Python

Tutor: Ignacio Pérez Martín.

Curso: Programación Python y HTML-5. CEP Marbella-Coín.

Resumen

Este tema pretende que comprendáis la filosofía de Python y su sintaxis, os capacitará para realizar páginas en Django (siguiente tema) y scripts complejos. Para una consulta más profunda del lenguaje he dejado en las referencias un libro completo, "Python para todos" [1], el cual recomiendo ojear; y para consultas concretas de funciones y librerías consultad la web oficial de Python [2]. En este tema doy por supuesto que todos tenéis conocimientos de programación.

Índice

1. Introducción

Al fin llegamos al lenguaje Python, del que muchos habéis oído hablar por su claridad y flexibilidad.

Python es un lenguaje de programación creado **alrededor de 1990 por Guido van Rossum** y cuyo nombre, al contrario de lo que parece, no se refiere a la serpiente pitón sino que hace referencia al grupo humorístico británico **Monty Python**.

No puedo resistirme a poneros ya algo de código, voy a empezar sorprendiendo con la simpleza del clásico "*Hola mundo*". Mientras que en Java es algo como:

```
class HolaMundo {
    public static void main (String args[]) {
        System.out.print("¡Hola mundo!");
    }
}
```

En Python es simplemente:

```
print "¡Hola mundo!"
```

Como véis ya empieza a demostrar su claridad.

Una de las características sintácticas que más llaman la atención sobre Python es que no usa ningún tipo de etiqueta ni delimitadores de bloque como llaves ({ }) o begin-end, simplemente hace uso del sangrado del código o **indentación** para reconocer el anidamiento. Esto hace que el código de un programa Python deba estar correctamente sangrado y lo hace fácilmente légible a la fuerza. Tampoco necesita de caracteres de finalización de línea como el punto y coma (;) de otros lenguajes.

Veamos un ejemplo de código Pyhon donde al eliminar caracteres innecesarios nos fijamos mejor en las líneas de programación importantes:

```
def factorial(x):
    if x == 0:
        return 1
    return x * factorial(x - 1)
```

La primera línea define una función *factorial* que recibe un argumento *x*. Python utiliza **tipado dinámico** por lo que no hay que especificar el tipo de las variables, esa *x* puede ser cualquier cosa a la hora de llamar a la función.

La segunda línea al estar indentada ya indica que pertenece a la función, lo mismo ocurre con la tercera línea, está dentro del *if*.

Como el tipado es dinámico se pueden hacer asignaciones como éstas:

```
a = 2
a = "Hola"
```

También asignar varias a la vez:

```
a, b = "hola", 33
```

Otra de las buenas características de Python, pero no exclusiva, es el disponer de un **modo interactivo** o *shell* donde poder escribir las expresiones o probar nuestro código. Simplemente con ejecutar el comando *python* ya tenemos el shell abierto:

```
$ python
Python 2.7.1 (r271:86832, Jun 16 2011, 16:59:05)
[GCC 4.2.1 (Based on Apple Inc. build 5658) (LLVM build 2335.15.00)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Y ya solo habría que ir introduciendo el código:

```
>>> def factorial(x):
...     if x == 0:
...         return 1
...     return x * factorial(x - 1)
...
>>> factorial(5)
120
>>> f = factorial(5)
>>> print f
120
```

1.1. Instalación de Python

Recomiendo usar Python para ir haciendo pruebas conforme se lee el tema, los usuarios de Linux y Mac OSX ya lo tendrán instalado, y para los usuarios de Windows podéis descargar el instalador aquí [1]. Además será necesario tenerlo instalado para el tema de Django, a ser posible de la versión Python 2.5 en adelante para poder utilizar la última versión de Django. Sino al menos Python 2.4. Si tenéis dudas sobre la instalación preguntad en el foro.

El modo interactivo por defecto de Python está bien pero recomiendo utilizar *ipython* [4] que está aún mejor, con autocompletado. Si disponéis de *easy_install* su instalación es tan simple como poner:

```
$ easy_install ipython
```

O aún mejor recomiendo instalar antes *pip*, que es otro instalador de paquetes de Python:

```
$ easy_install pip
$ pip install ipython
```

2. Listas, tuplas, conjuntos y diccionarios

Los elementos iterables como las listas, tuplas, conjuntos y diccionarios son los tipos de datos por excelencia en Python, muy similares a los arrays de otros lenguajes.

2.1. Listas

En Python una lista puede contener cualquier tipo de datos y no tienen por qué ser del mismo tipo. Veamos una interacción con el shell de Python para entender su funcionamiento:

```
>>> lista = [1, "hola", 3, 4]
>>> lista[0]
1
>>> lista[-1]
4
>>> lista[1:3]
['hola', 3]
>>> lista[1:]
['hola', 3, 4]
>>> lista[:3]
['hola', 3]
>>> lista[:]
[1, 'hola', 3, 4]
>>> lista[1]
'hola'
>>> lista[1][0]
'h'
>>> lista[1][1:]
'ola'
>>>
>>> lista.append(5)
>>> lista
[1, 'hola', 3, 4, 5]
>>> lista + [6]
[1, 'hola', 3, 4, 5, 6]
>>> lista
[1, 'hola', 3, 4, 5]
>>>
>>> lista[0] = ['a','b', 'c']
>>> lista
[['a', 'b', 'c'], 'hola', 3, 4, 5]
>>> lista.remove(3)
>>> lista
[['a', 'b', 'c'], 'hola', 3, 5]
>>> abc = lista[0] + ['d', 'e']
>>> abc
['a', 'b', 'c', 'd', 'e']
>>>
>>> len(abc)
5
>>> abc.reverse()
>>> abc
['e', 'd', 'c', 'b', 'a']
>>> abc.sort()
>>> abc
```

```
['a', 'b', 'c', 'd', 'e']
>>>
>>> lista = ['a', 'a', 'c']
>>> lista.count('a')
2
>>> 3 * ['a']
['a', 'a', 'a']
```

Algunas operaciones más sobre listas (**split**, **strip** y **join**):

```
>>> cadena = "Hola, como estas, Juan"
>>> lista = cadena.split(",")
>>> lista
['Hola', ' como estas', ' Juan']
>>> lista[1]
' como estas'
>>> lista[1].strip()
'como estas'
>>> lista = map(lambda x: x.strip(), lista)
>>> lista
['Hola', 'como estas', 'Juan']
>>> ", ".join(lista)
'Hola, como estas, Juan'
```

2.2. Tuplas

Las tuplas son iguales que las listas en cuanto a acceso pero en una tupla no se permite insertar. eliminar, ni reasignar elementos. Veamos un ejemplo:

```
>>> tupla = (1, "dj", 3)
>>> tupla
(1, 'dj', 3)
>>> len(tupla)
3
>>> tupla[2]
3
>>> tupla + (4,)
(1, 'dj', 3, 4)
>>> tupla
(1, 'dj', 3)
>>> tupla = tupla + (4,)
>>> tupla
(1, 'dj', 3, 4)
>>> tupla = (1)
>>> tupla
1
```

```
>>> tupla = (1,)
>>> tupla
(1,)
```

2.3. Conjuntos

Los conjuntos, al igual que en álgebra, son agrupaciones de elementos no ordenados donde no se permiten elementos repetidos y que permiten realizar operaciones de grupo como unión, intersección y diferencia. Veámoslo con un ejemplo:

```
>>> conjunto1 = set([1, 2, 3])
>>> conjunto1
set([1, 2, 3])
>>> len(conjunto1)
3
>>> conjunto2 = set([3, 4 ,5])
>>> conjunto1 | conjunto2
set([1, 2, 3, 4, 5])
>>> conjunto1 & conjunto2
set([3])
>>> conjunto1 - conjunto2
set([1, 2])
```

Vamos a ver ahora como uno de los usos más comunes de los conjuntos es eliminar los elementos repetidos de una lista:

```
>>> lista = [1,2,2,3,4,4,4,5]
>>> conjunto = set(lista)
>>> conjunto
set([1, 2, 3, 4, 5])
>>> limpia = list(conjunto)
>>> limpia
[1, 2, 3, 4, 5]
>>>
>>> list(set(lista))
[1, 2, 3, 4, 5]
```

2.4. Diccionarios

Un diccionario es una colección de datos, no ordenados, donde cada dato tiene asociada una clave única. Los datos se pueden modificar pero no así la clave. Veamos algunos ejemplos de uso:

```
>>> diccionario = {1: 'hola', 'b':'adios', 3:[1,2,3]}
>>> len(diccionario)
3
>>> diccionario
{3: [1, 2, 3], 1: 'hdiola', 'b': 'adios'}
```

