PEQUEÑA INTRODUCCIÓN A PYTHON

Héctor Manuel Mora Escobar
Universidad Central, Bogotá

20 de marzo de 2012

Este documento (en elaboración) pretende dar una visión rápida de algunas de las características de Python. Se supone que el lector conoce medianamente otro lenguaje de programación y que desea empezar a programar en Python.

Libros, documentos, tutoriales y manuales hay muchos y muy buenos. Este documento no es ni exhaustivo ni profundo. Espero simplemente que pueda servir como un primer paso.

Agradezco cualquier sugerencia o corrección que hagan llegar a

`hectormora@yahoo.com`

Capítulo 1

Empezando

1.1. Introducción

Python es un lenguaje de programación creado por Guido van Rossum a principios de los años 90. El nombre proviene del grupo de cómicos ingleses "Monty Python".

Algunas características:

- Gratuito.
- Multiplataforma (Windows, Linux, Unix, Mac, ...).
- Lenguaje interpretado o de "scripts" o guiones. Los lenguajes compilados tienen una ejecución más rápida, los interpretados son más flexibles y más portables. Realmente Python es semiinterpretado, se puede obtener un pseudocódigo de máquina llamado "bytecode".
- Tipado dinámico, no es necesario declarar el tipo de las variables, Python escoge la manera más adecuada.
- Orientado a objetos.

No es adecuado para programación de bajo nivel o aplicaciones de rendimiento crítico.

1.2. En Linux

En muchas distribuciones Linux, el Python ya viene instalado por defecto. Sino, se puede bajar el software en

`www.python.org`

o instalarlo mediante

`apt-get install python`

o como sea necesario según la distribución.

1.2.1. Primeros pasos

Para entrar a python basta con abrir una consola y digitar

`python`

Aparecerá algo semejante a:

```
Python 2.7.1+ (r271:86832, Apr 11 2011, 18:05:24)
[GCC 4.5.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Los símbolos `>>>>` forman el "*prompt*" de Python. A su derecha se escriben las órdenes. Al escribir

```
11/2  (+ tecla Enter)
```

aparecerá

```
5
>>>
```

El resultado de la división es 5 y aparece otro prompt para escribir una nueva orden. Al escribir

```
11.0/2  (+ tecla Enter)
```

aparecerá

```
5.5
>>>
```

En este documento, **no se mostrará más el prompt ni se indicará explícitamente el uso de la tecla Enter**.

Al escribir

```
print 'Hola todos'
```

aparecerá

```
Hola todos
```

También se puede utilizar comilla doble, es decir,

```
print "Hola todos"
```

Combinando adecuadamente se puede escribir

```
print "pa'l diablo"
```

Al escribir

```
aviso = 'Hola todos'
```

no aparece ningún resultado, pero al agregar

```
print aviso
```

aparecerá

```
Hola todos
```

Para salir de python: digitar `quit()` o `exit()` u oprimir Ctrl-D.

1.2.2. El primer programa

El Python usual para Linux no viene con ambiente de desarrollo. Hay un ambiente de desarrollo llamado `idle` ???? Este se puede instalar mediante ... ???? En este documento se supondrá que los programas en Python se escriben con cualquier editor de texto ASCII, de preferencia uno que tenga orientación para Python, por ejemplo, emacs o kate.

Consideremos dos archivos muy parecidos, solamente hay una linea adicional. El archivo `ej001.py`:

```
#!/usr/bin/python
# programa  ej001.py
# esta linea es un comentario
# Hector Mora, 9 de julio de 2011
print "Hola al fin."
```

y el archivo `ej002.py`:

```
# programa  ej002.py
# esta linea es un comentario
# Hector Mora, 9 de julio de 2011
print "Hola al fin."
```

Si desde una consola de Linux, sin abrir Python, se da la orden

```
python  ej001.py
```

o se da la orden

```
python  ej002.py
```

el resultado en la pantalla es exactamente el mismo. Además, a ambos se les puede dar permiso de ejecución, por ejemplo,

`chmod 777 ej001.py` o bien `chmod +x ej001.py`

y de forma análoga para `ej002.py`. Hasta este punto todo sigue siendo igual. Sin embargo el archivo `ej001.py` es realmente un archivo ejecutable, es decir al dar la orden

```
./ej001.py
```

funciona. En cambio la orden `./ej002.py` produce errores.

Queda por aclarar la primera línea del archivo `ej001.py`. Los dos primeros símbolos, `#!` , deben ser así. Lo que sigue es simplemente la carpeta donde está instalado Python. Esto se puede saber por medio de

```
which python
```

También se puede, desde el interpretador, ejecutar el programa mediante

```
>>> execfile('ej001.py')
```

En este caso, el archivo `ej001.py` puede tener o no tener la línea
`#!/usr/bin/python`

En lo que sigue, mientras no se diga lo contrario, se pupone que los archivos de Python en Linux siempre tienen esta línea, pero ésta no aparecerá en los ejemplos de este documento.

1.3. En Windows

1.3.1. Instalación

El instalador de Python para Windows se puede descargar en la página

`www.python.org`

A la fecha de hoy, 7 de julio de 2011, hay dos versiones para descargar, la 2.7.2 y la 3.2.

Supongamos que se va a instalar la versión 2.7.2. Picando en

▷*DOWNLOAD* ▷*Python 2.7.2* ▷*Windows x86 MSI Installer (2.7.2)*

se obtiene el archivo `python_2.7.2.msi`

Después de descargado este archivo, al activarlo se obtiene la instalación de Pyhton.

1.3.2. Primeros pasos

Python se puede utilizar de manera interactiva o por medio de programas. Al activar el ambiente Python, (IDLE (Python GUI), se obtiene el interpretador para un uso interactivo. Aparecerá algo semejante a:

```
Python 2.7.2 (default, Jun 12 2011, 15:08:59) [MSC v.1500 32 bit
(Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>>
```

Los símbolos `>>>>` forman el "*prompt*" de Python. A su derecha se escriben las órdenes. Al escribir

```
11/2  (+ tecla Enter)
```

aparecerá

```
5
>>>
```

El resultado de la división es 5 y aparece otro prompt para escribir una nueva orden. Al escribir

```
11.0/2  (+ tecla Enter)
```

aparecerá

```
5.5
>>>
```

En este documento, **no se mostrará más el prompt ni se indicará explícitamente el uso de la tecla Enter**.

Al escribir

```
print 'Hola todos'
```

aparecerá

```
Hola todos
```

También se puede utilizar comilla doble, `''`, es decir,

```
print "Hola todos"
```

Combinando adecuadamente se puede escribir

```
print "pa'l diablo"
```

Al escribir

```
aviso = 'Hola todos'
```

no aparece ningún resultado, pero al agregar

```
print aviso
```

aparecerá

```
Hola todos
```

Se puede salir de Python por las formas usuales de Windows o por medio de CTRL-D o escribiendo

```
quit()
```

1.3.3. Primer programa

Los programas en Python se pueden escribir con el editor de texto ASCII de su preferencia o utilizando el editor del ambiente Python. Es conveniente usar un editor tal que una de sus orientaciones sea Python. Los archivos (los fuentes o códigos) de Python tienen extensión `.py`. Cuando un editor orientado a Python detecta que se trata de un archivo Python, usa colores diferentes para las palabras claves, para las cadenas, etc.

Si va a utilizar el editor del ambiente Python, puede picar

▷*File* ▷*New Window*

Enseguida, en la pantalla blanca que aparece, escriba el programa, por ejemplo,

```
# primer ensayo de Python
# 8 de julio 2011
# Hector Mora
x = 11/2
print 'x = ', x
```

y lo guarda. Una vez escrito y guardado (supongamos con nombre `ej001.py`), lo ejecuta picando en la ventana del editor Python

▷*Run* ▷*Run Module*

o, simplemente, mediante la tecla F5. Así en el ambiente Python aparecerán los resultados del programa (o los errores del programa).

1.3.4. Con otro editor

Si por el contrario, no se usa el IDLE de Pyhton y se usa el editor de texto ASCII de su preferencia, el camino es un poquito más largo, pero posible.

Es necesario saber abrir una ventana de Símbolo del sistema (una ventana de fondo negro y letras blancas)

▷*Todos los programas* ▷*Accesorios* ▷*Símbolo del sistema*

También es necesario saber donde fue instalado Python, posiblemente

```
C:\Pyhton27
```

Supongamos que con el editor preferido se escribió el archivo `ej001.py`, que contiene

```
# primer ensayo de Python
# 8 de julio 2011
# Hector Mora
x = 11/2
print 'x = ', x
```

Después de abrir una ventana de Símbolo del sistema, es necesario ordenar que Python ejecute el programa `ej001.py`. Para esto es necesario digitar en la "ventana negra":

```
C:\Python27\python ej001.py
```

Otra opción consiste en modificar el `path` del sistema mediante:

▷*Equipo* ▷*Propiedades de sistema*

▷*Configuracion avanzada del sistema* ▷*Opciones avanzadas*

▷*Variables de entorno* ▷*Variables del sistema*

Escoger `path`,

▷*Editar*

Agregar al final, punto y coma, más la ruta completa de la carpeta donde está instalado Python, por ejemplo,

```
;C:\Python27\
```

y después

▷*Aceptar* ▷*Aceptar* ▷*Aceptar*

Una vez modificado adecuadamente el `path` basta con digitar en la ventana negra

```
python ej001.py
```

Capítulo 2

Generalidades

En lo que sigue, mientras no se diga lo contrario, se supone que las órdenes (o comandos) de Python están escritas en un archivo.

Por el momento, **las órdenes siempre deben empezar en la primera columna de cada línea**. Las líneas de comentarios pueden empezar en cualquier columna. Sin embargo, es preferible que también empiecen en la primera columna. El siguiente archivo funciona perfectamente.

