Introducción a Python

Evento Python Madrid 2011

@franchukelly

Python Madrid

Python Madrid

Grupo de entusiastas de Python ubicados en Madrid.

Reuniones mensuales en las que se realizan un par de charlas rápidas y se discute la organización de eventos como éste.

Lo podéis encontrar en:

- http://groups.google.es/group/python-madrid
- http://twitter.com/python_madrid

¿Qué es Python y por qué usarlo?

¿Qué es Python?

Un lenguaje de programación.

Libre y gratuito.

Creado por Guido van Rossum.

Controlado por la Python Software Foundation.

Actualmente hay dos ramas de desarrollo:

- La 2.x (cuya última versión es la 2.7.2)
- La 3.x (cuya última versión es la 3.2)
- Me centraré en la versión 2.7.2

¡Cuidado! La rama 3.x tiene muchos cambios que rompen la compatibilidad con la rama 2.x.

- http://wiki.python.org/moin/Python2orPython3

¿Quién usa Python?

Algunos ejemplos de compañías y/o entidades que lo usan:

- Bea Systems
- Google
- Hewlett-Packard
- IBM
- Microsoft
- NASA
- Red Hat
- Walt Disney Company
- Xerox
- Yahoo!

Más información en: http://www.python.org/about/success/

¿Por qué usar Python?

Fácil de leer y escribir.

Es un lenguaje de alto nivel.

Dispone de orientación a objetos.

- Con herencia múltiple.

Control de excepciones.

Introspección.

Dinámico.

Multi-plataforma.

El intérprete

El intérprete

Existen varias implementaciones:

- CPython realizada en C.
- PyPy realizada en Python.
- Jython realizada en Java.

La implementación que actualmente usa Python es Cpython.

El intérprete permite ejecutar código de Python de forma similar a cómo funciona cualquier consola.

```
$ python
Python 2.7.1+ (default, Apr 20 2011, 22:33:39)
[GCC 4.5.2] on linux2
Type "help", "copyright", "credits" or "license" for more
information.
>>> print 'Hola panda'
Hola panda
```

El intérprete

También ejecutará cualquier archivo con código de Python.

```
$ python hola-panda.py
Hola panda
```

Los archivos de Python pueden contener una cabecera especial para que los ejecute el intérprete (sólo en sistemas UNIX y Linux).

```
#!/usr/bin/env python
```

También pueden contener otra cabecera con la codificación del archivo.

```
# -*- coding: utf-8 -*-
```

Sintaxis básica

Sintaxis básica

No existe un delimitador, como el clásico punto y coma, para la ejecución de instrucciones.

```
print 'Hola mundo'
print 'Hello world'
```

Los bloques de código se definen mediante su nivel de tabulación.

```
bloque-1
    bloque-2

bloque-1
```

Los comentarios empiezan por una almohadilla.

```
# Soy un comentario
```

Sintaxis básica

Las variables no necesitan ningún tipo de identificador especial, como su tipo, var o $.

```
variable = valor
```

Diferencia entre minúsculas y mayúsculas.

```
>>> saludo = 'hola'
>>> print saludo
hola
>>> print SaLuDo
NameError: name 'SaLuDo' is not defined
```

Existe una propuesta para la guía de estilo de programación llamada PEP-8.

- http://python.org/dev/peps/pep-0008/

Tipos de datos

Tipos de datos

Python no es tipado, no se declara el tipo de datos de las variables.

Las variables se convierten al tipo deseado en base a lo que hacen, tipado dinámico.

Todos los tipos son objetos que tienen métodos asociados.

- Números.
- Booleanos.
- Cadenas de texto.
- Listas.
- Tuplas.
- Diccionarios.
- Conjuntos.
- Archivos.

Números

Existen varios tipos de números: enteros, decimales, imaginarios.

```
entero = 1
decimal = 1.25
imaginario = 3 + 4j
```

Disponen de las operaciones básicas:

- Suma.
- Resta.
- Multiplicación.
- División.
- Módulo.

Booleanos

Identifican los valores: verdad y falso.

```
verdad = True
falso = False
```

Los siguientes valores también se consideran falso:

- El valor especial `None`.
- El cero en cualquier tipo de número.
- Una secuencia vacía: una cadena de texto, una lista, ...
- Un diccionario vacío.

