Notas del curso

# Programación en Python

Curso de actualización docente
Facultad de Ciencias

David P. Sanders

16 de junio de 2010

# Capítulo 1

# ¿Qué es Python?

Python es un lenguaje de programación que surgió en 1990. Desde entonces se ha desarrollado enormemente, para volverse un lenguaje de programación moderno, y uno de los más utilizados en el mundo.

Python es un lenguaje *interpretado* –no es necesario compilar constantemente cada programa, si no se puede utilizar de manera interactiva. Por lo tanto, es una herramienta idónea para llevar a cabo tareas de cómputo científico, tales como análisis de datos, graficación de resultados, y exploración de problemas. En este sentido, se puede comparar con programas comerciales tales como Matlab. También hay paquetes que permiten llevar a cabo cálculos simbólicos a través de Python, de los cuales destaca `sage` (`www.sagemath.org`), por lo cual también se puede considerar como competidor de Mathematica y Maple.

Python viene con una filosofía de "baterías incluidas". Esto quiere decir que hay muchas bibliotecas disponibles que están diseñadas para facilitarnos la vida al hacer diferentes tareas, desde el cómputo científico hasta la manipulación de páginas web. Por lo tanto, Python es un lenguaje sumamente versátil.

Cabe enfatizar que se suele encontrar Python fácil de aprenderse y de utilizarse. Por lo tanto, considero que Python es también el lenguaje idóneo para la enseñanza del cómputo científico en la Facultad de Ciencias, un campo que se ha vuelto de suma importancia.

## 1.1. Meta del curso

La meta principal de este curso es la de enseñar las técnicas básicas de Python en el contexto del cómputo científico, y en particular de la física computacional, con un fin de actualización docente en la Facultad de Ciencias.

# Capítulo 2

# Comenzando con Python

En este capítulo, empezaremos a explorar el lenguaje Python. Se darán las bases tanto de la sintáxis básica, como de las herramientas principales para escribir programas en el lenguaje. En todo el curso se hará énfasis en las aplicaciones al cómputo científico y a la enseñanza del mismo.

La versión de Python que ocuparemos es la 2.6. La nueva versión 3 rompe la compatibilidad con los paquetes básicos para el cómputo científico por el momento.

## 2.1. El entorno `ipython`

El entorno principal que ocuparemos para trabajar de manera interactiva es `ipython`. Otro posible es `idle`.

`ipython` está diseñado para maximar la productividad al utilizar Python. En particular, tiene ciertos comandos que no existen en otros interpretadores de Python. Uno muy útil es

```
In [1]: logstart primero_log.py
```

Este comando graba un "log" (bitácora) de todo lo que se teclea en la sesión actual en el archivo dado, aqui `primero_log.py`.

Otra función de suma utilidad en `ipython` es que la de completar palabras parciales con la tecla `<TAB>`. Además, los comandos `who` y `whos` proveen información de las funciones nuevas agregadas en la sesión.

La documentación para cualquier función o comando está disponible en `ipython` al poner `?`, por ejemplo

```
who?
```

Para funciones, `??` muestra el código fuente de la función, cuando está disponible.

## 2.2. Aritmética

Python se puede utilizar como una calculadora:

```
3
3 + 2
3 * (-7.1 + 10**2)
```

Aquí, `**` indica una potencia, como en Fortran, y las paréntesis alteran la prioridad en las operaciones.

Es necesario[1] tener cuidado al hacer manipulaciones con enteros:

```
1 / 2
```

[1] En Python 2.6; la situación cambia en Python 3.

```
3 / 2
5 / 2
```

ya que el resultado se redondea al entero más cercano. Para evitar eso, basta un punto decimal:

```
1 / 2.
```

Los números sin punto decimal se consideran enteros, y los con punto decimal flotantes (de doble precisión). Los enteros pueden ser *arbitrariamente grandes*:

```
2 ** 2 ** 2 ** 2 ** 2
```

Python cuenta con números complejos, utilizando la notación `1j` para la raiz de $-1$:

```
1 + 3j
1j * 1j
j    # da un error
```

'`#`' indica un comentario que se extiende hasta el final de la línea.

## 2.3. Variables

Para llevar a cabo cálculos, necesitamos *variables*. Las variables se declaran como debe de ser:

```
a = 3
b = 17.5
c = 1 + 3j

print a + b / c
```

Python reconece de manera *automática* el tipo de la variable según su forma. El comando `print` imprime su argumento de una manera bonita.

Al reasignar una variable, se pierde la información de su valor interior, incluyendo el tipo:

```
a = 3
a = -5.5
```

## 2.4. Entrada por parte del usuario

Se puede pedir información del usuario con

```
a = raw_input('Dame el valor de a: ')
print "El cuadrado de a es ", a*a
```

Las cadenas en Python son cadenas de caracteres entre apóstrofes o comillas.

## 2.5. Listas

La estructura de datos principal en Python es la *lista*. Consiste literalmente en una lista ordenada de cosas, y reemplaza a los arreglos en otros lenguajes. La diferencia es que las listas en Python son automáticamente de longitud *variable*, y pueden contener objetos de *cualquier* tipo.