```
# comentario
      # otro comentario

x = 11/2
y = 11/2.0
z = x + y
print 'x = ', x, '  y = ', y, '  z = ', z
```

En cambio el siguiente archivo produce errores:

```
x = 11/2
 y = 11/2.0
z = x + y
print 'x = ', x, '  y = ', y, '  z = ', z
```

Más adelante se verá que cuando se utiliza `if, for`, etc, o en las funciones, debe haber una sangría y debe ser respetada estrictamente.

Primeros consejos de estilo:

- Un espacio antes del igual, del más y del menos.
- Un espacio después del igual, del más y del menos.
- Un espacio después de la coma.
- La longitud máxima de una línea debe estar entre 60 y 80 columnas.

2.1. Variables, asignaciones, operaciones, ...

Considere el siguiente archivo (o las órdenes, no los comentarios, digitadas una a una en el intepretador):

```
#!/usr/bin/python
# ejemplos varios

a = 10/3
b = 10/3.0
t = 'ejemplo de cadena'
print a, b, t

print 'tipo de a : ', type(a), '   tipo de b : ', type(b)
print 'tipo de t : ', type(t)

a = a*b  # otro comentario
print 'a = ', a,'  tipo de a : ', type(a)

pesoEsp = 1.01
vel_luz = 3.0e5

x = (( a + b )/(c*d) + e )/5.2

a += 2

m = 29%6
```

```
n = 23.33//5.1
print m, n

w = 3 + 4j
w1 = 3 - 5J
print type(w), type(w1), w.imag, abs(w)

w2 = complex(m, n)
print type(w2)

b = True
print type(b)

u, v = 3, 4.2
print 'u = ', 'v = ', v

x = y = z = 3.2
print 'x, y, z : ', x, y, z

f = 1234567890
g = 12345678901
h = 12L
print type(f), type(g), type(h)

t1 = 'Hola '
t2 = 'Juanito'
t3 = t1+t2
t4 = t1*4
print t3, t4

t5 = 'Hola'   ' Pedro'
print 't5 =', t5

t6 = t5[0:3]
t7 = t5[4:]
print t6, t5[-1], t7
```

- En una línea, lo que sigue al símbolo `#` es un comentario. Si el símbolo `#` está en la primera columna de la línea, entonces toda

la línea es un comentario.

- Los nombres de variables o identificadores pueden contener, letras minúsculas, mayúsculas, dígitos o la barra baja `_` . Python diferencia las minúsculas de las mayúsculas. Los nombres de las variables empiezan por una letra. Usualmente la primera letra es minúscula y dentro del nombre de la variable una letra mayúscula se utiliza para hacer más visible una palabra o una abreviatura.

- Las palabras reservadas de Python, no deben ser usadas como nombres de variables:

```
and       assert    break     class     continue
def       del       elif      else      except
exec      finally   for       from      global
if        import    in        is        lambda
not       or        pass      print     raise
return    try       while     yield
```

- La asignación (no es un igualdad matemática) se hace mediante el símbolo `=` .

- Las cuatro operaciones aritméticas y la potencia se representan por:

```
+   -   *   /   **
```

- Los paréntesis sirven para agrupar operaciones.

- Escribir `x += 2` produce el mismo resultado que `x = x + 2` . Tambien se puede hacer con las otras tres operaciones aritméticas.

- Hay tres tipos fundamentales de datos: `int` o entero, `float` o número de punto flotante y `str` o cadena. Python asigna de manera adecuada el tipo a cada variable y este puede cambiar (variable `a`). Aunque el conjunto de números reales en matemáticas y el conjunto de números de punto flotante en Python son diferentes, en este documento, abusando del lenguaje, diremos que los números de punto flotante son los números reales.

- Es posible hacer asignaciones en paralelo: `u, v = 3, 4.2`

- También es posible hacer una asignación múltiple: `x = y = z = 3.2`

- Para obtener el residuo de la división se usa `%` . Los dos números pueden ser enteros o de punto flotante. El residuo siempre será menor que el valor absoluto del divisor.
- Por medio de `//` se efectúa la división entera entre dos reales, es decir, se obtiene el cociente entero de la división.
- También hay otros tipos, por ejemplo los números complejos y las variables boolenas (pueden se `True` o `False` .
- Por medio de `nombre.imag` se obtiene la parte imaginaria de un complejo. La función `abs`, utilizada con un complejo, calcula el módulo o norma del complejo.
- La variable `f`, según lo esperado, resulta ser entera. La variable `g` , tiene un dígito más, es de tipo `long` , entero largo. Colocando `L` al final del número se obliga a Python a que lo tome como entero largo.
- Las cadenas se pueden concatenar mediante `+` y se pueden concatenar repetidamente por medio de `*`
- En la asignación para `t5` hay dos cadenas constantes separadas por espacios, Python las concatena automáticamente.
- La cadena `t6` se obtiene como una subcadena de `t5` con las posiciones 0 (la primera), 1 y 2.
- La cadena `t7` va desde la posición 4 hasta el final de `t5` .

2.2. Funciones matemáticas

Python tiene predefinidas varias funciones matemáticas. Para usarlas es necesario el módulo de matemáticas y llamarlas indicando el módulo de matemáticas.

```
import math
print math.sqrt(2)
print math.cos(3.14)
```

Algunas de las principales funciones matemáticas son:

```
acos      copysign  fabs      isinf     pow
acosh     cos       factorial isnan     radians
asin      cosh      floor     ldexp     sin
asinh     degrees   fmod      lgamma    sinh
atan      erf       frexp     log       sqrt
atan2     erfc      fsum      log10     tan
atanh     exp       gamma     loglp     tanh
ceil      expm1     hypot     modf      trunc
```

Para obtener ayuda, desde el interpretador digite

```
help()
```

Lo anterior permite ingresar al ambiente de ayuda; aparecerá el prompt

```
help>
```

Allí digite, por ejemplo,

```
math
```

Para salir del ambiente de ayuda y regresar al interpretador, digite

```
quit
```

Hay otras funciones relacionadas con matemáticas, que están directamente en Python y no en el módulo de matemáticas. Por ejemplo,

```
x = -4.123456789
print abs(x), round(x), round(x, 3)
```

Hay otra manera de importar un módulo y usar sus funciones:

```
from math import *

print sqrt(2)
print cos(2)
```

Así el uso de las funciones es más sencillo y corto. El inconveniente se puede presentar en un programa grande cuando haya conflicto entre dos funciones, de módulos diferentes, pero con el mismo nombre.

Existe una tercera manera de importar un módulo, dándole un nombre corto:

```
import math as mt

print mt.sqrt(2)
print mt.cos(3.14)
```

2.3. Operadores lógicos y relacionales

`or`	o
`and`	y
`not`	no
`==`	igual
`!=`	diferente
`<=`	menor o igual
`>=`	mayor o igual
`<`	menor
`>`	mayor

Las variables booleanas, como `p`, toman los valores `True` o `False`. Por ejemplo:

```
x = 2
p = x <= 20/10  # o tambien  p = ( x <= 20/10 )
print 'p = ', p
q = p and x != 6/3 # q = ( p and ( x != 6/3 ) )
print 'q = ', q
```

En matemáticas es usual escribir $1 \leq x \leq 10$. En la muchos lenguajes de programación es necesario escribir, $1 \leq x \ \& \ x \leq 10$. En Python se puede escribir como en matemáticas,

```
r =  1 <= x <= 10
print 'r = ', r
```

2.4. Formato

Hasta ahora, en los ejemplos vistos, Python escoge una forma para escribir con la orden `print`. A veces, se desea que los resultados queden alineados verticalmente. En este caso, se puede utilzar un formato escogido por el programador. Considere el siguiente ejemplo:

```
import math

i = 8
j = 10

print i, math.pow(i, 1/3.0), i**3
print j, math.pow(j, 1/3.0), j**3

print '%3d %15.6f %8d' % ( i, math.pow(i, 1.0/3), i**3 )
print '%3d %15.6f %+8d' % ( j, math.pow(j, 1.0/3), j**3 )

n1 = 'Juan'
a1 = 'Saldarriaga'

n2 = 'Abelardo'
a2 = 'Toro'


print '%12s%12s' % (n1, a1 )
print '%12s%12s' % (n2, a2 )

print '%-12s%-12s' % (n1, a1 )
print '%-12s%-12s' % (n2, a2 )

print 'i = %3d,  i^2 = %3d, raiz = %f' % (i, i*i, math.sqrt(i))
```

- En el primer grupo de `print`, aparecen los resultados con el formato predefinido de Python, pero estos resultados no están alineados.
- En los `print` del segundo grupo, primero está una cadena con el formato general, después `%` y después una lista con los datos

que van a ser impresos.

- En el formato general hay tres formatos particulares, uno para cada elemento de la lista.
- El formato `%3d` indica que se va a imprimir un número entero utilizando tres columnas. El número quedará justificado a la derecha.
- El formato `%15.6f` indica que se va a imprimir un número de punto flotante en 15 columnas de las cuales las últimas 6 son para las cifras decimales.
- El `+` del formato `%+8d` indica de todas maneras se escribirá el signo `+` de los números positivos.
- El formato `%12s` indica que se va a imprimir una cadena en 12 columnas.
- En los `print` del último grupo, el formato `%-12s` indica que la cadena será justificada a la izquierda.
- El último `print` muestra que dentro de la cadena de formato puede haber información adicional que aparecerá en el resultado.

Capítulo 3

Colecciones

3.1. Listas

Una lista es una colección ordenada de datos no necesariamente del mismo tipo.