Disponen de las clásicas operaciones lógicas:

- Negación: `not`
- Y-lógico: `and`
- O-lógico: `or`
- O-exclusivo: `^`
- Igualdad y desigualdad: `==` y `!=`
- Mayor y menor: `>` y `<`

Cadenas de texto

Se identifican mediante comillas simples o dobles.

```
comillas_simples = 'hola'
comillas_dobles = “hola”
```

Acceso a los caracteres mediante su posición.

```
>>> saludo = 'hola'
>>> print saludo[0], saludo[3]
h a
```

Son inmutables.

```
>>> saludo = 'hola'
>>> saludo[0] = 'b'
TypeError: 'str' object does not support item assignment
```

Se pueden definir cadenas multi-línea con triples comillas.

```
>>> saludo = '''Hola
... mundo'''
>>> print saludo
Hola
mundo
```

Cadenas de texto

Posibilidad de usar UNICODE.

```
>>> saludo = u'¡hola niños!'
>>> print saludo
¡hola niños!
```

Concatenación con el operador de la suma.

```
>>> saludo = 'hola' + 'mundo'
>>> print saludo
hola mundo
```

Concatenación repetida con el operador de la multiplicación.

```
>>> saludo = 'hola' * 3
>>> print saludo
holaholahola
```

Unión de los elementos de una secuencia.

```
>>> lista_numeros = ', '.join (['a', 'b', 'c'])
>>> print lista_numeros
'a, b, c'
```

Listas

Un tipo particular de Python, similar a los arrays.

Se identifican mediante corchetes.

```
lista = []
lista = list ()
```

Puede contener elementos heterogéneos.

```
lista = [1, “dos”, True, [“hola”, “adios”]]
```

Acceso a los elementos mediante su posición.

```
>>> lista = [1, “dos”, True, [“hola”, “adios”]]
>>> print lista[0], lista[3][1]
1 adios
```

Son mutables.

```
>>> lista[3] = False
>>> print lista[3]
False
```

Tuplas

Un tipo particular de Python, es una secuencia de elementos.

Se identifican mediante paréntesis o elementos separados por comas.

```
tupla = ()
tupla = tuple ()
tupla = 1, 2, 3
```

Puede contener elementos heterogéneos.

```
tupla = (1, “dos”, True, [“hola”, “adios”])
```

Acceso a los elementos mediante su posición.

```
>>> tupla = (1, “dos”, True, [“hola”, “adios”])
>>> print tupla[0], tupla[3][1]
1 adios
```

Son inmutables.

```
>>> tupla = (1, “dos”, True, [“hola”, “adios”])
>>> tupla[0] = 'uno'
TypeError: 'tuple' object does not support item assignment
```

Diccionarios

Un tipo particular de Python, similar al `Map` de Java.

Se identifican mediante llaves.

```
diccionario = {}
diccionario = dict ()
```

Contiene pares de elementos clave-valor.

```
diccionario = {1: “uno”, “verdad”: True}
```

Acceso a los elementos mediante su clave.

```
>>> diccionario = {1: “uno”, “verdad”: True}
>>> print diccionario[1], diccionario[“verdad”]
uno True
```

Son mutables.

```
>>> diccionario[“verdad”] = False
>>> print diccionario[“verdad”]
False
```

Conjuntos

Un tipo particular de Python, similar a los conjuntos de elementos matemáticos.

Se identifican mediante la palabra `set`.

```
conjunto = set ()
```

No se puede acceder a los elementos mediante su posición.

```
>>> conjunto = set ([1, 2, 3])
>>> print conjunto[0]
TypeError: 'set' object does not support indexing
```

Son mutables, mediante los métodos `add` y `remove`.

```
>>> conjunto.add (4)
>>> print conjunto
set([1, 2, 3, 4])
>>> conjunto.remove (2)
>>> print conjunto
set([1, 3, 4])
```

Archivos

Se abren utilizando la función open.

```
archivo = open ('archivo.txt')
```

Se cierran utilizando la función `close`.

```
archivo.close ()
```

Se escribe en ellos utilizando la función `write`.

```
archivo.write ('cadena de texto a escribir')
```

Lectura de un número concreto de bytes.

```
archivo.read ()  # Se lee el archivo entero
archivo.read (bytes-a-leer)  # Se leen el número de bytes
```

Lectura por líneas.

```
archivo.readline ()  # Lee la línea actual
archivo.readlines ()  # Lee todas las líneas del archivo y
                      # devuelve una lista con las líneas
```

Operaciones con índices

Python permite realizar algunas operaciones “especiales” con los tipos de datos que disponen de índices.

