usuarios de Linux
Universidad Carlos III de Madrid.

18 de Noviembre de 2004

Introducción.

¿Por qué Python?

- Python es fácil de aprender.
- Python es sencillo de usar.
- Python es potente.

¿Qué es Python?

- Es un lenguaje de programación.
- Fue creado en las navidades de 1989.
    - Su autor es Guido van Rossum.
    - En su origen era un lenguaje para la gestión de Amoeba.
- Basado en ABC y Modula-3
- En febrero de 1991 pasa a USENET
- A partir de ahí el lenguaje no ha dejado de crecer.
    - Actualmente tenemos la versión 2.3.

¿Por qué es especial?

- Es libre (y gratis).
- Es fácil de escribir.
- Es fácil de leer.
- Es fácil de mantener.
- Es de propósito general.

¿Por qué es especial?

- Alto nivel.
- Orientado a objetos.
- Interpretado
- Introspectivo

¿Por qué es especial?

- Extensible.
- Completo.
- Dinámico.
- Robusto.
- Múltiples plataformas.
- Colaborativo.

¿Por qué es especial?

- Herencia múltiple.
- Funciones sobre listas.
- Funciones lambda
- . . .

¿Quien lo usa?

- Bea Systems
- Walt Disney Company
- GE Aircraft Engines
- Google
- Hewlett-Packard
- IBM
- Microsoft
- NASA
- Nacional Center for Atmosferic Research
- Red Hat
- U.S. National Weather Service
- U.S. Navy
- Xerox
- Yahoo!
- Zope Corporation
- ...

Python.

El interprete

```
$ python
Python 2.3.4 (#2, Sep 24 2004, 08:39:09)
[GCC 3.3.4 (Debian 1:3.3.4-12)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Definición de variables

- No hace falta definir variables.
- Los tipos de datos son dinámicos.
- Es sensible mayúsculas y minúsculas.

```
>>> a=1
>>> b=1.0
>>> c="1.0"
>>> d='hola'
>>> e=5j
```

Tipos de datos básicos

- Enteros.
- Coma flotante.
- Números complejos.
- Números de precisión arbitraria.
- Cadenas de caracteres.
- Tuplas.
- Listas.
- Diccionarios.
- Son dinámicos.

Listas

- Se identifica por []
- Lista vacía: []
- Elementos separados por comas
  - `[1,2,3,4]`
- Elementos heterogéneos
  - `[1,(2,4),"avión",["gul","linux","python"]]`
- Acceso a un elemento
  - `lista[posición]`
- Listas dentro de listas
  - `lista[indice1][indice2]..[indiceN]`

Listas

- Los índices pueden contar también desde el final

0	1	2	3	4
-5	-4	-3	-2	-1

Listas

- También se pueden seleccionar fragmentos

```
>>> lista = range(5)
>>> lista
[0, 1, 2, 3, 4]
>>> lista[1:-2]
[1, 2]
```

- Devuelve una lista.

Tuplas

- Similares a las estructuras de C.
- No hace falta definirlas.
- Se crean usando `()`.
- Sus elementos pueden ser heterogéneos.
- Se accede a sus elementos igual que una lista.

Ejemplos con tuplas

```
>>> tupla1=(1,2,3)
>>> tupla1
(1, 2, 3)
>>> tupla2=(tupla1,["gul","linux"])
>>> tupla2
((1, 2, 3), ['gul', 'linux'])
>>> tupla1[2]
3
>>> tupla2[1]
['gul', 'linux']
```

Diccionarios

- Como las tablas hash en Java.
- Se identifican con `{}`.
- Sus elementos están asociados a una clave.
- Para acceder a un elemento:
  - `diccionario[clave]`
- Las claves deben ser únicas.
- Los elementos complejos no pueden ser claves.

Diccionarios

- Algunos métodos:
  - `has_key(clave)`: Devuelve 1 si existe la clave.
  - `items()`: Devuelve una lista con el contenido.
  - `iteritems()`: Itera sobre la tupla (clave:elemento)
  - `iterkeys()`: Itera sobre las claves.
  - `keys()`: Devuelve una lista con las claves.
- Mas información `help(dict)`.

Buceando dentro de Python

- Lenguaje introspectivo.
- `dir()` muestra los objetos que hay en la memoria.
- Los métodos también son objetos.