```
a = 3.14
b = False
x = [ a, 10, b, 'Hola', [2, 3]]
```

Se puede usar un dato de la lista escribiendo el nombre de la lista seguido de la posición del dato (empezando desde 0) entre corchetes (paréntesis rectangulares)

```
import math
print math.cos(x[0]), x[1], x[4][0]
```

Como el elemento en la posición 4 de `x` es a su vez una lista también se puede utilizar un elemento de esa lista.

Los elementos de la lista se pueden modificar por medio de una asignación:

```
x[1] = 4
print  x
```

En una lista se pueden utilizar subíndices negativos, -1 corresponde al último elemento de la lista, -2 corresponde al penúltimo, etc.

```
print x[-2]
```

Es posible utilizar sublistas o partes de una lista:

```
y = [10, 11, 12, 13, 14, 15, 16]

z = y[0:3]
print 'z: ', z

w = y[0:5:2]
print 'w: ', w

u = y[2:]
print 'u: ', u

y[2:5] = [22, 23, 24]
print 'y: ', y
```

En la definición de `z` se toman los elementos de `y` desde la posición 0 hasta la posición 3 sin incluirlo, es decir, hasta la posición 2. En la definición de `w` se toman los elementos de `y` desde la posición 0 hasta la posición 4, de dos en dos. En la definición de `u` se toman los elementos de `y` desde la posición 2 hasta el final. Una sublista puede ser modificada por medio de una asignación, como en `y[2:5] = ...`

Usando `del` es posible eliminar elementos de una lista. Por medio de `in` se puede averiguar si un valor está en la lista. Las funciones `len`, `min`, `max` permiten obtener la longitud (o tamaño), el mínimo y máximo de una lista.

```
x = [10, 11, 12, 13, 14, 15]
y = [20, 21, 22, 23, 24, 25]

print len(x), min(x), max(x)

del x[2]
```

```
del y[2:]
print 'x = ', x, 'y = ', y

p = 12 in x
q = 12 in y
print p, q
```

3.1.1. Métodos de las listas

Enseguida del nombre de la lista se escribe punto, el método y entre paréntesis el parámetro o los parámetros, `nombre.metodo(...)` . Algunos de los métodos de las listas son:

`append`	agrega un elemento al final de la lista
`count`	número de veces de un elemento
`index`	índice de un elemento
`extend`	agrega los elementos de una lista
`insert`	inserta
`pop`	quita el último
`remove`	quita la primera aparición de un elemento
`reverse`	invierte las posiciones
`sort`	ordena

```
x = [10, 11, 12, 13, 14, 15]
x.append(10)
print 'x, count index: ', x, x.count(10), x.index(13)

x.extend([100, 200] )
print 'despues de extend: ', x

x.insert(3, 300)
print 'despues de insert: ', x

x.pop()
print 'despues de pop: ', x

x.remove(10)
print 'despues de remove: ', x
```

```
x.reverse()
print 'despues de reverse: ', x

x.sort()
print 'despues de sort: ', x
```

3.2. Métodos de las cadenas

Algunos de los métodos para las cadenas son los usados en el siguiente ejemplo:

```
t = 'Erase una pobre viejecita'
print t
print('01234567890123456789012345678901234567890')

i = t.find('una')
j = t.find('b')
k = t.find('uno')
l = t.find('e', 21)
m = t.find('e', 10)
print 'find:  i, j, k, l, m: ', i, j, k, l, m


s = '-x-'
x = ['a', 'be', 'ce']
z = s.join(x)
print 'join:  ', z

print 'lower:   ', t.lower()

print 'replace: ', t.replace('e ', 'EE ')

tt = '   Erase una  pobre   viejecita   '

print 'tt:', '---'+tt+'---'
x = tt.split()
print 'split: ', x
```

```
y = tt.strip()
print 'strip: ', '---'+y+'---'

import string
tabla = string.maketrans('e', 'E')
print 'translate: ', t.translate(tabla)
```

- Por medio de `find` se obtiene el índice o posición de una cadena en otra. Cuando hay un segundo parámetro, éste indica donde se debe empezar la búsqueda. Devuelve -1 cuando no encuentra la cadena.
- Usando `join`, una cadena sirve para concatenar los elementos de una lista de cadenas.
- Con el método `lower` todas las mayúsculas se convierten en minúsculas.
- En una cadena grande, hay varias partes separadas por un símbolo o por una cadena corta. Con `split` se obtiene una lista cuyos elementos son las partes de la cadena grande.
- El método `strip` quita los espacios en blanco al comienzo y al final de una cadena.
- Por medio de `translate`, en una cadena grande se hace una busqueda y remplazo de una cadena por otra del mismo tamaño. Estas dos cadenas del mismo tamaño deben haber servido para construir una tabla usando la función `maketrans` del módulo `string`

3.3. Tuplas

Una tupla (perdón por el anglicismo), a diferencia de una lista, es inmutable (los valores no se pueden modificar y no se puede cambiar el tamaño). Se definen de manera análoga a las listas y su manejo tiene varios aspectos comunes con las listas.

```
t = (11, 12, 13, 14, 15, 16)
```

```
print 't: ', t

t2 = 11, 12, 13, 14, 15, 16
print 't2: ', t2
print t == t2

t3 = (4)
t4 = (4,)
print 'tipos: ', type(t3), type(t4)

print t[4], t[0:5]
```

Se definen usualmente con paréntesis (redondos) pero no son indispensables. En las líneas anteriores `t` y `t2` son iguales. Cuando la tupla tiene un único elemento es necesaria una coma. La subtupla `t[0:5]` va desde la posición 0 hasta la posición 4.

El siguiente ejemplo muestra asignaciones en paralelo usando listas y tuplas. Además muestra que una lista se puede modificar y una tupla no.

```
x = [2, 3, 4]
y = (5, 6, 7)

a, b, c = x
d, e, f = y
print a, b, c, d, e, f
x[2] += 20
y[2] += 20 # produce error
```

Los símbolos de una cadena se pueden obtener como en las listas y tuplas. A partir de una cadena se puede obtener una lista o una tupla:

```
c = 'Hola'
print c[0]

x = list(c)
y = tuple(c)

print c, x, y
```

3.4. Diccionarios

Un elemento de un diccionario, un `dict`, es una pareja clave-valor. La clave y el valor se separan por dos puntos `:`, un elemento se separa de otro por una coma. Al comienzo y al final se usan llaves `{ }`. No puede haber claves repetidas.

```
d = {"Claudia": 3451, "Directora": 3452, 0: 3451,
     'Clau':'Jefe'}
print d, type(d)
print d['Claudia'], d[0]

d['Clau'] = 3453
print d
```

Capítulo 4

Estructuras de control

Los ejemplos presentados hasta ahora podían ser realizados interactivamente con el interpretador o escritos en un archivo. A partir de ahora es casi indispensable escribir los ejemplos o ensayos en un archivo, es decir, es necesario hacer programas. Si está usando Linux recuerde que la primera línea es semejante a:

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

Algunas veces una orden de Python puede resultar muy larga. Si se desea se puede continuar escribiendo en el siguiente renglón. Para ello el primer renglon debe acabar con \ ("backslash") o debe haber un paréntesis izquierdo sin su correspondiente derecho. Observe el siguiente ejemplo:

```
x = 1 + 2 + 3  \
+ 4 + 5
print x

y = ( 1 + 2 + 3
+ 4 + 5 )*10
print y

z = 1 + 2 + 3
+ 4 + 5
print z
```

4.1. `if`

El siguiente programita trata de lo siguiente. El computador genera aleatoriamente un entero entre 1 y 5 inclusive y se desea tratar de adivinar el entero generado.