Una lista se define con corchetes (`[` y `\`):

```
l = [3, 4, 6]
```

Puede contener cualquier cosa, ¡incluyendo a otras listas!:

```
l = [3.5, -1, "hola", [1.0, [3, 17]]]
```

Por lo tanto, es una estructura de datos muy sencillo.

Los elementos de la lista se pueden extraer y cambiar usando la notación `l[i]`, donde `i` indica el número del elemento, empezando desde 0:

```
l = [1, 2, 3]
print l[0], l[1]
l[0] = 5
l
```

Se pueden manipular rebanadas ("slices") de la lista con la notación `l[1:3]`:

```
l = [1, 2, 3, 6, 7]
l[1:3]
l[:3]
l[3:]
```

**Ejercicio:** ¿Qué hace `l[-1]`?

La longitud de una lista se puede encontrar con

```
len(l)
```

Se pueden agregar elementos a la lista con

```
l = []    # lista vacia
l.append(17)
l.append(3)
print l, len(l)
```

Como siempre, las otras operaciones que se pueden llevar a cabo con la lista se pueden averiguar en `ipython` con `l.<TAB>`.

## 2.6. *n*-adas / tuples

Otra estructura parecida es una *n*-ada ("tuple"). Es parecido a una lista, pero se escribe con (o incluso, a veces, sin paréntesis), y no se puede modificar:

```
t = (3, 5)
t[0]
t[0] = 1        # error!
```

Las *n*-adas se utilizan para agrupar información. Se pueden asignar como sigue:

```
a, b = 3, 5
print a; print b
```

## 2.7. La biblioteca estándar

Python tiene una biblioteca estándar amplia, lo cual siempre está disponible en cualquier instalación de Python. La documentación para esta biblioteca está disponible en `http://docs.python.org/library/`

Por ejemplo, la sección 9 incluye información sobre el módulo `fractions`:

```
from fractions import Fraction

a = Fraction(3, 5)
```

```
b = Fraction(1)
c = a + b
d = Fraction(4, 6)
d
e = a + d
```

Cada 'cosa' en Python es un *objeto*, que tiene propiedades y operaciones disponibles. `Fraction` es un tipo de objeto (*clase*); sus operaciones están disponibles en `ipython` a través de `Fraction.<TAB>`. Las que empiezan con __ son internos y deberían de ignorarse por el momento; las demás son accesibles al usuario.

## 2.8. Programitas de Python: scripts

Una vez que las secuencias de comandos se complican, es útil poder guardarlos en un programa, o *script*. Eso consiste en un archivo de texto, cuyo nombre termina en `.py`, donde se guarda la secuencia de comandos.

Por ejemplo, guardemos el código anterior en el archivo `cuad.py` Podemos correr el código desde `ipython` con

```
run cuad
```

Al terminar de correr el programa, el control regresa a `ipython`, pero todavía tenemos acceso a las variables y funciones definidas en el script.

Alternativamente, podemos tratar el script como un programa en sí al ejecutar desde el shell

```
python cuad.py
```

Al terminar, el control regresa al shell, y perdemos la información adentro del script.

De hecho, un script se puede considerar un módulo en sí, que también se puede importar:

```
from cuad import *
```

# Capítulo 3

# Estructuras de control

Por estructuras de control, se entiende los comandos tales como condicionales, bucles, y funciones, que cambian el orden de ejecución de un programa.

Las estructuras de control tienen un formato en común. A diferencia de C++ y Fortran, no existen instrucciones que indiquen donde empieza y termina un bloque del programa: en Python, un bloque empieza por `:`, y su extensión se indica por el uso de una cantidad definida de *espacio en blanco* al principio de cada línea. Un buen editor de texto te permite hacer eso de manera automática.

## 3.1. Condicionales

Para checar una condición y ejecutar código dependiente del resultado, se ocupa **`if`**:

```
a = raw_input('Dame el valor de a')
if a > 0:
  print "a es positiva"
  a = a + 1
elif a == 0:            # NB: dos veces =
  print "a es 0"
else:
  print "a es negativa"
```

En el primer caso, hay dos comandos en el bloque que se ejecutará en el caso de que $a > 0$. Nótese que **`elif`** es una abreviación de **`else:if`**

Las condiciones que se pueden ocupar incluyen: <, <= y `!=` (no es igual). Para combinar condiciones se ocupan las palabras **`and`** y **`or`**:

```
a = 3; b=7
if a>1 and b>2:
  print "Grandes"
if a>0 or b>0:
  print "Al menos uno positivo"
```

## 3.2. Bucles

La parte central de muchos cálculos científicos consiste en llevar a cabo bucles, aunque en Python, como veremos, eso se desenfatiza.

### 3.2.1. `while`

El tipo de bucle más sencillo es un **`while`**, que ejecuta un bloque de código *mientras* una cierta condición se satisfaga, y suele ocuparse para llevar a cabo una iteración en la cual no se conoce de antemano el número de iteraciones necesario.

El ejemplo más sencillo de un **`while`**, donde sí se conoce la condición terminal, para contar hasta 9, se puede hacer como sigue:

```
i = 0
while i < 10:
  i += 1          # equivalente a i = i + 1
  print i
```

Nótese que si la condición deseada es del tipo "hasta…", entonces se tiene que utilizar un **`while`** con la condición opuesta.

