
Ahora, algo totalmente diferente...

Python instantáneo

por Magnus Lie Hetland

Esto es un curso intensivo de introducción *mínima* al lenguaje de programación [Python](#). Para obtener más información, eche un vistazo a la documentación del web de Python, www.python.org, especialmente la [tutoría](#). Si se pregunta por qué debería interesarle, mire la página de [comparación](#), en la que aparece Python comparado con otros lenguajes.

Esta introducción ha recibido gran cantidad de [alabanzas](#) de lectores satisfechos y está traducida a varios idiomas, entre ellos [portugués](#), ruso, noruego y [castellano](#). La versión original, en inglés, está [aquí](#).

([Página principal de Python](#))

1. Lo básico

Para empezar, piense en Python con pseudo-código. Esto es casi cierto. Las variables no tienen tipo, así que no hay que declararlas. Aparecen cuando se les asigna algo y desaparecen al dejar de usarlas. La asignación se realiza mediante el operador `=`. Para comprobar la igualdad se utiliza el operador `==`. Se puede asignar varias variables a la vez:

```
x,y,z = 1,2,3

primero, segundo = segundo, primero

a = b = 123
```

Para definir bloques de código, se utiliza el sangrado (o indentación) *sólamente* (nada de `BEGIN/END` ni llaves). Éstas son algunas estructuras de control comunes:

```
if x < 5 or (x > 10 and x < 20):
    print "El valor es correcto."

if x < 5 or 10 < x < 20:
    print "El valor es correcto."

for i in [1,2,3,4,5]:
    print "Pasada nº ", i

x = 10
while x >= 0:
    print "x todavía no es negativo."
    x = x-1
```

Los dos primeros ejemplos son equivalentes.

La variable de índice en el bucle `for` recorre los elementos de una *lista* (escrita como en el ejemplo). Para realizar un bucle `for` "normal" (o sea, contando), utilice la función de serie `range()`.

```
# Mostrar los valores de 0 a 99 inclusive.
for value in range(100):
    print value
```

(La línea que comienza por "#" es un comentario y el intérprete le hace caso omiso)

Vale, ahora ya sabe suficiente para implementar cualquier algoritmo en Python. Vamos a añadir algo de interacción *básica*. Para obtener entrada del usuario, (de un indicador de texto), utilice la función de serie `input`.

```
x = input("Introduzca un número: ")
print "El cuadrado de ese número es: ", x*x
```

La función `input` muestra la solicitud dada (que podría estar vacía) y permite que el usuario introduzca cualquier valor Python válido. En este caso esperábamos un número, si se introduce algo diferente (una cadena, por ejemplo), el programa falla. Para evitarlo necesitaríamos algo de comprobación de errores. No voy a entrar en ese tema ahora, valga decir que si quiere guardar lo que el usuario ha introducido *textualmente* como un cadena (para que se pueda introducir *cualquier cosa*), utilice la función `raw_input`. Si desea convertir la cadena de entrada `s` a un entero, podría utilizar `int(s)`.

Nota: Si desea introducir una cadena con `input`, el usuario debe escribir las comillas explícitamente. En Python, las cadena pueden encerrarse entre comillas simples o dobles.

Así que tenemos cubiertas las estructuras de control, la entrada y la salida. Ahora necesitamos estructuras de datos impresionantes?. Las más importantes son las *listas* y los *diccionarios*. Las listas se escriben entre corchetes, y se pueden (por supuesto) anidar:

```
nombre = ["Cleese", "John"]

x = [[1,2,3],[y,z],[[[]]]]
```

Una de las ventajas de las listas es que se puede acceder a sus elementos por separado o en grupos, mediante *indexado* y *corte*. El indexado se realiza (como en muchos otros lenguajes) añadiendo el índice entre corchetes a la lista (observe que el primer elemento es el 0).

```
print name[1], name[0]
Muestra "John Cleese"

name[0] = "Smith"
```

El corte es casi como el indexado, pero se indican los índices de inicio y fin del resultado, con dos puntos (":") de separación:

```
x = ["spam", "spam", "spam", "spam", "spam", "eggs", "and", "spam"]

print x[5:7]
Muestra la lista ["eggs", "and"]
```

Observe que el índice final no se incluye en el resultado. Si se omite uno de los índices se supone que se quiere obtener todo en la dirección correspondiente. Esto es, `lista[:3]` quiere decir "cada elemento desde el principio de `lista` hasta el elemento 3, no incluido". (se podría decir en realidad el elemento 4, ya que contamos desde 0... bueno). `lista[3:]` significaría, por otra parte "cada elemento de `lista`, empezando por el 3 (inclusive), hasta el último inclusive". Se pueden utilizar números negativos para obtener resultados muy interesantes: `lista[-3]` es el tercer elemento desde el final de la lista...