```
import random
x = random.randint(1,5)

n = input('Adivine un entero, de 1 a 5: ')

if n == x :
    print '\nBravo'
    print 'Felicitaciones'
print '\n\nHasta pronto.\n\n'
```

El módulo para generar números aleatoriamente es `random`. La función `randint` genera un entero entre los límites indicados. La función `input` coloca un aviso y permite entrar un número.

En el `if`, después de la condición, en este caso `n == x`, debe haber dos puntos (`:`). Las órdenes que se ejecutan cuando la condición es verdadera deben tener sangría. En muchos lenguajes, C, Java, ..., la sangría dentro de las estructuras de control es aconsejable mas no indispensable. En Python , en las estructuras de control, **la sangría es indispensable**.

En este ejemplo cuando la condición es verdadera, el programa escribe Bravo y Felicitaciones. Después, de todas maneras escribe Hasta pronto. Cuando no se cumple la condición, el programa simplemente escribe Hasta pronto.

En algunas de las cadenas anteriores aparece `\n` que indica un cambio de línea o línea en blanco adicional.

El `if` también puede tener `else`, que sirve para las acciones que se deben realizar si la condición es falsa. En el siguiente ejemplo, modificación del anterior, cuando el usuario no adivina el entero generado, el programa escribe `Fallaste corazon` y enseguida escribe `Hasta pronto`, la despedida común a los dos casos.

```
import random
x = random.randint(1,5)

n = input('\nAdivine un numero, de 1 a 5: ')

if n == x :
    print '\nBravo'
    print 'Felicitaciones'
else:
    print '\nFallaste corazon.'

print '\n\nHasta pronto.\n\n'
```

A veces es necesario utilizar varios `if` en serie, uno después de otro. En este caso se se puede utilizar el `if ... elif ... else`. Consideremos el siguiente ejemplo ficticio. Dependiendo de la edad, una persona se puede clasificar de la siguiente manera

0 - 17	menor
18 - 30	adulto joven
31 - 61	adulto maduro
62 -	adulto mayor

Esta clasificación se podría hacer de la siguiente manera:

```
n = input('\nEdad: ')

if n <= 17 :
    print 'Menor.'
elif n <= 30 :
    print 'Adulto joven.'
elif n <= 61 :
    print 'Adulto maduro.'
else :
    print 'Adulto mayor.'

print '\nHasta pronto.\n\n'
```

4.1.1. Sobre input y raw_input

La orden `input` permite leer naturalmente números enteros, enteros largos, números de punto flotante o valores booleanos. Considere

```
x = input('\n\nDigite algo: ')
print 'x = ', x, '  y su tipo es: ', type(x)
```

Lo anterior funciona perfectamente si lo digitado es

```
23
12345678901234
-23L
2.5
True
```

Por el contrario, si digita `Juanito` , habrá error. Si desea entrar una cadena debe digitar `'Juanito'`, lo cual parece incómodo. Para entrar una cadena sin tener que digitar las comillas, se puede usar `raw_input`

```
x = raw_input('\n\nDigite su nombre: ')
print 'x = ', x, '  y su tipo es: ', type(x)
```

Una cadena que represente adecuadamente un número puede ser convertida en número:

```
t = '234'
s = '234.56'
print t, float(t), int(t), long(t), float(s)
```

4.2. while

Mientras una condición sea verdadera, se repite un grupo de órdenes.

Consideremos ahora el siguiente ejemplo. Se desea adivinar un número entero entre 0 y 100 inclusive, pero ahora el usuario puede hacer varios ensayos hasta que adivine o hasta que digite un entero negativo.

```
print'\n\n\n'

import random
x = random.randint(0,100)

bien = 0
k = 0

while bien == 0 :
    n = input('Digite un entero entre 0 y 100 inclusive: ')
    k = k+1
    if n == x :
        bien = 1
    elif n < x :
        print n, ' es muy pequeno.'
    else :
        print n, ' es muy grande.'

print '\nMuy bien, hizo ', k, 'intentos.\n'
```

Dentro de un `while` puede haber un `if` u otra estructura de control. En general, dentro de una estructura de control puede haber una o varias estructuras anidadas y éstas a su vez pueden tener otras estructuras anidadas.

4.3. `for`

Consideremos el siguiente ejemplo

```
print '\n\n\n'

n = input('entero  n  mayor o igual a 1:  n = ')
if n < 1 :
    print '\nNumero indecuado.\n\n'
    quit()

suma = 0
```

```
prod = 1
for i in range(1,n+1) :
    suma = suma + i
    prod = prod*i

print 'la suma es: ', suma,
print 'el producto es: ', prod, ' = ', float(prod)
```

La estructura `for` es más general que en la mayoría de los lenguajes. En ellos, la variación se hace de manera aritmética, es decir, con valores que se van incrementando en un valor fijo, 1, 3, -1, 0.01. En Python la variación se hace con los elementos de una lista.

Sean `a` y `b` enteros con `a < b`. La orden `range(a, b)` construye una lista con valores entre el primer parámetro inclusive y el segundo parámetro exclusive, es decir, `range(a, b)` construye una lista con los valores a, $a+1$, ..., $b-1$. Si se quiere desde a hasta b inclusive, entonces se debe utilizar

```
range(a, b+1)
```

Con un solo parámetro, se supone que el primer parámetro es 0.

```
algo = range(3, 10)
otro = range(10)
print algo, type(algo), otro
```

Si se necesita un incremento diferente de 1, se usa un tercer parámetro:

```
range(a, b, 2)
```

Laa órdenes `range(10, 20, 2)` y `range(10, 19, 2)` producen la misma lista. El tercer parámetro puede ser negativo, por ejemplo

```
range(20, 10, -1)
```

produce una lista con los valores 20, 19, 18, ..., 12, 11. Si se desea desde b, disminuyendo hasta a inclusive,

```
range(b, a-1, -1)
```

El siguiente ejemplo muestra el uso de un `for` con los elementos de una lista predeterminada. Se trata de saber si un entero entre 2 y 120 es primo.

```
n = input('entero  entre  2  y 120:  n = ')

if n < 2 or n > 120:
    print '\nNumero indecuado.\n\n'
    quit()

p = [2, 3, 5, 7]
primo = 1
for i in p :
    c = n/i # cociente
    r = n%i # residuo
    if c > 1 and r == 0 :
        print n, 'no es primo'
        primo = 0
        break
    if i*i > n : break

if primo == 1: print n, ' es primo'
```

4.4. break

Por medio de `break` es posible salir de un bucle `for` o `while` desde un sitio intermedio del bucle. El control es transferido a la siguiente orden después del final del bucle. Generalmente `break` está dentro de un `if`. Si hay varios bucles anidados, `break` sale del bucle más interno.

```
for i in range(11) :
    print 'i = ', i
    if ( i*i )%10 == 9 :
        break
    print 'su cuadrado es: ', i*i

print '\nFin del ejemplo.\n\n'
```

4.5. `continue`

Por medio de `continue` es posible saltar una parte de un bucle, `for` o `while`, desde un sitio intermedio del bucle. El control es transferido para una nueva iteración. Generalmente `continue` está dentro de un `if`.

```
for i in range(11) :
    print 'i = ', i
    if ( i*i )%10 == 9 :
        continue
    print 'su cuadrado es: ', i*i

print '\nFin del ejemplo.\n\n'
```

4.6. Miscelánea

Como se vió anteriormente, el `for` clásico para los valores 1, 2, ..., n es

```
for i in range(1, n+1):
    ...
    ...
```

Lo anterior se puede simular por medio de un `while`:

```
i = 1
while i <= n :
    ...
    ...
    i += 1
```

Considere el siguiente ejemplo:

```
n = input('\n\nDigite n  entero mayor o igual a 1 :  ')

if n < 1 :
    print '\n\nn deber ser mayor o igual a  2.\n\n'
```

```
    quit()

import time

t0 = time.clock()
s = 0
i = 1
while i <= n :
    s += i
    i += 1
print 'la suma es: ', s
print 't con i += 1  :      ', time.clock() - t0

t0 = time.clock()
s = 0
for i in range(1, n+1):
    s += i
print 'la suma es: ', s
print 't con range(1,n+1) = ', time.clock() - t0
```

Las dos partes producen el mismo valor de `s`, pero hay dos observaciones:

- Con $n = 1234567890$, la primera parte funciona, pero la segunda (con `range`) no funciona, error de memoria por lista demasiado grande.
- Con $n = 123456789$, la primera gasta 72 segundos, la segunda 59. Conclusión, con `i in range ...` el proceso es ligeramente más rápido.

4.7. Estilo

Estas son algunas recomendaciones sobre el estilo para escribir programas en Python (tomado de Van Rossum G., *Tutorial Python, 2.5.2*, 2009).

- Usar sangría ("identación") de 4 espacios (no tabs). Con 4 espacios hay un buen compromiso entre sangría pequeña (permite mayor

nivel de sangría) y sangría grande (más fácil de leer). Los tabs introducen confusión y es mejor dejarlos de lado.

- Recortar las líneas para que no superen los 79 caracteres. Esto ayuda a los usuarios con pantallas pequeñas y hace posible tener varios archivos de código abiertos, uno al lado del otro, en pantallas grandes. Usar líneas en blanco para separar funciones y clases, y bloques grandes de código dentro de funciones.
- Cuando sea posible, poner comentarios en una sola línea.
- Usar docstrings (ver más adelante en documentación de funciones).
- Usar espacios alrededor de operadores y luego de las comas, pero no directamente dentro de paréntesis: `a = f(1, 2) + g(3, 4)`.
- Nombrar las clases y funciones consistentemente; la convención es usar `NotacionCamello` para clases y `minusculas_con_guiones_bajos` para funciones y métodos.
- Siempre usar `self` como el nombre para el primer argumento en los métodos.
- No usar codificaciones estrafalarias si se espera usar el código en entornos internacionales. ASCII plano funciona bien en la mayoría de los casos.

Capítulo 5

Funciones

La palabra funciones agrupa al mismo tiempo las funciones propiamente dichas, funciones en el sentido matemático, y los procedimientos. Una función, parecida a la función de matemáticas, dados unos datos o parámetros de entrada o argumentos, calcula uno o varios resultados o parámetros de salida. Una función o procedimiento no necesariamente calcula o devuelve algo y no necesariamente tiene parámetros de entrada.