**Ejercicio:** Encuentra los números de Fibonacci menores que 1000.

**Ejercicio:** Implementa el llamado método babilónico para calcular la raíz cuadrada de un número dado $y$. Este método consiste en la iteración $x_{n+1} = \frac{1}{2}\left[x_n + (y/x_n)\right]$. ¿Qué tan rápido converge?

### 3.2.2. `for`

Un bucle de tipo **`for`** se suele ocupar para llevar a cabo un número de iteraciones que se conoce de antemano. En el bucle de tipo **`for`**, encontramos una diferencia importante con respecto a lenguajes más tradicionales: podemos iterar sobre *cualquier* lista y ejecutar un bloque de código *para* cada elemento de la lista:

```
l = [1, 2.5, -3.71, "hola", [2, 3]]
for i in l:
  print 2*l
```

Si queremos iterar sobre muchos elementos, es más útil construir la lista. Por ejemplo, para hacer una iteración para todos los números hasta 100, podemos utilizar

```
for i in range(100):
  print 2*i
```

¿Qué es lo que hace la función **`range`**? Para averiguarlo, lo investigamos de manera interactiva con `ipython`:

```
range(10)
range(3, 10, 2)
```

Nótese que **`range`** no acepta argumentos de punto flotante.

**Ejercicio:** Toma una lista, y crea una nueva que contiene la duplicación de cada elemento de la lista anterior.

## 3.3. Funciones

Las funciones se pueden considerar como subprogramas que ejecutan una tarea dada. Corresponden a las subrutinas en Fortran. En Python, las funciones pueden o no aceptar argumentos, y pueden o no regresar resultados.

La sintaxis para declarar una función es como sigue:

```
def f(x):
  print "Argumento x = ", x
  return x*x
```

y se llama así:

```
f(3)
f(3.5)
```

Las funciones se pueden utilizar con argumentos de *cualquier* tipo –el tipo de los argumentos nunca se especifica. Si las operaciones llevadas a cabo no se permiten para el tipo que se provee, entonces Python regresa un error:

```
f("hola")
```

Las funciones pueden regresar varios resultados al juntarlos en un tuple:

```
def f(x, y):
  return 2*x, 3*y
```

Se puede proporcionar una forma sencilla de documentación de una función al proporcionar un "docstring":

```
def cuad(x):
  """Funcion para llevar un numero al cuadrado.
  Funciona siempre y cuando el tipo de x permite multiplicacion.
  """
  return x*x
```

Ahora si interrogamos al objeto cuad con `ipython`:

```
cuad?
```

nos da esta información. Si la función está definida en un archivo, entonces `cuad??` muestra el código de la definición de la función.

**Ejercicio:** Pon el código para calcular la raíz cuadrada de un número adentro de una función. Calcula la raíz de los números de 0 hasta 10 en pasos de 0.2 e imprimir los resultados.

## 3.4. Bibliotecas matemáticas

Para llevar a cabo cálculos con funciones más avanzadas, es necesario *importar* la biblioteca estándar de matemáticas:

```
from math import *
```

Eso no es necesario al correr `ipython -pylab`, que invoca un modo específico de `ipython`[1]. Sin embargo, eso importa mucho más que la simple biblioteca de matemáticas. Nótese que Python está escrito en C, así que la biblioteca `math` provee acceso a las funciones matemáticas que están en la biblioteca estándar de C. Más adelante veremos como acceder a otras funciones, tales como funciones especiales.

**Ejercicio:** ¿Cuál es la respuesta de la entrada `sqrt(-1)`?

**Ejercicio:** Intenta resolver la ecuación cuadrática $ax^2 + bx + c = 0$ para distintos valores de $a$, $b$ y $c$.

Para permitir operaciones con números complejos como posibles respuestas, se ocupa la biblioteca (*módulo*) `cmath`. Si hacemos

```
from cmath import *
```

entonces se importan las funciones adecuadas:

```
sqrt(-1)
```

[1] Yo encuentro conveniente declarar un nuevo comando `pylab` agregando la línea `alias pylab='ipython -pylab'` en el archivo `.bashrc` en mi directorio HOME.

Sin embargo, ya se "perdieron" las funciones anteriores, es decir, las nuevas funciones han reemplazado a las distintas funciones con los mismos nombres que había antes.

Una solución es importar la biblioteca de otra manera:

```
import cmath
```

Ahora las funciones que se han importado de `cmath` tienen nombres que empiezan por `cmath.`:

```
cmath.sqrt(-1)
```

`ipython` ahora nos proporciona la lista de funciones adentro del módulo si hacemos `cmath.<TAB>`.

# Capítulo 4

# Animaciones sencillas con Visual Python

En este capítulo, veremos un paquete, Visual Python, que permite hacer animaciones en 3 dimensiones, en tiempo real, de una manera sencillísima. ¡Casi vale la pena aprender Python solamente para eso! Y evidentemente es excelente para enseñar conceptos básicos de la física. De hecho, fue diseñado principalmente para este fin.

## 4.1. El paquete Visual Python

La biblioteca de Visual Python (de aquí en adelante, "Visual") se carga con

```
from visual import *
```

La manera más fácil de entender cómo utilizarlo es por ejemplo.