Acceso invertido mediante índices negativos.

```
>>> lista = [0, 1, 2, 3, 4]
>>> print lista[-1], lista[-5]
4 0
```

Acceso a fragmentos mediante rangos.

```
>>> print lista[1:3]
[1, 2]
```

Los valores del rango son opcionales.

```
>>> print lista[:3]
[0, 1, 2]
>>> print lista[1:]
[1, 2, 3, 4]
>>> print lista[:]
[0, 1, 2, 3, 4]       # Es una copia de 'lista'
```

Control de flujo

Sentencias condicionales

Permiten ejecutar un bloque de código si se cumple la condición enunciada.

```
if condicion:
    # Haz algo

elif otra-condicion:
    # Haz otra cosa

else:
    # Haz lo que sea
```

A diferencia de la mayoría de lenguajes Python no dispone de una sentencia `switch`.

Bucle while

Permite ejecutar un bloque de código mientras se cumpla la condición enunciada.

```
while condicion:
    # Haz algo
```

Se puede añadir una sentencia `else` al bucle, que ejecutará el bloque de código cuando no se cumpla la condición enunciada en el bucle.

```
while condicion:
    # Haz algo mientras se cumpla 'condicion'
else:
    # Haz otra cosa cuando no se cumpla 'condicion'
```

Bucle for

Permite ejecutar un bloque de código iterando sobre los elementos de cualquier tipo de secuencia (p. ej. una lista).

```
for elemento in secuencia:
    # Haz algo
```

Se puede añadir una sentencia `else` al bucle, que ejecutará el bloque de código cuando se terminen los elementos de la secuencia.

```
for elemento in secuencia:
    # Haz algo mientras haya elementos
else:
    # Haz otra cosa cuando se terminen los elementos
```

Mediante la función `range` se puede utilizar un `for` más parecido al *clásico*.

```
for elemento in range (0, 5):
    # Haz algo mientras haya elementos
    # range (0, 5) = [0, 1, 2, 3, 4]
```

Comprensión de listas

Python dispone de un mecanismo llamado *comprensión de listas* para la creación de listas utilizando el bucle for.

```
# Números pares sin comprensión de listas
>>> numeros_pares = []
>>> for i in range (5):
...     numeros_pares.append (i * 2)
...
>>> print numeros_pares
[0, 2, 4, 6, 8]

# Números pares con comprensión de listas
>>> numeros_pares = [(i * 2) for i in range (5)]
>>> print numeros_pares
[0, 2, 4, 6, 8]
```

De hecho en la rama 3.x de Python se ha extendido este mecanismos a los diccionarios.

Instrucciones especiales

Al igual que en otros lenguajes, existen instrucciones especiales para controlar el flujo de los bucles:

- `break` sale del bucle sin pasar por el bloque del `else`.

```
while condicion:
    # Haz algo
    break
else:
    # Se ejecutará cuando no se cumpla la condicion

# Se ejecutará después del 'else' o después del 'break'
```

- `continue` salta a la siguiente iteración del bucle.
- `pass` es una instrucción que no hace nada, sirve para definir bloques de código vacíos, como bucles infinitos.

```
# Bucle infinito
while condicion:
    pass
```

Control de excepciones

Permite controlar la ejecución de una excepción en un bloque de código.

```
try:
    # Haz algo

except Excepcion as excepcion:
    # Se ejecutará al capturar la excepción

else:
    # Se ejecutará si no se captura ninguna excepción

finally:
    # Se ejecutará después de cualquiera de los bloques:
    # 'try', 'except' y/o 'else'
```

Para lanzar una excepción, se utiliza la función `raise`.

```
raise Excepcion ()
```

Funciones

Definición de funciones

Se identifican mediante la palabra reservada `def`.

```
def funcion ():
    # Haz algo
```