```
import math

def cosg(x):
    # coseno de un angulo en grados
    t = x*math.pi/180
    return math.cos(t)
#--------------------------------------
def ordena(a, b) :
    # devuelve los dos numeros en orden creciente
    if a <= b :
        return a, b
    else :
        return b, a
#--------------------------------------
def lineas(n) :
    # escribe  n  lineas en blanco
    for i in range(1,n+1) : print '\n'
```

```
#--------------------------------------
def final() :
    print 'Este programa acabo normalmente.\n'
#--------------------------------------

a = input('entre un angulo en grados:   ')
print 'cos(', a, ') = ', cosg(a)

x = input('entre un numero:   ')
y = input('entre otro numero: ')
x, y = ordena(x, y)
print 'numeros ordenados: ', x, y

lineas(10)
final()
```

- Después de la palabra clave `def` (y de un espacio) va el nombre de la función. Inmediatamente después van los paréntesis y dentro de ellos uno o varios parámetros o ninguno. Después del paréntesis derecho debe haber dos puntos `:` .

- Por medio de la sangría se determina el cuerpo de la función.

- Por medio de `return` la función devuelve el valor o los valores calculados cuando los hay.

5.1. Funciones recurrentes

Un ejemplo típico es el correpondiente al cálculo del factorial. Recordemos que dado n, un número entero mayor o igual a cero, su factorial está dado por

$$0! = 1$$
$$1! = 1$$
$$n! = 1 \times 2 \times 3 \times \cdots \times (n-1) \times n$$
$$5! = 120$$

o también

$$n! = \begin{cases} 1 & \text{si } n \leq 1 \\ n(n-1)! & \text{si } n \geq 2. \end{cases} \tag{5.1}$$

Una función es recurrente si ella se llama a sí misma. Python permite la recurrencia, algunos lenguajes no. Veamos primero una implementación no recurrente.

```
# factorial no recurrente

def factnr(n):
    # calcula el factorial de un entero no negativo
    f = 1
    for i in range(2,n+1):
        f *= i
    return f
#-----------------------------------------------------
n = input('\n\nDigite un entero >= 0,  n =  ')
f = factnr(n)
print '\nn = ', n, ' su factorial es ', f, type(f)
```

Algunas observaciones:

- Mi calculadora científica de bolsillo únicamente puede calcular hasta 69!. Con 70! produce error. Observe que la función anterior calcula 100! sin problemas y da todas la cifras (100! no es el límite superior). ¡Oh sorpresa!
- Hasta 12! la función devuelve un entero, a partir de 13! es un entero largo.

Considere ahora otra función ligeramente diferente:

```
#!/usr/bin/python
# factorial flotante

def factf(n):
```

```
    # calcula el factorial de un entero no negativo
    f = 1.0
    for i in range(2,n+1):
        f *= i
    return f
#-----------------------------------------------------

n = input('\n\nDigite un entero >= 0,  n =  ')
f = factf(n)
print '\nn = ', n, 'su factorial es ', f, type(f)
```

Otras observaciones:

- Como era de esperarse el resultado es un numero `float`.
- En mi computador, con sus características físicas (hardware), la versión 2.7 de Python y con Kubuntu 11.04, al calcular 171! produce `7.25741561531e+306`. Al calcular 171! produce `inf`, es decir, un número flotante demasiado grande.
- La penúltima función `factnr` no tiene problemas para calcular 171! (por lo menos coincide con el valor encontrado en wikipedia). De nuevo, ¡oh sorpresa! El manejo de enteros largos es muy bueno. Yo siempre había creido en el dogma "para trabajar con números muy grandes, no use enteros, use números de punto flotante".

Ahora sí la versión recurrente:

```
# factorial recurrente

def factr(n):
    # calcul recurrente del factorial de n entero >= 0
    if n <= 1:
        return 1
    else:
        return n*factr(n-1)
#-----------------------------------------------------

n = input('\n\nDigite un entero >= 0,  n =  ')
```

```
f = factr(n)
print '\nn = ', n, 'su factorial es ', f, type(f)
```

Tadavía más observaciones:

- Esta función sigue a la letra la definición (5.1).
- En este caso, cálculo de $n!$, las dos funciones (la no recurrente y la recurrente) son igualmente fáciles, pero en algunos casos un proceso se puede expresar recurrentemente de manera mucho más sencilla que sin usar la recurrencia.
- La facilidad en la programación y la elegancia tienen un costo. La función `factr` funciona bien con $n = 999$ pero con $n = 1000$ produce error por exceder el máximo nivel de recurrencia permitido. La función `factnr` aparentemente calcula sin problema 1000!.

Ahora una mirada al tiempo (no en el sentido meteorológico). Consideremos el siguiente ejemplo:

```
def factr(n):
    # calculo recurrente del factorial de n entero >= 0
    if n <= 1:
        return 1
    else:
        return n*factr(n-1)
#-------------------------------------------------
def factnr(n):
    # calcula el factorial de un entero no negativo
    f = 1
    for i in range(2,n+1):
        f *= i
    return f
#-------------------------------------------------
def factf(n):
    # calcula el factorial de un entero no negativo
    f = 1.0
    for i in range(2,n+1):
        f *= i
```

```
    return f
#=============================================
n = 170
nRepet = 200000

import time

t0 = time.clock()
for i in range(1, nRepet+1):
    f = factnr(n)

t = time.clock()-t0
print 'tiempo no recurrente = ', t

t0  = time.clock()
for i in range(1, nRepet+1):
    f = factr(n)

t  = time.clock()-t0
print 'tiempo recurrente =    ', t

t0  = time.clock()
for i in range(1, nRepet+1):
    f = factf(n)

t  = time.clock()-t0
print 'tiempo flotante =      ', t
```

En mi computador, el resultado es:

```
    tiempo no recurrente =  17.53
    tiempo recurrente =     27.22
    tiempo flotante =        7.34
```

Por fin un punto a favor del cálculo con números de punto flotante. El programador debe tener en cuenta la facilidad de desarrollo (escritura del programa), el tiempo de ejecución y las restricciones de tamaño para tomar, en cada caso, el camino que más le convenga.

5.2. Control de parámetros de entrada

Una buena función debe controlar los parámetros de entrada. Deben ser del tipo esperado y cumplir las otras condiciones esperadas. Por ejemplo, para el factorial de n, se debe tener un entero y debe ser mayor o igual a cero. Entonces después del llamado y ejecución de la función, debe ser posible saber si la función trabajó normalmente o no. Esto se puede hacer por un indicador o por un valor absurdo. Por ejemplo,

```
def fact(n):
    #
    # calcula el factorial de un entero no negativo
    #
    if type(n) != type(1) and type(n) != type(1L):
        print '\nn  debe ser un entero.\n\n'
        return 0
    if n < 0:
        print n, '  es un numero negativo.\n'
        return 0
    f = 1
    for i in range(2,n+1):
        f *= i
    return f
#====================================================

n = 5
f = fact(n)
if f > 0:
    print '\nEl factorial de', n, 'es: ', f

n = -2
f = fact(n)
if f > 0:
    print '\nEl factorial de', n, 'es: ', f

n = 9.9
f = fact(n)
if f > 0:
```

```
    print '\nEl factorial de', n, 'es: ', f

n = '10'
f = fact(n)
if f > 0:
    print '\nEl factorial de', n, 'es: ', f
```

O también

```
def fact_b(n):
    #
    # calcula el factorial de un entero no negativo
    #
    info = 0
    if type(n) != type(1) and type(n) != type(1L):
        print '\nn  debe ser un entero.\n\n'
        return 0, info
    if n < 0:
        print n, '  es un numero negativo.\n'
        return 0, info
    f = 1
    info = 1
    for i in range(2,n+1):
        f *= i
    return f, info
#================================================

n = 5
nf, indic = fact_b(n)
if indic == 1:
    print '\nEl factorial de', n, 'es: ', nf
```

5.3. Función parámetro

Una función también puede ser un parámetro de otra función. El siguiente ejemplo está basado en la fórmula del trapecio para aproximar

el valor de la integral definida de una función.

$$\int_a^b f(x)\,dx \approx h\left(\frac{f(a)+f(b)}{2}+\sum_{i=1}^{n-1} f(x_i)\right)$$
$$h=\frac{b-a}{n}$$
$$x_i = a+ih$$