Empecemos creando una esfera:

```
s = sphere()
```

Podemos ver `sphere()` como una función que llamamos para crear un objeto tipo esfera. Para poder manipular este objeto después, se lo asignamos el nombre `s`.

Al crear objetos en Visual Python, *se despliegan por automático*, y ¡en 3 dimensiones! Como se puede imaginar, hay una cantidad feroz de trabajo abajo que permite que funcione eso, pero afortunadamente no tenemos que preocuparnos por eso, y simplemente podemos aprovechar su existencia.

¿Qué podemos hacer con la esfera `s`? Como siempre, `ipython` nos permite averiguarlo al poner `s.<TAB>`. Básicamente, podemos cambiar sus *propiedades internas*, tales como su color, su radio y su posición:

```
s.color = color.red      # o  s.color = 1, 0, 0
s.radius = 0.5
s.pos = 1, 0, 0
```

Aquí, `s.pos` es un vector, también definido por Visual, como podemos ver al teclear `type(s.pos)`. Los vectores en Visual son diferentes de los que provee `numpy`. Los de Visual siempre tienen 3 componentes.

Ahora podemos construir otros objetos, incluyendo a `box`, `cylinder`, etc. Nótese que la gráfica se puede rotar en 3 dimensiones con el ratón, al arrastar con el botón de derecho puesto, y se puede hacer un acercamiento con el botón central.

## 4.2. Animaciones

Ahora llega lo bueno. ¿Cómo podemos hacer una animación? Una animación no es más que una secuencia de imágenes, desplegadas rápidamente una tras otra. Así que eso es lo que tenemos que hacer:

```
s = sphere()
b = box()
for i in range(10000):
  rate(100)
  s.pos = i/1000., 0, 0
```

## 4.3. Agregando propiedades a objetos

Supongamos que queremos pensar en nuestra esfera como una pelota. Entonces la pelota tendrá no solamente una posición, sino también una velocidad. Se lo podemos crear así:

```
pelota = sphere()
pelota.vel = vector(1,0,0)
```

Nótese que se tiene que poner explícitamente `vector`, ya que sino sería una $n$-ada (tupla). Ahora quedó definida la velocidad de la pelota como otra propiedad interna.

Eso es básicamente todo lo que hay que saber de Visual Python. También es posible interactuar con el teclado, extraer las coordenadas del ratón, etc.

# Capítulo 5

# Cómo leer y escribir archivos

Uno de los usos principales de Python para el cómputo científico es el de procesar datos generados en otro lado. Por lo tanto, es necesario saber cómo importar y exportar archivos.

## 5.1. Redirección de la salida

En Linux, la manera más fácil (pero no muy flexible) de guardar datos en un archivo es utilizando la llamada "redirección" que provee el shell (Bash). Al correr un programa así desde Bash:

```
./programa > salida.dat
```

la salida estándar (lo que aparece en la pantalla al correr el programa) se manda al archivo especificado. Así que

```
python prog.py > salida.dat
```

mandará la salida del programa de python `prog.py` al archivo `salida.dat`.

**Ejercicio:** Guarda los resultados de las raices cuadradas de diferentes números, y grafícalos con `gnuplot`: `plot "salida.dat"`

**Ejercicio:** Haz una gráfica de la velocidad de convergencia del método Babilónico para calcular la raíz cuadrada.

## 5.2. Leer archivos

Para leer un archivo, primero es necesario abrirlo para leerse. Supongamos que tenemos un archivo llamado `datos.dat`, entonces lo podemos abrir para su lectura con

```
entrada = open("datos.dat", "r")
```

El segundo argumento, `"r"`, es para indicar que se va a leer ("read") el archivo. El objeto `entrada` ahora representa el archivo.

Para leer del archivo abierto, hay varias posibilidades. Podemos leer todo de un golpe con `entrada.read()`, leer todo por líneas con `entrada.readlines()`, o línea por línea con `entrada.readline()`. [Nótese que lo que se ha leído ya no se puede leer de nuevo sin cerrar el archivo con `entrada.close()` y volverlo a abrir.]

Por ejemplo, podemos utilizar

```
for linea in entrada.readlines():
  print linea
```

Sin embargo, tal vez la opción más fácil e intuitiva es

```
for linea in entrada:
  print linea
```

Es decir, el archivo ¡se comporta como una secuencia!

Ahora, la línea viene como una sola cadena, con espacios etc. Para extraer la información, primero necesitamos dividirlo en palabras:

```
palabras = linea.split()
```

Si todos los datos en realidad son números, entonces tenemos que procesar cada palabra, convirtiéndola en un número:

```
datos = []
for i in palabras:
  datos.append( float(i) )
```

Resulta que hay una manera más fácil, de más alto nivel, y (!) más rápida de hacer eso:

```
map(float, palabras)
```

Eso literalmente mapea la función `float` sobre la lista `palabras`.

Combinando todo, podemos escribir

```
entrada = open("datos.dat", "r")

for linea in entrada:
  datos = map( float, linea.split() )
```

Ahora adentro del bucle podemos manipular los datos como queramos.