Ya que estamos en el tema del indexado, puede encontrar interesante que la función de serie `len` da la longitud de una lista.

Y ahora, ¿qué pasa con los diccionarios? Para ser breves, son como listas, pero su contenido no está ordenado. ¿Y cómo se indexan, entonces? Bueno, cada elemento tiene una *clave* o "nombre" que se utiliza para buscar el elemento, como en un diccionario de verdad. Un par de diccionarios de ejemplo:

```
{ "Alice" : 23452532, "Boris" : 252336, "Clarice" : 2352525 }

persona = { 'nombre': "Robin", 'apellido': "Hood",
            'trabajo u ocupación': "Ladrón" }
```

Ahora, para obtener la ocupación de `persona`, utilizamos la expresión `persona["trabajo u ocupación"]`. Si le queremos cambiar el apellido, escribiremos:

```
persona['apellido'] = "de Locksley"
```

Simple, ¿no? Como las listas, los diccionarios pueden contener otros diccionarios. O listas, ya que nos ponemos. Y naturalmente, también las listas pueden contener diccionarios. De este modo, se pueden conseguir estructuras de datos bastante avanzadas.

2. Funciones

Próximo paso: Abstracción. Queremos dar un nombre a un trozo de código y llamarlo con un par de parámetros. En otras palabras, queremos definir una función (o "procedimiento"). Es fácil. Utilice la palabra clave `def` así:

```
def cuadrado(x):
    return x*x

print cuadrado(2)
Muestra 4
```

Para los que lo entendáis: Todos los parámetros en Python se pasan *por referencia* (como, por ejemplo, en Java). Para los que no: No os preocupéis :)

Python tiene todo tipo de lujos, como *argumentos con nombre* y *argumentos por omisión* y puede manejar un número variable de argumentos para una función. Para obtener más información, vea la [sección 4.7](#) de la tutoría de Python.

Si sabe utilizar las funciones en general, esto es lo que necesita saber sobre ellas en Python, básicamente (ah, sí, la palabra clave `return` detiene la ejecución de la función y devuelve el resultado indicado).

Algo que podría resultar interesante conocer, sin embargo, es que las funciones son *valores* en Python. Así que, si tiene una función como `cuadrado`, podría hacer cosas como:

```
queeeble = cuadrado
queeeble(2)
Muestra 4
```

Para llamar a una función sin argumentos debe recordar escribir `hazlo()` y no `hazlo`. La segunda forma sólo devuelve la función en sí, como valor (esto vale también para los métodos de los objetos... vea lo siguiente).

3. Objetos y cosas...

Supongo que conoce cómo funciona la programación orientada a objetos (de otro modo, esta sección podría resultar un poco confusa, pero no importa, empiece a jugar con los objetos :)). En Python se definen las clases con la palabra clave (¡sorpresa!) `class`, de este modo

```
class Cesta:

    # Recuerde siempre el argumento self
```

```

def __init__(self, contenido=None):
    self.contenido = contenido or []

def añadir(self, elemento):
    self.contenido.append(elemento)

def muestra_me(self):
    resultado = ""
    for elemento in self.contenido:
        resultado = resultado + " " + `elemento`
    print "Contiene:" + resultado

```

Cosas nuevas:

1. Todos los métodos (funciones de un objeto) reciben un argumento adicional al principio de la lista de argumentos, que contiene el propio objeto. Este argumento, por convención, se suele llamar `self` (que significa 'uno mismo'), como en el ejemplo.
2. A los métodos se los llama de este modo: `objeto.método(arg1, arg2)`.
3. Algunos nombres de método, como `__init__` están predefinidos, y significan cosas especiales. `__init__` es el nombre del *constructor* de la clase, es decir, es la función a la que se llama cuando creas una instancia.
4. Algunos argumentos son *opcionales* y reciben un valor dado (según lo mencionado antes, en la sección de funciones). Esto se realiza escribiendo la definición así:

```
def spam(edad=32): ...
```

Aquí, se puede llamar a `spam` con uno o cero parámetros. Si no se pone ninguno, el parámetro `edad` tendrá el valor 32.