```
import math

def trapecio(f, a, b, n) :
    # formula del trapecio para aproximar la integral
    # de f entre a y b con n subintervalos

    h = (b-a)/float(n)
    s = ( f(a) + f(b) )/2.0
    xi = a + h
    for i in range(1, n):
        s = s + f(xi)
        xi += h
    return s*h
#-------------------------------------------------------
def f01(x):
    return math.exp(-x*x)
#-------------------------------------------------------
def f02(x):
    return 1.0/( 1.0+float(x)*x )
#-------------------------------------------------------
def f03(x):
    return x
#=======================================================
print trapecio(f01, -0.5, 0.5, 20)
print trapecio(f02, -0.5, 0.5, 20)
print trapecio(f03, 0, 1, 2)
print trapecio(math.sin, 0, math.pi, 1000)
```

El primer parámetro de la función `trapecio` es justamente la función. Ésta puede ser una función definida en el programa u otra función de un módulo de Python.

5.4. Alcance de un variable

De manera informal, las variables definidas en una función solamente existen dentro de la función. En el siguiente ejemplo, hay dos variables `t`, una dentro de la función que no afecta en nada la `t` definida fuera de la función. Por esto, inicialmente `t` vale 10, cuando se utiliza la función `f06`, se crea otra variable `t` que deja de existir tan pronto acaba el uso de `f06`. La `t` exterior a la función no cambió su valor.

```
def f06(x):
    t = (x-0.5)*(x-0.5)
    print 't en f06:', t
    return t*t - t + 5
#=====================================================
x = 2.5
t = 10
print 't antes: ', t
y = f06(x)
print 'f(x) = ', y
print 't despues: ', t
```

5.5. Documentación

Hasta ahora, en los ejemplos de funciones, éstas tenían una pequeña documentación informal por medio de líneas de comentarios. Esta documentación es útil únicamente para quien tenga acceso al código de la función.

Python tiene una forma más elaborada para documentar una función, sin tener que mirar explícitamente el código de la función. Se hace con por medio de una `docstring`.

```
def trapecio(f, a, b, n) :
    """Formula del trapecio para la integral."""

    h = (b-a)/float(n)
    s = ( f(a) + f(b) )/2.0
```

```
    xi = a + h
    for i in range(1, n):
        s = s + f(xi)
        xi += h
    return s*h
#--------------------------------------------------
def trapecio2(f, a, b, n) :
    """Formula del trapecio para la integral.

    Valor aproximado de la integral de  f  entre a y b
    utilizando  n  subintervalos.
    """

    h = (b-a)/float(n)
    s = ( f(a) + f(b) )/2.0
    xi = a + h
    for i in range(1, n):
        s = s + f(xi)
        xi += h
    return s*h
#==================================================
print trapecio.__doc__
print trapecio2.__doc__
```

- La *docstring* está o empieza en la primera línea después de `def`.
- La *docstring* empieza y acaba con tres comillas dobles.
- Se obtiene la documentación por medio de `funcion.__doc__` (doble guión bajo `doc` doble guión bajo).

  En el tutorial de Van Rossum, para *docstrings* multilínea, se sugiere hacerlo como está en la función `trapecio2`.

  - La primera línea de la *docstring* empieza con las tres comillas dobles.
  - Después sigue una explicacion corta que empieza con mayúscula y acaba con punto.
  - Después hay una línea en blanco.

- Después hay una o varias líneas con explicaciones más detalladas.
- Finalmente una línea con tres comillas dobles.

5.6. Parámetros por defecto

Una manera ineficiente, pero sencilla, de obtener una aproximación del minimizador de una función en el intervalo $[a, b]$, consiste en evaluar la función en $x = a$, $x = a + h$, $a + 2h$, $a + 3h$, ... y escoger el valor de x correspondiente al mínimo de los valores de f. Considere el siguiente ejemplo:

```
import math

def minf(f, a,  b, h = 0.01 ):
    # metodo sencillo pero poco eficiente para
    # obtener el minimizador de una funcion
    # en el intervalo  [a, b]
    xmin = a
    fmin = f(a)
    x = a+h
    #print 'h = ',h
    while x <= b :
        fx = f(x)
        if fx < fmin :
           xmin = x
           fmin = fx
        x = x+h
    return xmin, fmin
#-------------------------------------------------------
def f05(x):
    return x**5 - 20*x**4 + 10*x + 1 + math.cos(x)
#=======================================================
xopt, vmin = minf(f05, 10, 20)
print 'minimizador = ', xopt, '  f(x) = ', vmin

xopt, vmin = minf(math.cos, 0, 4 , 0.000001)
```

```
print 'minimizador = ', xopt, '  f(x) = ', vmin
```

El quinto parámetro de la función `minf`, está definido por defecto con el valor 0.01. En el primer llamado a la función `minf`, no hay quinto parámetro, entonces `minf` utilizará $h = 0.01$. En el segundo llamado, `minf` utilizará 0.000001.

Capítulo 6

El módulo `turtle`

Este módulo permite usar Python de manera muy parecida al lenguaje de programación Logo. Éste fue creado en 1967 por Papert, Bobrow y otros, con fines educativos (enseñanza de la programación para niños, ...).

Se supone que una tortuga, siguiendo órdenes muy precisas, se desplaza por la pantalla y va dibujando o dejando huella con su cola.

6.1. Primeras ordenes

Supongamos que no habrá conflicto con funciones de otros módulos. Entonces se puede importar mediante:

```
from turtle import *
```

Si se da la orden

```
>>>  forward(100)
```

se abre una nueva ventana, en ella la tortuga ha dibujado un segmento de recta cuya longitud es 100 pixeles (o píxeles).

Al dar la orden `right(90)`, la punta de la flecha gira hacia la derecha 90 grados.

Órdenes análogas a las anteriores son: `back(50), left(60)`.

6.2. Más órdenes

Otras órdenes sencillas son:

```
reset()
up()
down()
color('red')
width(3)
speed(4)
goto(-20,200)
```

Por medio de `reset()` se borra lo dibujado en la pantalla hasta el momento y la tortuga se vuelve a situar en el centro de la ventana. Si se utiliza la orden `up()`, la tortuga levanta su cola. Esto implica que ahora cuando la tortuga se desplaza no deja huella (no dibuja nada). Para que la tortuga vuelva a dibujar se requiere utilizar `down()`. Por ejemplo,

```
a = 10
for i in range(20):
    forward(a)
    up()
    forward(a)
    down()
```

dibuja una línea discontinua a trazos.

La línea trazada por la tortuga es, por defecto, negra. A partir del momento en que se use `color('red')` la línea será roja. Otras opciones son `'yellow'`, `'blue'`, `'green'`, `'black'`, `'white'`, `'gray'`, `'orange'`, `'brown'`, `'beige'`, `'violet'`, `'pink'`, `'magenta'`, `'cyan'`, `'purple'`.

Con `width` se modifica el ancho de la línea y con `speed` la velocidad de la tortuga. Por medio de `goto(x, y)` se desplaza la tortuga al

punto de coordenadas (x, y). Si la ventana está en su tamaño máximo, en un caso usual, la huella de la tortuga es visible si $-793 \leq x \leq 793$ y $-419 \leq y \leq 419$. La altura y anchura de la ventana se pueden conocer por medio de `window_height()` y `window_width()`.

6.3. Ayuda

Se puede obtener información general sobre `turtle` o específica de alguna función mediante:

- Entrar al ambiente Python.
- `>>> from turtle import *`
- `>>> help()`
- `help> turtle`

Aparece entonces un documento grande, bastante completo, con información sobre `turtle`, las clases, las funciones, etc. Por medio de las flechas o de las teclas PGUP o PGDN se puede avanzar o retroceder dentro del documento. Para salir de este documento se oprime la tecla `q` y para salir de la ayuda mediante `quit`.

También se puede buscar directamente información más específica:

- Entrar al ambiente Python.
- `>>> from turtle import *`
- `>>> help(forward)`

6.4. Lista de funciones

```
Move and draw
    forward() | fd()
    backward() | bk() | back()
    right() | rt()
```

```
    left() | lt()
    goto() | setpos() | setposition()
    setx()
    sety()
    setheading() | seth()
    home()
    circle()
    dot()
    stamp()
    clearstamp()
    clearstamps()
    undo()
    speed()
Tell Turtle's state
    position() | pos()
    towards()
    xcor()
    ycor()
    heading()
    distance()
Setting and measurement
    degrees()
    radians()
Drawing state
    pendown() | pd() | down()
    penup() | pu() | up()
    pensize() | width()
    pen()
    isdown()
Color control
    color()
    pencolor()
    fillcolor()
Filling
    fill()
    begin_fill()
    end_fill()
More drawing control
```

