



Curso de Introducción a Python

Andrés Orcajo

`andres@acm.asoc.fi.upm.es`

Daniel Morán

`daniel@acm.asoc.fi.upm.es`

ACM Facultad de Informática
Universidad Politécnica de Madrid
`http://acm.asoc.fi.upm.es`

28 de febrero y 1 de marzo de 2011



Contenido

- 1 Introducción
- 2 Primeros Pasos
- 3 Tipos de Datos
- 4 Entrada/Salida
- 5 Control de Flujo
- 6 Módulos
- 7 Excepciones
- 8 POO
- 9 Ficheros
- 10 Interfaces gráficas



Contenido

- 1 Introducción
- 2 Primeros Pasos
- 3 Tipos de Datos
- 4 Entrada/Salida
- 5 Control de Flujo
- 6 Módulos
- 7 Excepciones
- 8 POO
- 9 Ficheros
- 10 Interfaces gráficas



Características

Características del lenguaje:

- Es interpretado.
- Es un lenguaje de muy alto nivel.
- Es un lenguaje de tipado débil.
- Es multiparadigma (Estructurado, POO, Funcional)





¿Por qué Python?

¿Y por qué debería de usar yo Python?

- Es un lenguaje de desarrollo **rápido**.
- Lo que lo hace ideal para prototipados...
- ... siendo igualmente potente para proyectos mayores.
- Multiplataforma.
- Dispone de una API muy extensa.



Contenido

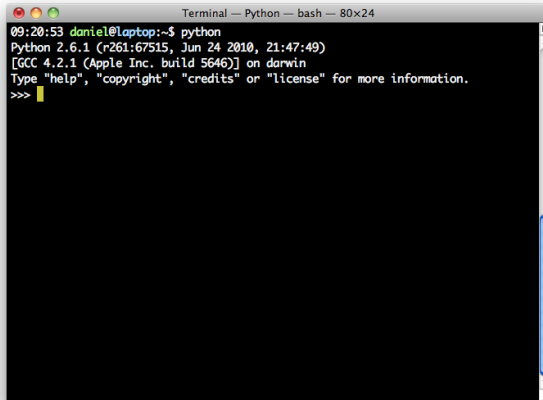
- 1 Introducción
- 2 Primeros Pasos**
- 3 Tipos de Datos
- 4 Entrada/Salida
- 5 Control de Flujo
- 6 Módulos
- 7 Excepciones
- 8 POO
- 9 Ficheros
- 10 Interfaces gráficas



Modo Interactivo

Lanzamiento

```
$ python
```

A screenshot of a terminal window titled "Terminal — Python — bash — 80x24". The window shows the output of running the 'python' command. The prompt is '09:20:53 daniel@laptop:~\$'. The output lines are: 'Python 2.6.1 (r261:67515, Jun 24 2010, 21:47:49)', '[GCC 4.2.1 (Apple Inc. build 5646)] on darwin', and 'Type "help", "copyright", "credits" or "license" for more information.'. The prompt changes to '>>>' and a yellow cursor is visible on the line following it.

```
Terminal — Python — bash — 80x24
09:20:53 daniel@laptop:~$ python
Python 2.6.1 (r261:67515, Jun 24 2010, 21:47:49)
[GCC 4.2.1 (Apple Inc. build 5646)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> |
```



Creando un Script

Path del Intérprete

/usr/bin/env python

A terminal window titled "Terminal — bash — bash — 80x24" showing the following commands and output:

```
22:13:42 daniel@laptop:~$ cat script.py
#!/usr/bin/env python

print "Hola"

22:13:45 daniel@laptop:~$ chmod +x script.py
22:13:56 daniel@laptop:~$ ./script.py
Hola
22:14:00 daniel@laptop:~$
```




Contenido

- 1 Introducción
- 2 Primeros Pasos
- 3 Tipos de Datos**
- 4 Entrada/Salida
- 5 Control de Flujo
- 6 Módulos
- 7 Excepciones
- 8 POO
- 9 Ficheros
- 10 Interfaces gráficas



Tipos de Datos

Python tiene los siguientes tipos de datos:

- Boolean → Booleanos (True - False)
- Int → Números enteros
- Float → Números en coma flotante
- String → Cadenas de caracteres inmutables
- List → Listas (arrays) de elementos
- Tuple → Listas inmutables
- Set → Lista sin repeticiones
- Dict → Diccionarios (arrays asociativos)



Variables

- Se declaran al vuelo
- Son de tipado débil
- Para saber el tipo de una variable se puede usar la función `type()`

Example

```
>>> a = b = c = 1
>>> a = "ha!"
>>> a, b, c
("ha!", 1, 1)
>>> type(a)
<type 'str'>
```

Example

```
>>> variable = "Bu!"
>>> variable = 24
>>> variable
24
```



Números (enteros y flotantes)

- ¡Ojo a la conversión de tipos!