5. "Lógica de cortocircuito." Esto es un punto... Ver más tarde.
6. Las comillas invertidas convierten un objeto en su representación como cadena (así que si `elemento` contiene el número 1, ``elemento`` es lo mismo que `"1"` mientras que `'elemento'` es una cadena literal).
7. El signo más `+` se utiliza también para concatenar listas, y las cadenas son sólo listas de caracteres (lo que significa que se puede utilizar indexado, corte y la función `len` en ellas; chulo, ¿eh?).

Ningún método o variable miembro es protegido (ni privado, ni nada de eso) en Python. La encapsulación es en su mayoría cuestión de estilo al programar.

Retomando el tema de la lógica de cortocircuito...

Todos los valores de Python se pueden utilizar como valores lógicos. Algunos, los más "vacíos", como `[]`, `0`, `""` y `None` representan el valor lógico "falso", mientras el resto (como `[0]`, `1` or `"Hola, mundo"`) representan el valor lógico "verdadero".

Las expresiones lógicas como `a and b` se evalúan de este modo: Primero, se comprueba si `a` es verdadero. Si *no*, simplemente se devuelve su valor. Si *sí*, simplemente se devuelve `b` (que representa el valor lógico de la expresión). La lógica correspondiente a `a or b` es: Si `a` es verdadero, devolver su valor. Si no, devolver `b`.

Este mecanismo hace que `and` y `or` se comporten como los operadores booleanos que supuestamente implementan, pero también permite escribir expresiones condicionales muy curiosas. Por ejemplo, el código

```

if a:
    print a
else:
    print b

```

Se puede sustituir por:

```
print a or b
```

De hecho, esto es casi un 'deje' en Python, así que mejor irse acostumbrando. Esto es lo que hacemos en el método `Cesta.__init__`. El argumento `contenido` tiene el valor por defecto `None` (que es, entre otras cosas, falso). Por lo tanto, para comprobar si tenía valor, podríamos escribir:

```
if contenido:
    self.contenido = contents
else:
    self.contenido = []
```

Por supuesto, ahora conocemos un método mejor, Y, ¿por qué no le damos el valor por omisión `[]` para empezar? Por el modo en que funciona Python, esto daría a todas las Cestas la misma lista vacía como contenido por omisión. Tan pronto como se empezara a llenar una de ellas, todas tendrían los mismos elementos y el valor por omisión dejaría de ser vacío... Para informarse sobre el tema, lea la documentación y busque la diferencia entre *identidad* e *igualdad*.

Otro modo de hacer lo anterior es:

```
def __init__(self, contenido=[]):
    self.contenido = contenido[:]
```

¿Adivina como funciona esto? En lugar de utilizar la misma lista vacía siempre, utilizamos la expresión `contenido[:]` para hacer una copia (hacemos un corte que contiene toda la lista).

Así que, para hacer realmente una `Cesta` y utilizarla (o sea, llamar a alguno de sus métodos) haríamos algo así:

```
b = Cesta(['manzana', 'naranja'])
b.añadir("limón")
b.muestra_me()
```

Hay más métodos mágicos además de `__init__`. Uno de ellos es `__str__`, que define el aspecto que quiere tener el objeto si se le trata como una cadena. Lo utilizaríamos en nuestra cesta en lugar de `presenta_me`:

```
def __str__(self):
    resultado = ""
    for elemento in self.contenido:
        resultado = resultado + " " + `element`
    return "Contiene:"+resultado
```

Y, si quisiéramos mostrar la cesta `b`, simplemente diríamos:

```
print b
```

chulo, ¿eh?

La herencia se realiza de este modo:

```
class CestaSpam(Cesta):
    # ...
```

Python permite la herencia múltiple, así que puede indicar varias super-clases entre los paréntesis, separadas por comas. Las clases se instancian así: `x = Cesta()`. Los constructores se definen, como dije, implementando la función miembro especial `__init__`. Pongamos que `CestaSpam` tuviera un constructor `__init__(self, tipo)`. Podría realizar una cesta de spam así:

```
y = CestaSpam("manzanas").
```

Si necesitase llamar al constructor de una super-clase desde el constructor H de `CestaSpam`, lo haría así: `Cesta.__init__(self)`. Observe que, además de proporcionar los parámetros normales, debe proporcionar explícitamente `self`, ya que `__init__` de la super-clase no sabe con qué instancia está tratando.