**Ejercicio:** Para un archivo con dos columnas, leer los datos de cada columna en dos listas por separado.

## 5.3. Escribir archivos

El método de redirigir la salida que vimos arriba es rápido y fácil. Sin embargo, no es de ninguna manera flexible, por ejemplo no podemos especificar desde nuestro programa el nombre del archivo de salida, ni escribir en dos archivos diferentes.

Para hacer eso, necesitamos poder abrir un archivo para escribirse:

```
salida = open("resultados.dat", "w")
```

La parte más complicada viene al momento de escribir en el archivo. Para hacerlo, ocupamos la función `salida.write()`. Sin embargo, esta función puede escribir solamente *cadenas*. Por lo tanto, si queremos escribir números contenidos en variables, es necesario convertirlos primero a la forma de cadena.

Una manera de hacer eso es con la función `str()`, y concatenar distintas cadenas con `+`:

```
a = 3
s = "El valor de a es " + str(a)
```

Sin embargo, para largas secuencias de datos eso se vuelve latoso.

## 5.4. Sustitución de variables en cadenas

La manera más elegante de hacerlo es con la sustitución de variables en cadenas[1]. En una cadena ponemos una secuencia especial de caracteres, empezando por `%`, que indica que se sustituirá el valor de una variable:

[1] Eso es muy parecido a `printf` en C.

```
a = 3
s = "El valor de a es %d" % a
b = 3.5; c = 10.1
s = "b = %g;  c = %g" % (b, c)
```

El caracter despues del `%` indica el tipo de variable que incluir en la cadena: `d` corresponde a un entero, `g` o `f` a un flotante, y `s` a otra cadena. También se puede poner un número entero, que es el tamaño del campo, y un punto decimal seguido por un número, que da el número de decimales para el caso de `f`.

Finalmente ahora podemos imprimir en el archivo, por ejemplo

```
salida.write("%g\t%g" % (a,b) )
```

Aquí, `\t` representa un tabulador, y `\n` una nueva línea.

# Capítulo 6

# Operaciones matemáticas y la biblioteca de arreglos numpy

## 6.1. Conversión de números

Acordémonos que hay distintos tipos de números en Python, principalmente enteros (`int`) y flotantes (`float`). Para convertir entre diferentes tipos, incluyendo cadenas, podemos utilizar

```
a = float(3)
b = float('3.5')
```

## 6.2. Aritmética con precisión arbitraria

A veces, es necesaria poder llevar a cabo operaciones aritméticas con números flotantes (reales) con precisión superior a los 16 dígitos que provee el `float` (número de "doble precisión") de Python. Para hacerlo, existen varios proyectos que proveen bibliotecas con este fin. La mayoría de estas bibliotecas son interfaces a otros proyectos escritos en C++.

Aquí veremos una opción, la biblioteca `mpmath`, que está escrito completamente en Python. En principio eso lo hace más lento, pero más fácil de entender y modificar el código.

Para cargar la biblioteca, hacemos

```
from mpmath import *
```

Para cambiar la precisión, hacemos

```
mp.dps = 50
```

Ahora, para crear un número flotante de esta precisión, hacemos

```
x = mpf('1.0')
```

donde el número se expresa como cadena.

Al hacer manipulaciones con `x`, los cálculos se llevan a cabo en precisión múltiple. Por ejemplo,

```
print x/6., x*10
print mpf('2.0')**2**2**2**2
```

Con `mpmath`, no hay límite del exponente que se puede manejar. También están definidas muchas funciones, por ejemplo `sin`, `exp` y `log`.

Para imprimir un número con una precisión dada, usamos

```
nprint(x, 20)
```

## 6.3. La biblioteca numpy

Hasta ahora, hemos utilizado listas para guardar y manipular datos. Sin embargo, las listas no se comportan como vectores, y menos como matrices –al sumarlos no se comportan de la manera adecuada, etc. El propósito de la biblioteca `numpy` es justamente el de proporcionar objetos que represantan a vectores y matrices matemáticos, con todas las bondades que traen consigo este tipo de objetos.

La biblioteca se carga con

```
from numpy import *
```

Provee muchas funciones para trabajar con vectores y matrices.

## 6.4. Creando vectores

Los vectores se llaman (aunque es un poco confuso) `array`s, y se pueden crear de distintas maneras. Son como listas, pero con propiedades y métodos adicionales para funcionar como objetos matemáticos. La manera más general de crear un vector es justamente convirtiendo desde una lista de números:

```
from numpy import *

a = array( [1, 2, -1, 100] )
```

Un vector de este tipo puede contener sólo un tipo de objetos, a diferencia de una lista normal de Python. Si todos los números en la lista son enteros, entonces el tipo del arreglo también lo es, como se puede comprobar con `a.dtype`.