Operaciones básicas

```
>>> 1 + 1
2
>>> 5 / 2
2
>>> 5.0 / 2
2.5
```

Funciones matemáticas

```
>>> import math
>>> math.sqrt(4)
2.0
>>> math.log10(100)
2.0
>>> math.pi
3.1415926535897931
```



Cadenas de Caracteres I

- Da igual el tipo de comillas usadas (simples o dobles)
- A pesar de ser indexables, son **inmodificables**

Example

```
>>> cadena = "Hola!"
>>> cadena[0]
'H'
>>> cadena[0] = 'A'
[...] -> Error!
>>> len(cadena)
5
```

Example

```
>>> cadena1 = "Hola "
>>> cadena2 = "Mundo!"
>>> cadena1 + cadena2
'Hola Mundo!'
>>> cadena1 * 2
'Hola Hola '
```



- Podemos manejar sub-grupos de cadenas

Example

```
>>> cadena = "Hola!"
>>> cadena[0] + cadena[1]*10 + cadena[2:]
'Hoooooooooolola!'
>>> cadena = """
Esto tiene varias lineas
"""
>>> cadena
'\nEsto tiene varias lineas\n'
```



- Pueden contener elementos de distinto tipo
- Se accede a ellas mediante un índice numérico

Example

```
>>> lista = ["Una Cadena", 1.5, [1998, "Bu!"]]
>>> lista[2][0]
1998
>>> lista = ["A", "B", "D", "E"]
>>> lista[1] = "C"
>>> lista[0:2]
['A', 'C']
```

Lo mismo, pero sin poder modificarlas :)

- Se crean con paréntesis.

Example

```
>>> tupla = ("hola", 2)
>>> tupla
('Hola', 2)
>>> tupla[0] = "Adios"
[...] -> Error!
```




- Contienen conjuntos sin repetición y desordenados
- Se pueden ejecutar operaciones de conjuntos

Example

```
>>> vocales = set("aeiou")
>>> var = set("murcielago")
>>> var - vocales
set(['c', 'r', 'm', 'l', 'g'])
>>> vocales.issubset(var)
True
```



- Son arrays asociativos (par clave \rightarrow valor)
- Las claves han de ser objetos inmutables (números, cadenas, tuplas, ...)

Example

```
>>> pagado = {"Jaime" : 5000, "Paco" : 1754}
>>> pagado["Jaime"]
5000
>>> "Paco" in pagado
True
>>> pagado.keys()
['Jaime', 'Paco']
```



¿Qué creéis que hacen estas cosas?

Example

```
lista = [1, 2.0, "hey!", [5]]  
lista[1]  
lista[0] += 2  
lista.index(2.0)  
lista.insert(3,2)  
lista.count(2)  
lista[lista.index([5])].append([1,2,3])  
dict([('Nombre', 'Juan'), ('Curso', '4!')])
```



¿Qué creéis que hacen estas cosas?

Example

```
lista = [1, 2.0, "hey!", [5]]  
lista[1]  
lista[0] += 2  
lista.index(2.0)  
lista.insert(3,2)  
lista.count(2)  
lista[lista.index([5])].append([1,2,3])  
dict([('Nombre', 'Juan'), ('Curso', '4!')])
```



¿Qué creéis que hacen estas cosas?

Example

```
lista = [1, 2.0, "hey!", [5]]  
lista[1]  
lista[0] += 2  
lista.index(2.0)  
lista.insert(3,2)  
lista.count(2)  
lista[lista.index([5])].append([1,2,3])  
dict([('Nombre', 'Juan'), ('Curso', '4!')])
```



¿Qué creéis que hacen estas cosas?