Para obtener más información sobre las maravillas de la programación orientada a objetos en Python, mire la [sección 9](#) de la tutoría.

4. Truco mental Jedi

(Esta sección está aquí sólo porque creo que mola. *No* es necesario en absoluto leerla para empezar a aprender Python)

¿Le gustan los ejercicios mentales? Si es así, si es realmente osado, debería echarle un vistazo al ensayo de Guido van Rossum sobre [metaclases](#). Si, por el contrario, prefiere que el cerebro *no* le explote, igual le satisface este truquito.

Python utiliza espacios de nombres dinámicos (no léxicos). Esto quiere decir que si tienes una función como ésta:

```
def zumo_naranja():
    return x*2
```

... donde una variable (en este caso `x`) no está ligada a un argumento, y no se le asigna un valor desde dentro de la función, Python utilizará el valor que tenga cuando se llama a la función. En este caso:

```
x = 3
zumo()
Devuelve 6
x=1
zumo_naranja()
Devuelve 2
```

Normalmente, éste es el comportamiento deseado (aunque el ejemplo es un poco rebuscado, pues es raro acceder a las variable de este modo). *Sin embargo*, a veces puede ser útil tener un espacio de nombres estático, o sea, guardas algún valor del entorno en que se crea la función. El modo de hacer esto en Python es por medio de los argumentos por omisión.

```
x = 4
def zumo_manzana(x=x):
    return x*2
```

Aquí, al argumento `x` se le asigna un valor por defecto que coincide con el *valor* de la variable `x` en el instante en que la función es definida. Por lo tanto, siempre que nadie proporcione un argumento para la función, funcionará así:

```
x = 3
zumo_manzana():
Devuelve 8
x = 1
zumo_manzana():
Devuelve 8
```

Concluyendo: El valor de `x` no cambia. Si esto fuese todo lo que quieríamos, podríamos limitarnos a escribir

```
def zumo_tomate():
    x = 4
    return x*2
```

incluso

```
def zumo_zanahoria():  
    return 8
```

Sin embargo, lo *importante* es que el valor de `x` se toma del *entorno* en el instante en que se define la función. ¿Qué utilidad tiene esto? Tomemos un ejemplo: Una función compuesta.

Queremos una función que funcione así:

```
from math import sin, cos  
  
sincos = componer(sin,cos)  
  
x = sincos(3)
```

Donde `componer` es la función que queremos realizar y `x` tiene el valor `-0.836021861538`, que es lo mismo que `sin(cos(3))`. Y ¿cómo lo hacemos?

Observe que estamos utilizando funciones como argumentos y eso ya es un truco en sí mismo.

Obviamente, `componer` toma dos funciones como parámetros y devuelve una función que a su vez toma un parámetro. Un esqueleto de la solución podría ser:

```
def componer(fun1, fun2):  
    def interior(x):  
        # ...  
    return interior
```

Nos tentaría poner `return fun1(fun2(x))` dentro de la función `interior` y dejarlo tal cual. No, no y no. Eso tendría resultados muy extraños. Imagine la siguiente situación:

```
from math import sin, cos  
  
def fun1(x):  
    return x + " mundo"  
  
def fun2(x):  
    return "Hola,"  
  
sincos = componer(sin,cos) # Versión incorrecta  
  
x = sincos(3)
```

Y bien, ¿qué valor tendría `x`? Correcto: `"Hola, mundo"`. Y ¿por qué? Porque cuando se la llama, toma los valores de `fun1` y `fun2` del entorno, no los que andaban por ahí cuando se creó. Para conseguir una función correcta, sólo hay que utilizar la técnica descrita anteriormente:

```
def componer(fun1, fun2):  
    def interior(x, fun1=fun1, fun2=fun2):  
        return fun1(fun2(x))  
    return interior
```

Ahora sólo nos queda esperar que nadie proporcione a la función resultante más de un argumento, ya que eso nos rompería los esquemas :). Y, a propósito, como no necesitamos el nombre `interior` y sólo contiene una expresión, podemos utilizar una función *anónima*, utilizando la palabra clave `lambda`:

```
def componer(f1, f2):  
    return lambda x, f1=f1, f2=f2: f1(f2(x))
```

Espartano, pero claro. Te tiene que gustar :)