Es común querer crear vectores de cierto tamaño con todos ceros:

```
b = zeros( 10 )
print b
```

o todos unos:

```
b = ones( 10 )
print b
```

También hay distintas maneras de crear vectores que consisten en rangos ordenados, por ejemplo `arange`, que funciona como **`range`**, con un punto inicial, un punto final, y un paso:

```
a = arange(0., 10., 0.1)
```

y `linspace`, donde se especifica puntos iniciales y finales y un número de entradas:

```
l = linspace(0., 10., 11)
```

Una notación abreviada para construir vectores es `r_`, que se puede pensar como una abreviación de “vector *r*englón”:

```
a = r_[1,2,10,-1.]
```

Este método se extiende para dar una manera rápida de construir rangos:

```
r_[3:7]
r_[3:7:0.5]
r_[3:7:10j]
```

Este último utiliza un “número complejo” simplemente como otra notación, y es el equivalente de `linspace(3,7,10)`.

## 6.5. Operando con vectores

Los vectores creados de esta manera se pueden sumar, restar etc., como si fueran vectores matemáticos. Todas las operaciones se llevan a cabo entrada por entrada:

```
a = array( [1., 4., 7. ])
b = array( [1., 2., -2. ])
print a+b, a-b, a*b, a/b, a**b
```

**Ejercicio:** ¿Cómo se puede crear un vector de 100 veces $-3$?

Las funciones más comunes entre vectores ya están definidas en `numpy`, entre las cuales se encuentran `dot(a,b)` para productos escalares de dos vectores de la mismo longitud, y `cross(a,b)` para el producto cruz de dos vectores de longitud 3.

Además, cualquier función matemática como `sin` y `exp` se puede aplicar directamente a un vector, y regresará un vector compuesto por esta función aplicada a cada entrada del vector, tipo **`map`**.

Es más: al definir una función el usuario, esta función normalmente también se pueden aplicar directamente a un vector:

```
def gauss(x):
  return 1./ (sqrt(2.)) * exp(-x*x / 2.)

gauss( r_[0:10] )
```

## 6.6. Extrayendo partes de un vector

Para extraer subpartes de un vector, la misma sintaxis funciona como para listas: se extraen componentes (entradas) individuales con

```
a = array([0, 1, 2, 3])
print a[0], a[2]
```

y subvectores con

```
b = a[1:3]
```

Nótese, sin embargo, que en este caso la variable `b` *no* es una *copia* de esta parte de `a`. Más bien, es una *vista* de `a`, así que ahora si hacemos

```
b[1] = 10
```

entonces la entrada correspondiente de `a` *¡también cambia!* Este fenómeno también funciona con listas:

```
l=[1,2,3]; k=l; k[1] = 10
```

En general, en Python las variables son *nombres* de objetos; al poner `b = a`, tenemos ¡un mismo objeto con dos nombres!

## 6.7. Matrices

Las matrices se tratan como vectores de vectores, o listas de listas:

```
M = array( [ [1, 2], [3, 4] ] )
```

La forma de la matriz se puede ver con

```
M.shape
```

y se puede manipular con

```
M.shape = (4, 1); print M
```

o con

```
M.reshape( 2, 2 )
```

De hecho, eso es una manera útil de crear las matrices:

```
M = r_[0:4].reshape(2,2)
```

Podemos crear una matriz desde una función:

```
def f(i, j):
  return i+j
M = fromfunction(f, (3, 3))
```

Dado que las matrices son vectores de vectores, al hacer

```
print M[0]
```

nos regresa la primera componente de `M`, que es justamente un *vector*, viz. el primer renglón de `M`. Si queremos cierta entrada de la matriz, entonces más bien necesitamos especificar dos coordenadas:

```
M[0][1]
M[0, 1]
```

[También podemos poner `M.item(1)` que aparentemente es maś eficiente.]

Para extraer ciertas renglones o columnas de `M`, utilizamos una extensión de la notación para vectores:

```
M = identity(10)                 # matriz identidad de 10x10
M[3:5]
M[:, 3:5]
M[3:9, 3:5]
```

Una función poderosa para construir matrices repetidas es `tile`

```
tile( M, (2,2) )
```

Otros métodos útiles son `diagonal`, que regresa una diagonal de un arreglo:

```
diagonal(M)
diagonal(M, 1)
```

y `diag`, que construye una matriz con el vector dado como diagonal:

```
diag([1,2,3])
diag([1,2,3], 2)
```

## 6.8. Álgebra lineal

Adentro de `numpy` hay un módulo de álgebra lineal que permite hacer las operaciones más frecuentes que involucran matrices y vectores.

Para empezar, todavía no adentro del módulo `numpy`, están los productos de una matriz `M` con un vector `v` y de dos matrices. Los dos se llevan a cabo con `dot`:

```
M = r_[0:4].reshape(2,2)
v = r_[3, 5]
dot(M, v)
dot(M, M)
```

**Ejercicio:** Utiliza el *método de potencias* para calcular el vector propio de una matriz cuadrada $M$ correspondiente al valor propio de mayor módulo. Este método consiste en considerar la iteración $M^n \cdot v$.

Para álgebra lineal, se ocupa el módulo `linalg`:

```
from numpy import linalg
```

Entonces las funciones del módulo se llaman e.g. `norm` para la norma de un vector se llama

```
linalg.norm(v)
```

Una alternativa es importar todas las funciones de este submódulo

```
from numpy.linalg import *
```

entonces ya no se pone `linalg.`, y se puede referir simplemente a `norm`.