Example

```
lista = [1, 2.0, "hey!", [5]]  
lista[1]  
lista[0] += 2  
lista.index(2.0)  
lista.insert(3,2)  
lista.count(2)  
lista[lista.index([5])].append([1,2,3])  
dict([('Nombre', 'Juan'), ('Curso', '4!')])
```



¿Qué creéis que hacen estas cosas?

Example

```
lista = [1, 2.0, "hey!", [5]]  
lista[1]  
lista[0] += 2  
lista.index(2.0)  
lista.insert(3,2)  
lista.count(2)  
lista[lista.index([5])].append([1,2,3])  
dict([('Nombre', 'Juan'), ('Curso', '4!')])
```



¿Qué creéis que hacen estas cosas?

Example

```
lista = [1, 2.0, "hey!", [5]]  
lista[1]  
lista[0] += 2  
lista.index(2.0)  
lista.insert(3,2)  
lista.count(2)  
lista[lista.index([5])].append([1,2,3])  
dict([('Nombre', 'Juan'), ('Curso', '4!')])
```




¿Qué creéis que hacen estas cosas?

Example

```
lista = [1, 2.0, "hey!", [5]]  
lista[1]  
lista[0] += 2  
lista.index(2.0)  
lista.insert(3,2)  
lista.count(2)  
lista[lista.index([5])].append([1,2,3])  
dict([('Nombre', 'Juan'), ('Curso', '4!')])
```



¿Qué creéis que hacen estas cosas?

Example

```
lista = [1, 2.0, "hey!", [5]]  
lista[1]  
lista[0] += 2  
lista.index(2.0)  
lista.insert(3,2)  
lista.count(2)  
lista[lista.index([5])].append([1,2,3])  
dict([('Nombre', 'Juan'), ('Curso', '4!')])
```



Contenido

- 1 Introducción
- 2 Primeros Pasos
- 3 Tipos de Datos
- 4 Entrada/Salida**
- 5 Control de Flujo
- 6 Módulos
- 7 Excepciones
- 8 POO
- 9 Ficheros
- 10 Interfaces gráficas



- La sentencia estándar de salida es print
- Para dar formato con print se usa el símbolo %

Example

```
>>> print "Holaaa"
Holaaa
>>> n = 5
>>> m = 3
>>> print "%d x %d = %d" % (n, m, n*m)
5 x 3 = 15
>>> nombre = "Anastasio"
>>> print "Hola, %s!" % (nombre)
Hola, Anastasio!
```



- Para coger datos por entrada estándar se usan las funciones `input()` para enteros y `raw_input()` para strings

Example

```
>>> a = input()  
5  
>>> type(a)  
<type 'int'>  
>>> print a*3  
15
```

Example

```
>>> b = raw_input()  
9  
>>> type(b)  
<type 'str'>  
>>> print b*3  
999
```



Contenido

- 1 Introducción
- 2 Primeros Pasos
- 3 Tipos de Datos
- 4 Entrada/Salida
- 5 Control de Flujo**
- 6 Módulos
- 7 Excepciones
- 8 POO
- 9 Ficheros
- 10 Interfaces gráficas



- En Python no hay llaves, todo se **indenta**

Sintaxis

```
if <condicion>:  
    bloque_1  
elif <condicion2>:  
    bloque_2  
else:  
    bloque_3
```

Example

```
>>> llave = "azul"  
>>> if llave == "azul":  
...     print "Es Azul"  
... else:  
...     print "Ni idea"  
...  
Es Azul
```



- Se itera sobre una secuencia, no sobre un número
- En los bucles de Python también tenemos **break** y **continue**

Sintaxis

```
for <var. iteración> in <secuencia>:  
    cuerpo
```

Example

```
>>> llave = ["azul", "roja"]  
>>> for color in llave:  
...     print "La llave puede ser " + color  
...  
La llave puede ser azul  
La llave puede ser roja
```




Oye! yo quiero hacer el **for** de toda la vida...

Example

```
>>> for i in range(1,10):  
...     print i,  
...  
1 2 3 4 5 6 7 8 9
```

■ Atención a la coma!!!