```
>>> diccionario[1]
'hola'
>>> diccionario['b']
'adios'
>>> diccionario['nueva clave'] = 4
>>> diccionario
{3: [1, 2, 3], 1: 'hola', 'b': 'adios', 'nueva clave': 4}
>>> diccionario['nueva clave'] = 1
>>> diccionario
{3: [1, 2, 3], 1: 'hola', 'b': 'adios', 'nueva clave': 1}
>>> diccionario[3][1]
2
>>>
>>> diccionario.keys()
[3, 1, 'b', 'nueva clave']
>>> diccionario.values()
[[1, 2, 3], 'hola', 'adios', 1]
>>> diccionario.items()
[(1, 'hola'), (3, [1, 2, 3]), ('b', 'adios'), ('nueva clave', 1)]
>>> diccionario.has_key('b')
True
>>> diccionario.get('b')
'adios'
>>> diccionario.get('z', 'Devuelve esto si no existe la clave')
'Devuelve esto si no existe la clave'
```

3. Condicionales y bucles

3.1. Sentencia if

La construcción de la sentencia if en Python es similar a la de otros lenguajes, permite varios o ningún else-if, que se escribe como *elif,* y uno o ningún *else*.

```
# Esto es un comentario en Python
if opcion == "algo":
        print "algo"
elif opcion == "otra cosa":
        print opcion
else:
        print "nada"
```

Y también permite una construcción en una sola línea del tipo:

```
>>> print "ok" if True else "no ok"
ok
>>> print "ok" if False else "no ok"
```

```
no ok
>>> # O bien
... a = "ok" if True else "nok"
>>> a
'ok'
>>> # O su versión más simple
... "ok" if True else "nok"
'ok'
```

Aprovechamos que estamos hablando de sentencias condicionales para exponer algunas operaciones booleanas:

```
>>> 1 > 10
False
>>> 1 in [1, 2, 3]
True
>>> 1 in [[1,2], 2, 3]
False
>>> not 1
False
>>> not 0
True
>>> not -1
False
>>> sin_valor = None
>>> if sin_valor: print "esto no"
...
>>> if not sin_valor: print "ahora si"
...
ahora si
```

3.2. While

```
>>> i = 0
>>> while i < 3:
...     print i
...     i += 1
...
0
1
2
```

3.3. For

Sin duda el bucle for será lo que más utilicéis a la hora de programar en Python ya que puede iterar sobre cualquier elemento iterable: listas, tuplas, resultados de consultas...

```
>>> personas = ['pepe', 'manolo', 'sara']
>>> for p in personas:
...     print p
...
pepe
manolo
sara
```

Compliquemos un poco más el código:

```
>>> personas = [
...     {'nombre': 'Pepe', 'edad': 30},
...     {'nombre': 'Manolo', 'edad': 40},
...     {'nombre': 'Sara', 'edad': 50},
...]
>>> for p in personas:
...     if p['edad'] > 40:
...         print "%s de %s años de edad." % (p['nombre'], p['edad'])
...
Sara de 50 años de edad.
```

Como véis se hace uso de *%s* para imprimir variables como cadenas dentro de otras, podríamos haber puesto también cualquiera de estas dos opciones:

```
>>> nombre = 'Sara'
>>> edad = 50
>>> print nombre, "de", edad, "años de edad."
Sara de 50 años de edad.
>>> print nombre + " de " + str(edad) + " años de edad."
Sara de 50 años de edad.
```

Podemos hacer uso también de generadores de cadenas como range:

```
>>> range(3)
[0, 1, 2]
>>> for n in range(1, 3):
...     print n
...
1
2
```

O utilizar for para constriur cadenas:

```
>>> [2*n for n in range(3)]
[0, 2, 4]
>>> personas = [
...     {'nombre': 'Pepe', 'edad': 30},
...     {'nombre': 'Manolo', 'edad': 40},
...     {'nombre': 'Sara', 'edad': 50},
...]
```

```
>>> [p['edad'] for p in personas]
[30, 40, 50]
```

4. Funciones

Las funciones se definen con la palabra *def* seguida del nombre de la función y los parámetros entre paréntesis, si tiene. Veamos como ejemplo la función factorial con la que empezamos:

```
def factorial(x):
    if x == 0:
        return 1
    return x * factorial(x - 1)
```

Voy a aprovechar para avanzar un poco más y explicar como se organiza el código Python en módulos que pueden ser importados. Imaginaos un fichero llamado *sucesiones.py* con el contenido:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

def fibonacci(n):
        if n == 0:
                return 0
        elif n == 1:
                return 1
        return fibonacci(n-1) + fibonacci(n-2)

def sucesion_fibonacci(numero):
        """ Calcula la sucesión de fibonacci de 'numero' elementos. """
        return [fibonacci(n) for n in range(0, numero)]
```

Las primeras dos líneas no son obligatorias pero sí convenientes, indican la ruta del intérprete Python y el encoding del fichero. La primera línea de la función *sucesion_fibonacci* entre triples dobles comillas indica el texto que aparecerá cuando busquemos información sobre esta función.

Ahora, si en el mismo directorio que el fichero anterior ejecutamos:

```
>>> from sucesiones import sucesion_fibonacci
>>> sucesion_fibonacci(10)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
>>>
>>> help(sucesion_fibonacci)
sucesion_fibonacci(numero)
    Calcula la sucesión de fibonacci de 'numero' elementos.
```

También podríamos importar las dos funciones:

```
from sucesiones import fibonacci, sucesion_fibonacci
```

Todo:

```
from sucesiones import *
```

O solo el nombre del módulo:

```
>>> import sucesiones
>>> sucesiones.sucesion_fibonacci(10)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Volviendo al tema de las funciones, podemos especificar un valor por defecto en una función de la forma:

```
>>> def saludo(nombre="Juan"):
...     print "Hola %s" % nombre
...
>>> saludo()
Hola Juan
>>> saludo("Pepe")
Hola Pepe
```

4.1. Paso de argumentos

Supongamos que tenemos un fichero con código Python llamado suma.py, podríamos ejecutar el fichero de la forma:

```
$ python suma.py 3 4
7
$ chmod +x suma.py
$ ./suma.py 3 4
7
```

Y cuyo código sea:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import sys

if len(sys.argv) < 3:
   print "Debe pasar dos argumentos"
else:
   try:
      print int(sys.argv[1]) + int(sys.argv[2])
   except ValueError:
      print "Debe pasar dos números"
```

4.2. Funciones lambda, map y filter

Una función ***lambda*** es una función anónima creada en una sola línea que se suele utilizar en conjunción con otras funciones como *map* y *filter*. ***Map*** aplica una función dada a todos los elementos de la colección pasada. ***Filter*** realiza un filtrado de la colección según cada elemento cumpla o no la función aplicada.

Por ejemplo:

```
>>> lambda x: x*2
<function <lambda> at 0x10c40fc08>
>>> lista = [1, 2, 3]
>>> map(lambda x: x*2, lista)
[2, 4, 6]
>>> filter(lambda x: x > 2, lista)
[3]
>>> # Podemos incluir unas dentro de otras...
... filter(lambda x: x > 2, map(lambda x: x*2, lista))
[4, 6]
```

5. Orientación a objetos: clases y herencia

5.1. Clases

Las clases en Python siguen la misma filosofía que en otros lenguajes orientados a objetos. Cojamos como ejemplo la clase que utiliza *Raúl González* en el libro *Python para todos*:

```
class Coche:
        """Abstraccion de los objetos coche."""
        def __init__(self, gasolina):
                self.gasolina = gasolina
                print "Tenemos", gasolina, "litros"
        def arrancar(self):
                if self.gasolina > 0:
                        print "Arranca"
                else:
                        print "No arranca"
        def conducir(self):
                if self.gasolina > 0:
                        self.gasolina -= 1
                        print "Quedan", self.gasolina, "litros"
                else:
                        print "No se mueve"
```

Y hagamos uso de ella:

```
>>> coche = Coche(2)
Tenemos 2 litros
>>> coche.arrancar()
Arranca
>>> coche.conducir()
Quedan 1 litros
>>> coche.conducir()
Quedan 0 litros
```

```
>>> coche.conducir()
No se mueve
```

5.2. Herencia

A diferencia de otros lenguajes Python permite la herencia múltiple. Veamos un ejemplo de herencia simple y múltiple:

```
class Terrestre:
        def desplazar(self):
                print "El animal anda"

class Acuatico:
        def desplazar(self):
                print "El animal nada"

class Pez(Acuatico):
        pass

class Cocodrilo(Terrestre, Acuatico):
        pass

>>> pez = Pez()
>>> pez.desplazar()
El animal nada
>>>
>>> cocodrilo = Cocodrilo()
>>> cocodrilo.desplazar()
El animal anda
```

En el *cocodrilo* se utiliza el comportamiento de *Terrestre* al estar más a la izquierda en la definición de la clase.