Algunas de las funciones útiles incluyen `linalg.eig` para calcular los valores y vectores propios de una matrix:

```
linalg.eig(M)
```

y `linalg.eigvals` para solamente los valores propios. Hay versiones especiales `eigh` y `eigvalsh` para matrices hermitianas o simétricas reales.

También hay `det` para determinante, e `inv` para la inversa de una matriz. Finalmente, se puede resolver un sistema de ecuaciones lineales $M \cdot x = b$ con

```
linalg.solve(M, b)
```

## 6.9. Funciones para resumir información sobre vectores y matrices

Hay varias funciones que regresan información resumida sobre vectores y matrices, por ejemplo

```
a = r_[0:16].reshape(4,4)

a.max()
a.min(1)        # actua sobre solamente un eje y regresa un vector
a.mean(0)
a.mean(1)
a.sum(1)
```

También podemos seleccionar a los elementos de un arreglo que satisfacen cierta propiedad:

```
a > 5
a[a > 5] = 0
```

Además existe una función `where`, que es como una versión vectorizada de `if`:

```
b = r_[0:16].reshape(4,4)
c = list(b.flatten())
c.reverse()
c = array(c).reshape(4,4)

a = where(b < 5, b, c)
```

Este comando pone cada entrada de `a` igual a la entrada correspondiente de `b` si ésta es menor que 5, o a la de `c` si no.

## 6.10. Números aleatorios

La biblioteca `numpy` incluye un módulo amplio para manipular números aleatorios, llamado `random`. Como siempre, las funciones se llaman, por ejemplo, `random.random()`. Para facilitarnos la vida, podemos importar todas estas funciones al espacio de nombres con

```
from numpy import *
random? # informacion sobre el modulo
from random import *            # 'random' esta
```

o

```
from numpy.random import *
```

Nótese que hay otro módulo `random` que existe afuera de `numpy`, con distinta funcionalidad.

La funcionalidad básica del módulo es la de generar números aleatorios distribuidos de manera uniforme en el intervalo $[0,1)$:

```
random()
for i in xrange(10):
  random()
```

Al poner `random(N)`, nos regresa un vector de números aleatorios de longitud `N`.

Para generar una matriz aleatoria, podemos utilizar

```
rand(10, 20)
```

Para números en un cierto rango $[a,b)$, podemos utilizar

```
uniform(-5, 5, 10)
```

También hay diversas funciones para generar números aleatorios con distribuciones no-uniformes, por ejemplo `exponential(10)` y `randn(10,5,10)` para una distribución normal, o `binomial(10,3,100)`.

**Ejercicio:** Calcula la distribución de distancias entre valores propios consecutivos de una matriz aleatoria real y simétrica.

## 6.11. Transformadas de Fourier

Otro submódulo útil de `numpy` es `fft`, que provee transformadas rápidas de Fourier:

```
from numpy import *

x = arange(1024)
f = fft.fft(x)
y = fft.ifft(f)
linalg.norm( x - y )
```

# Capítulo 7

# Gráficas con `matplotlib`

En este capítulo, veremos cómo podemos crear gráficas dentro de Python, usando el paquete `matplotlib` y su entorno `pylab`.

## 7.1. Entorno `pylab`

El paquete `matplotlib` dibuja las gráficas. Para integrar mejor este módulo con el uso interactivo, y en particular con `ipython`, incluye un módulo `pylab`, que provee muchos comandos de utilidad para manejar `matplotlib`.

La manera más efectiva de cargar la biblioteca `pylab` es al mero momento de *correr* `ipython`: se da la opción `-pylab` en la línea de comandos[1]:

```
> ipython -pylab
```

Si todo funciona correctamente, deberías recibir el mensaje "Welcome to pylab, a matplotlib-based Python environment." En este caso, ya se habrá cargado el entorno `pylab`, que incluye el módulo `matplotlib` para llevar a cabo gráficas, y también carga automáticamente `numpy`, por lo cual no es necesario volver a cargar estos módulos para uso interactivo. [Sí es necesario cargarlos explícitamente en cualquier script que ocupe gráficas.]

## 7.2. Gráficas básicas

El comando principal de `matplotlib` / `pylab` es `plot`. Acepta uno o dos listas o vectores de `numpy`, que corresponden a las coordenadas $y$ (si hay un solo vector) o $x$ y $y$ de una gráfica 2D. Por ejemplo, si queremos graficar los cuadrados de los números de 1 a 10, podemos hacer

```
x = arange(10)
y = x * x
plot(x, y)
```

Si queremos puntos en lugar de líneas, hacemos

```
plot(x, y, 'o')
```

al estilo de MATLAB. (De hecho, `pylab` se creó basándose en el comportamiento de MATLAB.)

Nótese que por defecto las gráficas se *acumulan*. Este comportamiento se puede modficar con `hold(False)`, que reemplaza las gráficas con las nuevas, y `hold(True)`, que regresa a la funcionalidad por defecto. También se puede utilizar `clf()` para limpiar la figura.

[1] Yo encuentro conveniente declarar un nuevo comando `pylab` agregando la línea `alias pylab='ipython -pylab'` en el archivo `.bashrc` en mi directorio hogar.). Entonces se puede correr simplemente a poner `pylab` en la línea de comandos.