Example

```
for n, e in enumerate(lista):  
    print n, e
```



WHILE

- Exactamente igual que el while de cualquier otro lenguaje

Sintaxis

```
while condicion:  
    cuerpo
```

Example

```
while True:  
    print "Esto es un bucle infinito"
```



- Los argumentos se pasan por valor
- La primera línea puede ser texto de ayuda (comentario)

Sintaxis

```
def nombre(args):  
    cuerpo
```

Example

```
>>> def sumar(num1, num2):  
...     """ Ayuda """  
...     return num1 + num2  
...  
>>> sumar(3, 4)  
7
```



- Podemos dar valores predeterminados a los argumentos
- No es necesario que devuelvan un valor, ni que éste sea de un determinado tipo

Sintaxis

```
>>> def saluda(nombre = "Paco") :  
...     print "Hola " + nombre  
...  
>>> saluda("Pepe")  
Hola Pepe  
>>> saluda()  
Hola Paco
```



Contenido

- 1 Introducción
- 2 Primeros Pasos
- 3 Tipos de Datos
- 4 Entrada/Salida
- 5 Control de Flujo
- 6 Módulos**
- 7 Excepciones
- 8 POO
- 9 Ficheros
- 10 Interfaces gráficas



- Llamaremos módulo a un fichero con declaraciones
- Podremos importarlo mediante **import**

Fichero "saludar.py"

```
def saluda(nombre = "Paco") :  
    print "Hola " + nombre
```

Example

```
>>> import salutar  
>>> salutar.saluda("Manolo")  
Hola Manolo
```



- También podremos importar solo ciertas declaraciones; y hacerlo de manera interna

Example

```
>>> from saludar import saluda
>>> saluda("Manolo")
Hola Manolo
```

- Podríamos usar **from saludar import *** para importar todas las declaraciones del módulo



- ¿Qué métodos podemos ejecutar en X objeto?
- ¿Qué declaraciones contiene un módulo?

Example

```
>>> import saludar
>>> dir(saludar)
['__builtins__', '__doc__', .... , 'saluda']
>>> cadena = "Hola"
>>> dir(cadena)
[...]
```

¡**dir()** nos facilita eso y mucho más!



Contenido

- 1 Introducción
- 2 Primeros Pasos
- 3 Tipos de Datos
- 4 Entrada/Salida
- 5 Control de Flujo
- 6 Módulos
- 7 Excepciones**
- 8 POO
- 9 Ficheros
- 10 Interfaces gráficas



- Son errores lanzados en tiempo de ejecución.
- Python proporciona herramientas para manejarlos de forma controlada.

Example

```
>>> lista = ['a', 'b', 'c']
>>> try:
>>>     print lista[3]
>>> except IndexError:
>>>     print "Error de acceso al array."
>>> except:
>>>     print "Error desconocido."
Error de acceso al array.
```



Contenido

- 1 Introducción
- 2 Primeros Pasos
- 3 Tipos de Datos
- 4 Entrada/Salida
- 5 Control de Flujo
- 6 Módulos
- 7 Excepciones
- 8 POO**
- 9 Ficheros
- 10 Interfaces gráficas



¿Qué es eso de la POO?

- Programación **O**rientada a **O**bjetos
- Un nuevo paradigma de la programación
- Está orientado a modelizar el problema

Pues sigo sin enterarme de nada...

- Clases → Nuevo tipo de dato, "plantilla"
- Atributos → Estado interno de la clase
- Métodos → Define el comportamiento de la clase
- Objetos → Instancias de una clase

Con un ejemplo lo veremos mejor

- Clase → Coche
- Atributos → color, gasolina, modelo, potencia, arrancado...
- Métodos → arrancar(), conducir(), repostar()...
- Objeto → mi_coche, tu_coche...





- El constructor se llama `__init__()`
- Los métodos reciben un primer argumento adicional: `self`

Sintaxis

```
class Nombre (args1):  
    """Ayuda"""  
    atributo = valor  
    def __init__(self, args1):  
        cuerpo  
    def metodo(self, args2):  
        cuerpo
```



Encapsulación y Properties

- Realmente **no** existe el concepto de encapsulación
- Para emularlo, usaremos atributos que empiecen con **--**
- Para modificar atributos, podemos usar Properties

Sintaxis

```
class Nombre[(args1)]:  
    """Ayuda"""  
    __atributoPrivado = valor  
    def setAtributo(self, arg):  
        self.__atributoPrivado = arg  
    def getAtributo(self):  
        return self.__atributoPrivado  
    atributo = property(getAtributo, setAtributo)
```



- Ponemos la clase de la que se hereda entre parentesis
- No tiene porque ser solo una

Sintaxis

```
class Principal():  
    def __init__(self):  
        cuerpo  
class Derivada(Principal):  
    def __init__(self):  
        Principal.__init__(self)
```




Instanciando la Clase

- Cada objeto es independiente del resto

Sintaxis

```
objeto = nombreClase(args)
objeto.metodo(args)
objeto.atributo = valor
```

Veamos un ejemplo...