```
    reset()
    clear()
    write()
Visibility
    showturtle() | st()
    hideturtle() | ht()
    isvisible()
Appearance
    shape()
    resizemode()
    shapesize() | turtlesize()
    settiltangle()
    tiltangle()
    tilt()
Using events
    onclick()
    onrelease()
    ondrag()
Special Turtle methods
    begin_poly()
    end_poly()
    get_poly()
    clone()
    getturtle() | getpen()
    getscreen()
    setundobuffer()
    undobufferentries()
    tracer()
    window_width()
    window_height()

Window control
    bgcolor()
    bgpic()
    clear() | clearscreen()
    reset() | resetscreen()
    screensize()
    setworldcoordinates()
```

```
Animation control
    delay()
    tracer()
    update()
Using screen events
    listen()
    onkey()
    onclick() | onscreenclick()
    ontimer()

Settings and special methods
    mode()
    colormode()
    getcanvas()
    getshapes()
    register_shape() | addshape()
    turtles()
    window_height()
    window_width()
Methods specific to Screen
    bye()
    exitonclick()
    setup()
    title()
```

6.5. Algunos ejemplos

6.5.1. Un cuadrado sencillo

```
from turtle import *

def cuadrado1(x):
    i = 1
    while i <= 4:
        forward(x)
        right(90)
        i = i+1
```

```
a = 200
reset()
cuadrado1(a)
up()
goto(30,30)
down()
cuadrado1(a)
up()
goto(60,60)
down()
cuadrado1(a)
```

6.5.2. Un cuadrado con más parámetros

```
from turtle import *

def cuadrado(dir_ini, der_izq, lado, tono):
    # dibuja un cuadrado
    # dir_ini  :  direccion inicial en  grados
    # derecha o izquierda:  'd'  o  'i'
    # lado : medida
    # tono: 'red', 'blue', 'green', ...
    setheading(dir_ini)
    color(tono)
    i = 1
    while i <= 4:
        forward(lado)
        if der_izq == 'd':
            right(90)
        else:
            left(90)
        i = i+1

reset()
speed(50)
width(3)
a = 100
```

```
t = 0
while t <= 360:
    cuadrado(t, 'i', a, 'red')
    t = t+10
```

Capítulo 7

Clases

Una clase agrupa un conjunto de datos (variables) con un conjunto de funciones que operan sobre los datos. Se usan para obtener programas más modulares agrupando datos y funciones en unidades fácilmente manejables.

Muchos de los cálculos matemáticos o numéricos se pueden hacer fácilemente sin usar clases. Sin embargo, en algunos problemas el uso de clases da un manejo más elegante y abre la posibilidad de extender más fácilmente.

De manera un poco más general, una clase es un conjunto de atributos asociados con una colección de objetos llamados *instancias*. Entre los atributos están:

- las funciones llamadas *métodos*,
- las variables llamadas *variables de clase*,
- los atributos calculados llamados *propiedades*.

7.1. Primer ejemplo

Usualmente el nombre de una clase empieza con mayúscula. El siguiente es un ejemplo sencillísimo de la clase Punto, la clase de los puntos en

el plano. Poco a poco se irá completando el ejemplo, para que la clase Punto sea realmente útil.

```
class Punto:
    "Clase de los puntos del plano"

p = Punto()
q = punto()
```

El programita anterior funciona, `p` y `q` son instancias de la clase Punto. No produce nada, absolutamente nada. Ahora bien,

```
class Punto:
    "Clase de los puntos del plano"

p = Punto()
q = punto()
print p
print q
print p.__doc__
```

Este programa produce un resultado parecido al siguiente:

```
<__main__.Punto instance at 0xb732d96c>
<__main__.Punto instance at 0xb732d98c>
Clase de los puntos del plano
```

Este resultado muestra que se trata de dos instancias de la clase Punto con sus direcciones. También aparece la información sobre la clase.

Es posible agregar nuevos datos a una instancia mediante la notación punto.

```
p.x = -3
p.y = 4

q.r = 5
q.a = 2.2143
```

```
import math
d = math.sqrt(p.x**2 + p.y**2)
print 'd = ', d
```

Lo anterior funciona pero, salvo casos especiales, no es recomendable. En este ejemplo dos instancias de la misma clase tienen datos adicionales con diferente nombre.

7.2. init y repr

Consideremos la siguiente versión de la clase:

```
import math

class Punto:
    "Clase de los puntos del plano"
    def __init__(self, abcisa = 0, ordenada = 0):
        self.x = abcisa
        self.y = ordenada
    #.........................................
    def __repr__(self):
        return '(' + str(self.x) + ', ' + str(self.y) + ')'
    #.........................................
    def dist0(self):
        """Distancia al origen o norma."""
        return math.sqrt( self.x**2 + self.y**2 )
#-----------------------------------------------
p = Punto()
q = Punto(-3, 4)
print p, q
print 'distancia al origen: ', q.dist0()
```

El resultado del programa anterior es:

```
(0, 0) (-3, 4)
distancia al origen:  5.0
```

Características del uso del método especial `__init__()`:

- Permite precisar los atributos de los objetos de la clase. Todas las instancias de la clase Punto tienen dos atributos: `x` , `y`.

- Al hacer la instanciación (por ejemplo, `p = Punto()`) no se crea un objeto vacío (primer ejemplo) sino con atributos dados como parámetros o por defecto. No es necesario dar parámetros por defecto pero en este ejemplo los hay y tienen el valor cero.

- El primer parámetro de `init` corresponde a la instancia que se está creando. Es usual y aconsejable llamarlo `self` pero se hubiera podido llamar `cosita`.

Mediante el uso del método especial `__repr__()` se construye una cadena que será el resultado de `print`. Para el ejemplo, al usar `print`, aparece la pareja ordenada (x, y).

7.3. Sobrecarga de operadores

Se habla de sobrecarga de un operador cuando se usa, para otras operaciones, un operador ya definido. Por ejemplo el operador `+` se usa para sumar números. En Python el `+` está sobrecargado y se usa entre cadenas (las concatena). También se puede sobrecargar para usarlo con nuevos objetos.

Supongamos ahora que para puntos del plano están definidas las siguientes operaciones: suma de dos puntos, resta de dos puntos, multiplicación entre un número y un punto, división de un punto por un número, multiplicación (producto punto o producto interior) entre dos puntos, inverso aditivo de un punto (menos el punto) con la siguiente notación y signi-

ficado (el asterisco indica multiplicación):

$$
\begin{aligned}
(a,b) + (c,d) &= (a+c, b+d) \\
(a,b) - (c,d) &= (a-c, b-d) \\
k * (a,b) &= (k*a, k*b) \\
(a,b) * k &= (k*a, k*b) \\
(a,b)/k &= (a/k, b/k) \\
(a,b) * (c,d) &= a*c + b*d \\
-(a,b) &= (-a, -b)
\end{aligned}
$$

```
class Punto:
    "Clase de los puntos del plano"
    def __init__(self, abcisa = 0, ordenada = 0):
        self.x = abcisa
        self.y = ordenada

    def __repr__(self):
        return '(' + str(self.x) + ', ' + str(self.y) + ')'

    def dist0(self):
        """Distancia al origen o norma."""
        return math.sqrt( self.x**2 + self.y**2 )

    def __add__(self, v):
        return Punto( self.x + v.x, self.y + v.y )

    def __sub__(self, v):
        return Punto( self.x - v.x, self.y - v.y )

    def __mul__(self, w):
        # print 'entrando: self, w: ', self, w
        # self siempre sera Punto,
        # aun si en el llamado es el segundo operador.

        if isinstance(w, int) or isinstance(w, float):
            return Punto( w*self.x, w*self.y)
```

```
        else:
            return self.x*w.x + self.y*w.y

    __rmul__ = __mul__

    def __div__(self, k):
        fk = float(k)
        return Punto( self.x/fk,   self.y/fk )

    def __neg__(self):
        return Punto( -self.x,  -self.y )
#----------fin clase Punto---------------------------------
import math
p = Punto(-1, 2)
q = Punto(3, -4)
k = 0.5
print 'p = ', p, '   q = ', q, '   k = ', k
print 'p + q = ', p + q
print 'p - q = ', p - q
print 'k*p = ', k*p
print 'p*k = ', p*k
print 'p*q = ', p*q
print 'p/k = ', p/k
print '-p = ', -p
print 'd = ', math.sqrt(q*q)
```

El resultado es:

```
p =  (-1, 2)    q =  (3, -4)    k =  0.5
p + q =  (2, -2)
p - q =  (-4, 6)
k*p =  (-0.5, 1.0)
p*k =  (-0.5, 1.0)
p*q =  -11
p/k =  (-2.0, 4.0)
-p =  (1, -2)
d =  5.0
```

Es usual, conveniente y de mejor estilo, pero no absolutamente indis-

pensable, que el primer parámetro de un método sea `self`.

El uso de `__add__` permite que se use `+` (el signo más) para sumar dos instancias de la clase Punto. Su definición es casi inmediata.

El símbolo asterisco se va a usar para tres operaciones diferentes (número por punto, punto por número, punto por punto). La definición de `__mul__` es un poco más compleja. En la definición fue necesario detectar cuando el otro parámetro es un número, entero o flotante.

La línea