...y si no ha entendido nada, no se preocupe. Al menos le ha convencido de que Python es más

que "un lenguajillo para scripts"... :)

5. Y ahora...

Sólo unas cosillas para terminar. Las funciones y clases más útiles se ponen en *módulos*, que son en realidad ficheros de texto legible con código Python. Puede importarlos y utilizarlos en sus propios programas. Por ejemplo, para utilizar el método `split` (trocear) del módulo estándar `string` (cadena), puede hacer estas dos cosas:

```
import string

x = string.split(y)
```

O...

```
from string import split

x = split(y)
```

Para obtener más información sobre la biblioteca de módulos estándar, eche un vistazo a www.python.org/doc/lib. Contiene un montón de cosas útiles.

Todo el código del módulo/script se ejecuta cuando se importa. Si quiere que su programa sea tanto un módulo importable como un script ejecutable, puede añadir algo así al final:

```
if __name__ == "__main__": ejecutar()
```

Esto es un modo mágico de decir que si el módulo se ejecuta como un script ejecutable (o sea, que no está siendo importado por otro script o módulo), se debe ejecutar la función `ejecutar`. Por supuesto, puede hacer cualquier cosa tras los dos puntos... :)

Y, si desea hacer un script ejecutable en UN*X, escriba esto como primera línea para hacer que el script se ejecute sin llamar a python explícitamente:

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

Finalmente, una breve mención de un concepto importante: Las excepciones. Algunas operaciones (como dividir por cero o leer de un archivo inexistente) causan una condición de error o *excepción*. Puede incluso generar las suyas propias y lanzarlas en los momentos adecuados.

Si no se hace nada con la excepción, el programa termina y muestra un mensaje de error. Esto se puede evitar con una construcción `try/except`. Por ejemplo:

```
def dividirSeguro(a,b):
    try:
        return a/b
    except ZeroDivisionError:
        return None
```

`ZeroDivisionError` es una excepción estándar. En este caso, se *podría* haber mirado si `b` era cero, pero hay muchos casos en que esto no es posible. Además, si no tuviéramos la cláusula `try` in `dividirSeguro`, haciéndola de este modo una función arriesgada, podríamos hacer algo como:

```
try:
    dividirInseguro(a,b)
except ZeroDivisionError:
    print "Se ha intentado dividir por cero en dividirInseguro"
```

En casos en los que *normalmente* no debería haber problemas concretos, pero *podrían* ocurrir, la utilización de excepciones permite evitar tediosas comprobaciones.

Bueno, eso es todo. Espero que aprendieras algo. Ahora, a [jugar](#). Y recuerde el lema de Python para aprender: "Ve a las fuentes, Lucas" (léase: Leer todo el código al que se le pueda echar las manos encima). :) Para arrancar, aquí hay un [ejemplo](#). Es el conocido algoritmo de Hoare, *QuickSort*. Y [aquí](#) hay una versión con la sintaxis resaltada en colores.

Merece la pena resaltar una cosa sobre el ejemplo. La variable `done` controla si la `partition` ha terminado, o no, de recorrer los elementos. Así que cuando uno de los dos bucles internos desea terminar la secuencia de intercambio completa ponen `done` a 1 y salen ellos mismos mediante `break`. ¿Por qué utilizan `done` los bucles internos? Porque, cuando el primer bucle interno finaliza con un `break`, que el siguiente bucle deba rearmar depende de si el bucle principal ha finalizado, esto es, si `done` se ha puesto a 1:

```
while not done:
    while not done:
        # Se repite hasta un 'break'

    while not done:
        # Sólo se ejecuta si el primero no puso "done" a 1
```

Una versión equivalente, posiblemente más clara, pero en mi opinión menos elegante, sería:

```
while not done:
    while 1:
        # Se repite hasta un 'break'

    if not done:
        while 1:
            # Sólo se ejecuta si el primero no puso "done" a 1
```

La única razón por la que he utilizado la variable `done` en el primer bucle ha sido que me gustaba conservar la simetría entre los dos. De este modo, se podría invertir el orden y el algoritmo todavía funcionaría.

Se pueden encontrar más ejemplos en la página [tidbit](#) de Joe Strout.

[[Página principal de Python](#)]

Copyright © [Magnus Lie Hetland](#) (mlh@idi.ntnu.no)

traducción de Marcos Sánchez Provencio

Last modified: Sun Mar 14 19:38:30 MET 1999