Contenido

- 1 Introducción
- 2 Primeros Pasos
- 3 Tipos de Datos
- 4 Entrada/Salida
- 5 Control de Flujo
- 6 Módulos
- 7 Excepciones
- 8 POO
- 9 Ficheros**
- 10 Interfaces gráficas



Para manejar ficheros deberemos de hacer lo siguiente:

- Abrir el fichero con algún modo (lectura, escritura, ...).
- Realizar la operación (lectura, escritura, despl, ...).
- Cerrar el fichero.

Abrir/Cerrar un fichero

```
>>> fichero = open(path, modo)
[...]  
>>> fichero.close()
```



Tenemos varios métodos de lectura:

- `read(size)` → Lee size caracteres
- `readline()` → Lee una línea del fichero
- `readlines()` → Lee el fichero entero metiéndolo en una lista

Example

```
>>> fichero = open('/etc/X11/xorg.conf', 'r')
>>> fichero.readlines()
[...]
>>> fichero.readlines()
[]
>>> fichero.close()
```



Escribiendo Ficheros

Para escribir contenido en un fichero:

- Hemos de haber abierto el fichero en modo escritura
- `fichero.write(valor)` → Escribe en el fichero el contenido pasado como argumento

Example

```
>>> fichero = open("hola.txt", "r+")
>>> fichero.write("Hola Mundo!")
>>> fichero.tell()
11L
>>> fichero.seek(0)
>>> fichero.readline()
'Hola Mundo!'
```



- Podemos guardar una estructura en un fichero.
- Necesitamos usar pickle: `import pickle`

Example

```
>>> import pickle
>>> fichero = open("guardar.txt", "r+")
>>> lista = ["Cadena", 234]
>>> pickle.dump(lista, fichero)
>>> fichero.seek(0)
>>> lista2 = pickle.load(fichero)
>>> lista2
["Cadena", 234]
```



Contenido

- 1 Introducción
- 2 Primeros Pasos
- 3 Tipos de Datos
- 4 Entrada/Salida
- 5 Control de Flujo
- 6 Módulos
- 7 Excepciones
- 8 POO
- 9 Ficheros
- 10 Interfaces gráficas**



¿Qué es PyQt?

¿Qué es Qt?

- Una biblioteca de desarrollo multiplataforma.
- Para desarrollar interfaces gráficas, principalmente.

¿Qué es PyQt?

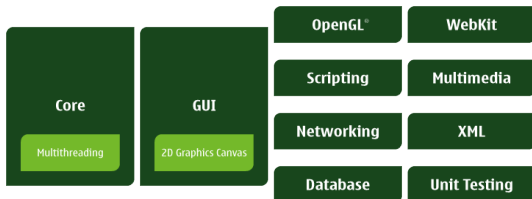
- Es un binding de Qt para python.





¿Qué nos ofrece?

- Dispone de una gran cantidad de recursos.
- Están divididos en módulos





Example

```
import sys
from PyQt4 import QtGui

app = QtGui.QApplication(sys.argv)

widget = QtGui.QWidget()
widget.resize(300, 200)
widget.setWindowTitle("Ejemplo")
widget.show()

app.exec_()
```



Primer ejemplo II

Example

```
app = QtGui.QApplication(sys.argv)
```

- Toda aplicación en PyQt debe tener un objeto `QApplication`.

Example

```
app.exec_()
```

- Finaliza el bucle.



- Es posible crear interfaces de forma rápida.
- Se puede transformar los archivos generados a un archivo de Python:

Example

```
pyuic mi_archivo.ui > mi_archivo.py
```



En Resumen...



- Python es un lenguaje muy completo y potente
- Aún así, es muy fácil programar y desarrollar con él de una forma muy rápida
- Aunque no lo hayamos visto, tiene una **enorme** cantidad de módulos

Estructuras de Datos:

- [] → Listas
- () → Tuplas
- {} → Diccionarios



¿ PREGUNTAS ?





Gracias

**MUCHAS GRACIAS POR VENIR
:)**