```
__rmul__ = __mul__
```

permite el producto de un número por un Punto, `k*p`. La abreviación `rmul` se refiere a *right multiplication*.

7.4. Terminología

En palabras sencillas:

- Una *clase* es equivalente a un nuevo tipo de datos.
- Un *objeto* o una *instancia* es un ejemplar particular de una clase.
- Una clase puede *encapsular* al mismo tiempo, datos y métodos aplicables a los objetos. Un objeto es un *cápsula* o paquete que contiene atributos y métodos.
- Un *atributo* es un dato de un objeto. Por ejemplo `.x` y `.y` son atributos de la clase Punto.
- Un *método* es una función aplicable al objeto. Por ejemplo `dist0` es un método de la clase Punto.
- Algunas clases tienen *métodos especiales* cuyos nombres están predefinidos. Estos nombres pueden ser usados por varias clases al tiempo con significados análogos o aún diferentes. Por ejemplo utilizando `__add__` se sobrecarga el operador `+`.
- La *herencia* es un mecanismo que permite utilizar una clase ya existente para crear una nueva con funcionalidades diferentes o complementarias. Ver más adelante.

- Mediante el *polimorfismo* dos o más métodos de clases diferentes tienen igual nombre y realizan acciones casi iguales, parecidas o aún completamente diferentes. Ver más adelante.

7.5. Un ejemplo de Teoría de Números: $\mathbb{Z}_5$

Antes de entrar en materia, consideremos, para a entero y b entero positivo, la siguiente notación, igual a la de Python:

$$a \,\%\, b = \text{residuo (entero) de la división entera } a \div b \tag{7.1}$$

Por ejemplo,

$$\begin{aligned} 23 \,\%\, 5 &= 3 \\ 30 \,\%\, 5 &= 0 \\ (-23) \,\%5 &= 2 \end{aligned}$$

Sea

$$\mathbb{Z}_5 = \{0, 1, 2, 3, 4\}$$

En este conjunto se definen dos operaciones, adición y multiplicación, mediante las tablas

$+$	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

$\cdot$	0	1	2	3	4
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	1	3
3	0	3	1	4	2
4	0	4	3	2	1

Estas tablas no son arbitrarias, se construyeron mediante:

$$a + b = (a + b)\,\%5$$
$$a \cdot b = (a \cdot b)\,\%5,$$

donde el signo + de la izquierda es en $\mathbb{Z}_5$, el + de la derecha es en $\mathbb{Z}$. De manera análoga para la multiplicación.

Por las propiedades de los enteros, en $\mathbb{Z}_5$ también se cumplen la conmutatividad y la asociatividad. En la tabla se observa que hay elemento identidad para la suma, obviamente el 0, y elemento identidad para el producto, el 1. En la tabla, también se observa que todos los elementos de $\mathbb{Z}_5$ tienen inverso para la suma Por ejemplo, el inverso aditivo de 4 es 1. También se observa que todos los elementos de $\mathbb{Z}_5$, salvo el 0, tienen inverso para la multiplicación. Por ejemplo, el inverso multiplicativo de 2 es 3.

Lo anterior permite definir la resta entre dos números de $\mathbb{Z}_5$ y la división por un número no nulo:

$$a - b = a + \text{inverso-aditivo}(b)$$
$$a/b = a \cdot \text{inverso-multiplicativo}(b) \quad \text{si } b \neq 0.$$

El inverso aditivo se obtiene muy fácilmente, si $x \in \mathbb{Z}_5$,

$$\text{inverso-aditivo}(x) = 5 - x.$$

Como todo número no nulo tiene inverso multiplicativo, una manera de obtenerlo es mediante una búsqueda uno por uno.

El siguiente ejemplo presenta una implementación de los números de $\mathbb{Z}_5$:

```
def inv_i( x):
    """Inverso en Z5.
    Dado un entero x, devuelve un entero en [1, 4]
    tal que el producto sea 1.
    O devuelve  0  cuando no es posible."""

    P = 5
```

```
    x0 = x
    x = x%P
    if x == 0:
        print x0, 'no tiene inverso en Z5'
        return 0

    for i in range(1, P):
        if (x*i)%P == 1:
            return i

class Z5:
    """Clase de los enteros modulo 5"""
    def operators(self):
        pass

    def __init__(self, n = 0):
        if isinstance(n, int):
            self.i = n%5
        else:
            print 'Error en Z5:', n, 'deberia ser un entero.'
            self.i = 0

    def __repr__(self):
        """Construye la cadena, representacion
        'oficial'."""
        return str(self.i)

    def __add__(self, y):
        if isinstance(y, int):
            return Z5( (self.i + y)%5 )
        else:
            return Z5( (self.i + y.i )%5 )


    def __radd__(self, y):
        return self + y

    def __mul__(self, y):
```

```
        if isinstance(y, int):
            return Z5( (self.i * y)%5 )
        else:
            return Z5( (self.i * y.i )%5 )

    def __rmul__(self, y):
        return self * y

    def __neg__(self):
        return Z5( -self.i )

    def __sub__(self, y):
        return  self + (-y)

    def __rsub__(self, y):
        return  Z5( y + (-self.i) )

    def __div__(self, y):
        if isinstance(y, int):
            # z5 / entero
            r = ( self.i*inv_i(y) )%5
        else:
            # z5_1 / z5_2
            r = ( self.i*inv_i(y.i) )%5
        return Z5(r)

    def __rdiv__(self, y):
        if isinstance(y, int):
            # entero / z5
            r = ( y*inv_i(self.i) )%5
        else:
            # z5_2 / z5_1
            r = ( y.i*inv_i(self.i) )%5
        return  Z5( r )
#================================================
print '\n\n\n\n\n'

w = Z5(24.9)
```

```
x = Z5(24)
y = Z5(-22)
z = Z5()
c = 21
d = -113

print 'x =', x, '  y =', y, '  z =', z, '  w =', w
print 'c =', c, '  d =', d

print 'x + y = ', x + y
print 'x + c = ', x + c
print 'd + x = ', d + x

print 'x*y = ', x*y
print 'x*c = ', x*c
print 'd*x = ', d*x

print '-x =', -x, '  -y =', - y

print 'x - y =', x - y
print 'y - x =', y - x
print 'x - c =', x - c

print 'c - x =', c - x

print 'x/y =', x/y
print 'y/x =', y/x
print 'x/c =', x/c
print 'c/x =', c/x
```

7.6. Herencia y polimorfismo

En el siguiente ejemplo, adaptado de [Cor], inicialmente se crea la clase rectángulo llamada `Rectang`. Uno de sus métodos es la función `area`. A partir de `Rectang`, se crea la subclase o hija, la clase `Cuadrado`. Esta última clase hereda el método `area` de la clase rectángulo. Hay polimorfismo entre el método `__init__` de un rectángulo y el método `__init__`

de un cuadrado.

```
class Rectang:
    """Rectangulo."""

    def __init__(self, base = 6, altura = 4):
        self.b = base
        self.h = altura

    def __repr__(self):
        return 'medidas: ' + str(self.b) +  ' , ' + str(self.h)

    def area(self):
        """Area del rectangulo."""
        return self.b * self.h

class Cuadrado(Rectang):
    """Cuadrado."""
    def __init__(self, lado = 5):
        Rectang.__init__(self, lado, lado)

s = Rectang()
t = Rectang(20,10)
u = Cuadrado()
v = Cuadrado(3)

print s, '  Area = ', s.area()
print t, '  Area = ', t.area()
print u, '  Area = ', u.area()
print v, '  Area = ', v.area()
```

El anterior program produce los siguientes resultados:

```
medidas: 6 , 4   Area =  24
medidas: 20 , 10   Area =  200
medidas: 5 , 5   Area =  25
medidas: 3 , 3   Area =  9
```

7.7. Otro ejemplo de herencia

```
class Punto:
    """Punto del plano."""

    def __init__(self, abcisa = 0, ordenada = 0):
        self.x = abcisa
        self.y = ordenada

    def __repr__(self):
        cadena = '('+str(self.x) + ', ' + str(self.y) + ')'
        return cadena
#-------------------------------------------------
class Circulo(Punto):
    """Circulo: centro y radio."""

    def __init__(self, h = 0, k = 0, radio = 1):
        Punto.__init__(self, h, k)
        self.r = radio

    def __repr__(self):
        cadena = '(' + str(self.x) + ', ' + str(self.y) + ' ), ' + str(self.r)
        return cadena
#=================================================
u = Punto()
v = Punto(-3, 4)
print u, v

q0 = Circulo()
q1 = Circulo(1.1)
q2 = Circulo(1.1, 1.2)
q3 = Circulo(1.1, 1.2, 1.3)
print q0
print q1
print q2
print q3
```

Este programa produce los siguientes resultados:

```
(0, 0) (-3, 4)
(0, 0 ), 1
(1.1, 0 ), 1
(1.1, 1.2 ), 1
(1.1, 1.2 ), 1.3
```

[Lan09] Langtangen H.P., *A Primer on Scientific Computing using Python*, Springer, New York, 2009.

[Cor] Cordeau B., *Introduction à Python 3*, version 2.71828, `www.iut-orsay.fr` y `www.creativecommons.org`.