La primera línea de la función puede contener una cadena multi-línea que servirá como documentación de la función, a esto se le conoce como *docstring*.

```
def funcion ():
    '''Esta función hace muchas cosas.'''
    # Haz muchas cosas
```

Las funciones deben tener como mínimo una instrucción `pass`, indicando que no hace nada.

```
>>> def funcion ():
...
IndentationError: expected an indented block
```

Parámetros de las funciones

Los parámetros se definen entre los paréntesis de la definición de la función separados por comas.

```
def funcion (parametro1, parametro2):
    # Haz algo con los parámetros
```

Los parámetros pueden tener valores predeterminados, convirtiéndose en parámetros opcionales.

```
>>> def saludar (nombre, saludo='Hola'):
...     print saludo, nombre
...
>>> saludar ('Pepe')
Hola Pepe
>>> saludar ('Pepe', 'Adiós')
Adiós Pepe
>>> saludar ()
TypeError: saludar() takes at least 1 argument (0 given)
>>> saludar (saludo='Adiós', nombre='Pepe')
Adiós Pepe
```

Parámetros de las funciones

Usando un asterisco, se podrán definir parámetros variables que se convertirán en una tupla.

```
>>> def imprimir (*parametros):
...     for parametro in parametros:
...         print parametro
...
>>> imprimir ('¡¡Esto', 'es', 'Esparta!!')
¡¡Esto
es
Esparta!!
```

Usando dos asteriscos, se podrán definir parámetros variables mediante clave-valor que se convertirán en un diccionario.

```
>>> def saludar (**parametros):
...     for clave in parametros.keys ():
...         print clave, ':', parametros[clave]
...
>>> saludar (saludo='Hola', nombre='Pepe')
saludo : Hola
nombre : Pepe
```

Parámetros de las funciones

Todo lo explicado se puede combinar.

```
def funcion (obligatorio, *opcionales, **pares_opcionales):
    # Haz algo
```

Los asteriscos también se pueden usar para “desempaquetar” los parámetros.

Con un asterisco se “desempaqueta” una lista o una tupla.

```
>>> r = (0, 5)
>>> range (*r)
[0, 1, 2, 3, 4, 5]
```

Con dos asteriscos se “desempaqueta” un diccionario.

```
>>> def saludar (nombre, saludo='Hola'):
...     print saludo, nombre
...
>>> saludo = {'nombre': 'Pepe', 'saludo': 'Bye, bye'}
>>> saludar (**saludo)
Bye, bye Pepe
```

Variables globales

En Python no permite modificar el valor de una variable definida fuera de una función.

```
>>> variable = 12
>>> def funcion ():
...     variable += 1
...     print variable
...
>>> funcion ()
UnboundLocalError: local variable 'variable' referenced
before assignment
```

Para modificar la variable se deberá utilizar `global`.

```
>>> variable = 12
>>> def funcion ():
...     global variable
...     variable += 1
...     print variable
...
>>> funcion ()
13
```

Devolución de valores

Las funciones pueden devolver valores mediante la instrucción `return`.

```
>>> def suma (a, b):
...     return a + b
...
>>> print suma (1, 2)
3
```

Es posible devolver varios valores gracias a las tuplas de Python.

```
>>> def devolver_valores ():
...     return 1, 2
...
>>> a, b = devolver_valores ()
>>> print a, b
1 2
```

Clases

Definición de clases

Se identifican mediante la palabra reservada `class`.

```
class Clase:
    # Contenido de la clase
```

La primera línea de la clase puede ser *docstring*.

```
class Clase:
    '''Esta clase hace muchas cosas.'''
    # Contenido de la clase que hace muchas cosas
```

Las clases deben tener como mínimo una instrucción `pass`, indicando que no hace nada.

```
>>> class Clase:
...
IndentationError: expected an indented block
```

Las instancias (los objetos) se crean del siguiente modo:

```
>>> objeto = Clase ()
```

Definición de clases

Los métodos de las clases se definen como las funciones, pero dentro de la clase.

```
class Clase:
    def metodo (self, parametro):
        # Haz algo

    def otro_metodo (self):
        # Haz otra cosa
```

El parámetro `self` identifica la instancia de la clase (el objeto) que llama a ese método, se llama `self` por convenio.