La sentencia *pass* de Python es como unas llaves vacías ({}) en C o Java. Se usa porque Python, al no utilizar delimitadores de bloque, no puede saber si está vacío o mal indentado.

6. Trabajar con fechas

Mostraré algunas de las operaciones más comunes sobre fechas, que os puedan servir.

```
>>> import datetime
>>> datetime.date.today()
datetime.date(2012, 5, 10)
>>> hoy = datetime.date.today()
>>> hoy.day
10
>>> hoy.month
```

```
5
>>> hoy.year
2012
>>> datetime.datetime.now()
datetime.datetime(2012, 5, 10, 21, 40, 0, 779620)
>>> ahora = datetime.datetime.now()
>>> ahora.hour
21
>>> manana = hoy + datetime.timedelta(days=1)
>>> hoy < manana
True
>>> manana - hoy
datetime.timedelta(1)

>>> import calendar
>>> # Primer dia de la semana y ultimo dia de un mes dado
... # El 2 es el martes y el último día de febrero de 2012 es el 29.
... calendar.monthrange(2012, 2)
(2, 29)
>>> (primer_dia_semana, ult_dia) = calendar.monthrange(2012, 2)
```

7. Tratamiento de ficheros

A continuación expongo algunos de los usos básicos de Python en cuanto a tratamiento de ficheros.

Lista con todos los ficheros de un directorio:

```
>>> import os
>>> ficheros = os.listdir('/home/usuario/dir')
['archivo.py', 'directorio']
```

Lista con todos los ficheros con una extensión dada:

```
>>> import glob
>>> glob.glob("*.py")
['sucesiones.py']
```

Discriminar entre directorio, fichero o enlace:

```
>>> import os
>>> for fichero in os.listdir('/home/usuario/dir'):
...     if os.path.isdir(fichero):
...         print 'd', fichero
...     elif os.path.isfile(fichero):
...         print 'f', fichero
...     elif os.path.islink(fichero):
...         print 'l', fichero
```

```
...
f archivo.py
d directorio
```

Lectura y escritura de ficheros:

```
>>> entrada = open("entrada.txt")
>>> salida = open("salida.txt")
>>> for linea in entrada:
...     salida.write(linea)
...
>>> entrada.close()
>>> salida.close()
```

8. Control de errores

La captura de errores se realiza a través de las sentencias *try/except*. Veamos un par de ejemplos:

```
>>> a = 1 * b
Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
NameError: name 'b' is not defined
>>>
>>> try:
...     a = 1 * b
... except NameError:
...     print "Error"
...
Error
```

```
>>> a = dict()
>>> a[1]
Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
KeyError: 1
>>>
>>> try:
...     a[1]
... except NameError:
...     print "No es de este tipo"
... except KeyError:
...     print "De éste sí"
... except:
...     print "Resto de casos"
...
De éste sí
```

9. Huevos de Python

Como buena serpiente Python pone huevos... Bueno, no es ese el motivo pero sí que viene de ahí su nombre. No quería terminar el tema sin explicar qué son los huevos de Python (Python eggs).

Un huevo Python no es más que módulos de Python empaquetados de manera que facilita su instalación, parecido al JAR de Java. En él se incluye información diversa como los módulos de los que depende, la documentación relativa al software o la versión de la que se trata, de manera que las dependencias del proyecto puedan ser comprobadas y satisfechas en el momento de la ejecución.

El gestor de paquetes por excelencia es *easy_install* o su sucesor *pip*. Con ellos instalamos, borramos y actualizamos los eggs de Python. En el tema siguiente instalaremos Django como un huevo más de Python.

Referencias

[1] Tutorial de Python "Python para todos": `http://mundogeek.net/tutorial-python/`.

[2] Web oficial de Python: `http://python.org`.

[3] Instalación de Python: `http://www.python.org/getit/`.

[4] Web de IPython: `http://ipython.org/`.