```
>>> dir()
['__builtins__', '__doc__', '__name__', 'a', 'b', 'c', 'd', 'e']
```

Buceando dentro de Python

- Es muy útil cuando tienes mala memoria.

```
>>> dir([])
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__','
__delslice__', '__doc__', '__eq__', '__ge__', '__getattribute__', '__ge
titem__', '__getslice__', '__gt__', '__hash__', '__iadd__', '__imul__',
'__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne
__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmul__',
'__setattr__', '__setitem__', '__setslice__', '__str__', 'append', 'cou
nt', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

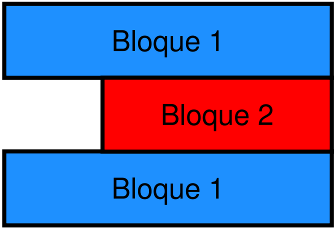
Un poquito de ayuda

```
>>> help([].append)
Help on built-in function append:

append(...)
    L.append(object) -- append object to end
```

Indentado

- El lenguaje es sensible al indentado.
- Después de : hay un bloque.



Bloque 1

Bloque 2

Bloque 1

Condiciones

```
if condicion1:
    bloque si se cumple la condición
elif condicion2:
    bloque si no se cumple la condición 1 y si la 2
else:
    bloque si no se cumple ninguna condición anterior.
```

Bucles

```
while condicion:
    Lo que se hace en el bucle
```

- `break` sale del bucle.
- `continue` pasa a la siguiente iteración.

Iteraciones

```
for variable in lista:
    Lo que esta dentro de la iteración
```

- Se basa en las listas.
- Función `range()`.

Rangos

- `range(x)` devuelve una lista con un intervalo

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(5,7)
[5, 6]
```

Definición de funciones

```
def funcion(argumento1,argumento2...):
    'Documentación de la función'
    Contenido de la función
    [Opción: return salida]
```

Parámetros de las funciones

- Puede tener cero, uno o varios.
- Puedo llamarla con menos parámetros.
- Puedo indicar los parámetros con los que llamo.
- Puedo poner valores por defecto.

Veamos un ejemplo

```
def diHolaMundo(mensaje="Hola Mundo!",numVeces=1):
    'Mi función hola mundo pesado de ejemplo.'
    for i in range(numVeces):
      print mensaje
```

Usando mi función de ejemplo

```
>>> diHolaMundo()
Hola Mundo!
>>> diHolaMundo("Hola a todos")
Hola a todos
>>> diHolaMundo("Hola a todos",3)
Hola a todos
Hola a todos
Hola a todos
>>> diHolaMundo(3)
3
>>> diHolaMundo(numVeces=3)
Hola Mundo!
Hola Mundo!
Hola Mundo!
```

Creando nuestra propia ayuda

```
>>> help(diHolaMundo)
Help on function diHolaMundo in module __main__:

diHolaMundo(mensaje='Hola Mundo!', numVeces=1)
    Mi función hola mundo pesado de ejemplo.

>>> diHolaMundo.__doc__
'Mi función hola mundo pesado de ejemplo.'
```

Escribiendo programas

- En archivos .py.
- Primera linea de un script de Unix.
    - `#!/usr/bin/python`
- Ficheros .pyc son bibliotecas precompiladas.
- Podemos llamarlo desde la linea de comandos.
    - python programa.py
    - ./programa.py

Entrada por teclado

- `input("cadena")`: Muestra la cadena y obtiene un entero.

```
>>> x=input("Obtener valor: ")
Obtener valor: 5
>>> x
5
```

- `raw_input("cadena")`: Muestra la cadena y obtiene una cadena.

```
>>> y=raw_input("Obtener valor: ")
Obtener valor: 5
>>> y
'5'
```

Parámetros de entrada

- sys.argv es una lista con los parámetros.
- Utilización:

```
import sys
nombre = sys.argv[0]
primer_parametro = sys.argv[1]
...
```

Consejos para los programas

- Es bueno dividir el código en funciones.
- Podemos incluir código de prueba en un archivo.
- Se ejecuta muy fácil con:

```
if __name__=='__main__':
    testme()
```

Un poco sobre ficheros

- Crear un objeto fichero:
  - `file = open('nombre','modo')`
- Cerrar un fichero:
  - `file.close()`
- Leer un fichero:
  - `file.read()`,`file.readline()`,`file.readlines()`
- Guardar en un fichero:
  - `file.write('texto')`,`file.writelines('texto')`

Trabajar con módulos

- Similar a las bibliotecas en C.
- Agrupan archivos.
- Se cargan con:
  - `import modulo`
- Para llamar al contenido se antepone el nombre del modulo:
  - `modulo.funcion()`
  - `modulo.variable`
- Similar a los espacios de nombres.

Trabajar con módulos

- Puede cargar todos los contenidos de un modulo al espacio de nombre actual:
  - `from modulo import *`
- Similar al `using namespace` de C++.
- También se pueden importar los ficheros del usuario.

Algunos módulos básicos

- sys
    - Contiene funciones del sistema:
        - `argv`,`exit`,`stderr`,. . .
- os
    - Permite llamadas al sistema operativo:
        - `popen`,`fork`,`chdir`,. . .
- os.path
    - Trabaja con las rutas de los archivos:
        - `isfile`,`exists`,`join`,. . .

Clases

```
class MiClase:
    def setDato(self,dato):
        self.Dato=dato
    def display(self):
        print self.Dato
```

Clases

```
>>> x=MiClase()
>>> y=MiClase()
>>> x.setDato(4)
>>> y.setDato('Hola')
>>> x.display()
4
>>> y.display()
Hola
>>>
```

Herencia

```
class OtraClase(MiClase):
    def display(self):
        print 'El valor actual es', self.Dato
```

```
>>> z=OtraClase()
>>> z.setDato('Herencia')
>>> z.display()
El valor actual es Herencia
```

Sobrecarga de operadores

- Existen métodos especiales dentro de las clases:
  - `__init__`: Constructor.
  - `__del__`: Destructor.
  - `__add__`: Operador de suma.
  - `__or__`: Operador O lógico.
  - `__getitem__`: Indexación.
  - `__setitem__`: Asignación indexada.
  - `__getslice__`: Seleccionar una parte.
  - `__repr__`: Para salida por pantalla.
  - `__len__`: Longitud.
  - `__cmp__`: Comparación.

Ejemplo con suma

```
class MiClase2(MiClase):
    def __init__(self, num=0):
        self.Dato=num
    def __add__(self, other):
        return MiClase2(self.Dato + other.Dato)
```

```
>>> x=MiClase2(3)
>>> y=MiClase2(6)
>>> (x+y).display()
9
```

Excepciones

- Para manejo de errores, notificación de eventos,. . .

```
try:
    <Código...>
except:
    Nombre_excepción:
    <Código para la excepción>
else:
    <Código a ejecutar si no se produce ninguna>
```

```
try:
    <Código...>
finally:
    <Código que se ejecuta siempre>
```

Excepciones

- raise permite lanzar una excepción de forma manual.
- Puedes crear excepciones propias como una cadena de caracteres.

```
>>> ex1= 'Problema indeterminado'
>>> raise ex1
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
Problema indeterminado
```

Ejemplo de excepciones

```
while True:
    try:
        x=int(raw_input("Introduzca un número: "))
        break
    except ValueError:
        print "Error: No ha introducido un número entero."
```

```
while True:
    try:
        x=raw_input("Introduzca un 5: ")
        if x!='5':
          raise 'Excepción de ejemplo'
    except 'Excepción de ejemplo':
        print "Error: El valor introducido no es un 5."
    else:
        print "Valor introducido correcto. Muchas gracias."
        break
```

¿Que queda en el tintero?

- Funciones lambda.
- Programación funcional.
- Comprensión de listas.
- Funcionamiento interno.
- Jython.
- Profile.
- Psyco.
- Extensión con C/C++.
- Empotramiento en C/C++.
- Mucho más. . .

Para terminar.

Referencias

- http://www.python.org
- Usos:
  - http://www.pythonology.org/success&story=esr
- Tutoriales:
  - http://www.python.org/doc/current/tut/tut.html
  - http://es.diveintopython.org
  - http://www.freenetpages.co.uk/hp/alan.gauld/spanish
- Listas de correo en castellano:
  - http://listas.aditel.org/listinfo/python-es
  - http://gul.uc3m.es/mailman/listinfo/gul

Dudas

. . .

Agradecimientos

- Gracias a Pablo Barrera por dejarme sus transparencias para hacer las mías.
- Gracias a todos los que me han ayudado para que estos cursos puedan salir adelante.
- Y gracias a ustedes por venir.

Fin