El constructor de la clase se define mediante el método especial `__init__`.

```
class Clase:
    def __init__ (self, parametro):
        self.atributo = parametro
```

Definición de clases

Los atributos de la clase se definen antes que los métodos.

```
class Clase:
    atributo_clase = 'atributo a nivel de clase'

    def metodo (self, parametro):
        # Haz algo
```

Los atributos de la instancia de la clase se definen en el constructor.

```
class Clase:
    def __init__ (self):
        self.atributo_objeto = 'atributo a nivel de objeto'
```

Se pueden añadir atributos “al vuelo” a las instancias.

```
>>> objeto = Clase ()
>>> objeto.otro_atributo = 'otro atributo'
>>> print objeto.otro_atributo
'otro atributo'
```

Atributos privados

Todos los atributos de una clase y/o instancia son públicos.

La única posibilidad de “ocultar” los atributos, es utilizar dos subrayados delante del nombre de los atributos.

```
>>> class Clase:
...     def __init__ (self):
...         self.__oculto = 'oculto'
...
>>> objeto = Clase ()
>>> print objeto.__oculto
AttributeError: Clase instance has no attribute '__oculto'
```

Esto no bloquea el acceso, sólo renombra los atributos del siguiente modo `_Clase__atributo`.

```
>>> print objeto._Clase__oculto
'oculto'
```

Por convenio también se usa `_atributo` para indicar que un atributo no se debe modificar porque es de uso interno.

Herencia de clases

Para que una clase herede de otra, en la definición de la clase se añade entre paréntesis el nombre de la que hereda.

```
class ClasePadre:
    # Código de la clase padre

class ClaseHija (ClasePadre):
    # Código de la clase hija
```

Python permite la herencia múltiple de clases, para ello busca en profundidad y de izquierda a derecha el método o atributo entre las clases de las que hereda.

```
class ClaseHija (ClasePadre1, ClasePadre2, ...):
    # Código de la clase hija
```

De este modo, primero buscaría el método o el atributo en `ClasePadre1`, luego en las clases padre de ésta, para luego pasar a `ClasePadre2` y repetir el proceso.

Herencia de clases

A partir de la versión 2.2 se añadió un nuevo estilo de clase, en el que las clases debían heredar como mínimo de una llamada `object`.

```
class ClasePadre (object):
    # Código de la clase padre

class ClaseHija (ClasePadre):
    # Código de la clase hija
```

Con este nuevo estilo además se introdujo la función `super` para poder acceder a las clases padres.

```
super (Clase, self).metodo ()
```

Así, se puede ejecutar el constructor de la clase padre.

```
class ClaseHija (ClasePadre):
    def __init__ (self):
        super (ClaseHija, self).__init__ ()
```

Métodos especiales

Permiten cambiar el comportamiento de la clase.

Se identifican mediante dos subrayados delante y detrás del nombre del método.

Algunos de estos métodos especiales son:

- `__init__` y `__del__` constructor y destructor.
- `__add__` sobrecarga el operador de la suma.
- `__or__` sobrecarga el operador O-lógico (a nivel de bit).
- `__repr__` y `__str__` transforma la clase en una cadena de caracteres.
- `__len__` permite utilizar la función len sobre la clase.
- `__iter__` y `__next__` permite iterar sobre la clase.
- `__getattr__` y `__setattr__` obtención y asignación de atributos.
- `__getitem__` y `__setitem__` obtención y asignación mediante índices.

Métodos especiales

Por ejemplo, para sobrecargar el operador de la suma.

```
class Numero (object):
    def __init__ (self, valor=0):
        self.valor = valor

    def __add__ (self, other):
        return Numero (self.valor + other.valor)
```

De este modo, se podría realizar la siguiente operación.

```
>>> uno = Numero (1)
>>> dos = Numero (2)
>>> print uno.valor, dos.valor
1 2
>>> tres = uno + dos
>>> print tres.valor
3
```

http://docs.python.org/reference/datamodel.html#special-method-names

Scripts, módulos y paquetes

Scripts

En principio, los archivos con código Python se ejecutan de arriba hacia abajo, como scripts.

```
def saludar (nombre, saludo="Hola"):
    print saludo, nombre

saludar ("Pepe")
saludar ("Brian", "Hi")
```

Aunque pueden tener un punto de entrada, similar a la función `main` de la mayoría de lenguajes.

Se añade el siguiente bloque de código al final del archivo.

```
if __name__ == '__main__':
    # Punto de entrada
```

Como dije al principio se ejecutaría:

```
$ python script.py [PARAMETROS]
```

Módulos

Cualquier archivo con código Python es un módulo.

Los módulos se pueden importar para poder acceder a sus elementos (variables, funciones y clases) desde otro archivo.

```
import modulo

modulo.elemento  # Accede a 'elemento' de 'modulo'
```

Se pueden importar sólo los elementos que se quieran.

```
from modulo import elemento1, elemento2, ...
```

También se pueden importar todos los elementos del módulo.

```
from modulo import *
```

Aunque no se importarán los elementos que empiecen por subrayado.

Paquetes

Permiten distribuir un conjunto de módulos bajo un espacio de nombres común.

Se utiliza la siguiente notación.

```
import paquete.modulo.elemento
```

Los paquetes pueden contener otros paquetes.

Los paquetes se crean siguiendo la siguiente estructura de directorios.

```
paquete/                    # Paquete superior
    __init__.py             # Inicializa los datos de 'paquete'
    paquete_interior/       # Paquete interior
        __init__.py
        modulo1.py          # Un módulo de 'paquete_interior'
        modulo2.py
        ...
    otro_paquete_interior/
        __init__.py
        modulo1.py
        modulo2.py
        ...
```

Paquetes

Los archivos `__init__.py` son obligatorios para que Python trate estos directorios como paquetes.

Éstos pueden estar vacíos o contener código para inicializar los elementos del paquete.

Pueden contener la variable `__all__` para indicar que módulos importar al importar todos los elementos del paquete.

```
__all__ = [“modulo1”, “modulo2”]
```

Los paquetes internos pueden importar elementos de otros paquetes superiores.

Por ejemplo, desde el `modulo1` de `paquete_interior` se podría realizar:

```
from paquete.otro_paquete_interior import modulo2
```

Entrada y salida (I/O)

Entrada y salida (I/O)

Python ofrece funciones para la lectura y escritura de datos.

Como ya se ha visto, para escribir algo se puede utilizar la función `print`.

```
>>> print 'hola'
hola
```

Para la lectura de datos, se pueden utilizar las siguientes funciones.

La función `raw_input` lee una cadena de texto y puede mostrar un mensaje delante del texto a leer.

```
>>> leido = raw_input ()
hola
>>> print leido
hola
>>> leido = raw_input ('Mensaje ')
Mensaje hola
>>> print leido
hola
```

Entrada y salida (I/O)

También existe la función `input` que ejecuta la función `eval` sobre los datos leídos.

```
>>> leido = input ()
'hola'
>>> print leido
hola
>>> leido = input ('Mensaje ')
Mensaje 'hola'
>>> print hola
hola
```

¡Cuidado! Esta función se considera peligrosa porque espera que los datos sean sintácticamente correctos.

```
>>> leido = input ()
hola
SyntaxError: invalid syntax
```

Y lo que falta ...

Módulos interesantes

Dentro de todos los módulos que trae Python hay algunos que son interesantes y que todo el mundo debería conocer.

Algunos de éstos son:

- `argparse`: Procesa los parámetros que se pasan al ejecutar el programa.
- `distutils`: Funciones para crear y construir paquetes de Python.
- `itertools`: Funciones para crear iteradores eficientes.
- `os`: Funciones asociadas al sistema operativo.
- `re`: Módulo para manejar expresiones regulares.
- `sys`: Funciones específicas del sistema.
- `unittest`: Módulo para tests unitarios.

Más información en: http://docs.python.org/library/

Temas interesantes

De todo lo que Python permite hacer, hay bastantes cosas que no se han visto aquí, por estar fuera del alcance o por falta de tiempo por mi parte.

Algunos de estos temas son:

- Cambios en Python 3.
- Extensiones de Python.
- Funciones internas como map, reduce, zip, ...
- Funciones lamdba.
- Generadores.
- Introspección.
- Metaclases.
- Programación funcional.

También recomiendo leer este artículo:

- http://lucumr.pocoo.org/2011/7/9/python-and-pola/

Muchas gracias