La ventana que cree matplotlib incluye botones para poder hacer acercamientos y moverse a través de la gráfica. También incluye un botón para exportar la figura a un archivo en distintos formatos. Los formatos de principal interés son PDF, que es un formato "vectorial" (que incluye las instrucciones para dibujar la figura), que da la calidad necesaria para las publicaciones, y PNG, que da una imagen de la figura, y es adecuada para páginas web.

## 7.3. Cambiando el formato

Hay distintos tipos de líneas y puntos disponibles, y se puede modificar el tamaño de ambos. Todas las opciones están disponible a través de la documentación de plot, a través de plot?. Además, los colores se pueden especificar explícitamente:

```
x = arange(10)
plot(x, x**2, 'ro', x, 2*x, 'gx')
plot(x, 3*x, 'bo-', linewidth=3, markersize=5, markerfacecolor='red', markeredgecolor='green', mar
```

Se pueden dar cualquier número de gráficas que dibujar en un solo comando. Si el formato no se da explícitamente, entonces matplotlib escoge el siguiente de una secuencia razonable de estilos.

## 7.4. Etiquetas

Se puede proporcionar un título de la gráfica con title

```
title("Algunas funciones sencillas")
```

Los ejes se pueden etiquetar con

```
xlabel("x")
ylabel("f(x)")
```

Una bondad de matplotlib es que las etiquetas se pueden expresar en formato LaTeX y se interpretará automáticamente de la forma adecuada:

```
xlabel("$x$")
ylabel("$x^2, 2x, \exp(x)$", fontsize=16)        # cambia tamano de fuente
```

También se pueden colocar etiquetas arbitrarias con

```
text(4.6, 35, "Punto de\ninteres")
```

Si se le asigna a una variable, entonces se puede volver a remover con

```
etiq = text(4.6, 35, "Punto de\ninteres")
etiq.remove()
draw()
```

Es necesario llamar a draw() para volver a dibujar la figura.

## 7.5. Figuras desde scripts

Una vez que una figura se vuelve complicada, es necesario recurrir a un script que contenga las instrucciones para dibujarla. Eso es *sumamente importante* en particular para preparar figuras para un documento, donde se ajustará varias veces una sola figura hasta que quede bien.

Para utilizar matplotlib desde un script, es necesario incluir la biblioteca pylab. Luego se puede utilizar plot. Para ver la imagen, a veces es necesario poner el comando show():

```
from pylab import *
x = arange(10)
plot(x, x**2)
show()
```

Eso también es necesario al utilizar `pylab` desde `ipython` sin poner `ipython-pylab`, pero en este caso, es necesario cerrar la gráfica para poder volver a utilizar el `ipython`.

## 7.6. Escalas logarítmicas

Para utilizar ejes con escalas logarítimicas, hay tres funciones: `loglog`, `semilogy` y `semilogx`. Se utilizan en lugar de `plot`:

```
t = arange(1000)
p = t**(-0.5)
loglog(t, p, 'o')
```

## 7.7. Múltiples dibujos

Está fácil hacer múltiples dibujos alineados con `subplot`. Su sintaxis es

```
subplot(num_renglones, num_columnas, num_dibujo)
```

Es decir, se especifican el número de renglones y columnas que uno quiere, y el último número especifica cuál dibujo es los `num_renglones` × `num_columnas` es. Aquí está un ejemplo adaptado de la documentación de `matplotlib`:

```
def f(t):
  """Oscilacion amortiguada"""
  c = cos(2*pi*t)
  e = exp(-t)
  return c*e

t1 = arange(0.0, 5.0, 0.1)
t2 = arange(0.0, 5.0, 0.02)
t3 = arange(0.0, 2.0, 0.01)

subplot(211)
l = plot(t1, f(t1), 'bo', t2, f(t2), 'k--', markerfacecolor='green')
grid(True)
title('Amortiguacion de oscilaciones')
ylabel('Amortiguada')

subplot(212)
plot(t3, cos(2*pi*t3), 'r.')
grid(True)
xlabel('tiempo $t$ (s)')
ylabel('No amortiguada')
show()
```

## 7.8. Animaciones

Las gráficas hechas en `matplotlib` se pueden animar. La manera más fácil es simplemente redibujar la gráfica completa cada vez que se cambian los datos. Sin embargo, eso no es eficiente, ya que se tiene que recalcular cada vez las etiquetas etc.

Una mejor solución es que cada vez se cambie simplemente el contenido de datos de la gráfica. Primero es necesario asignarle a la gráfica un nombre:

```
from pylab import *
x = arange(10)
y = x*x
ion()
p, = plot(x, y)

for a in arange(0, 10, 0.1):
  x = arange(10) + a
  p.set_xdata(x)
  draw()
```

Nótese el comando `ion()` ("interactividad prendida"). El comando `p, = plot(x,y)` es necesario ya que el comando `plot` regresa una lista de todos los objetos en el plot. Se utiliza esta asignación de tuplas para extraer realmente el primer elemento; sería equivalente (pero menos natural) poner `p = plot(x,y)[0]`.

## 7.9. Otros tipos de gráficas

Aparte del tipo básico de gráficas que hemos visto hasta ahora, que permiten llevar a cabo gráficas de datos como puntos y/o líneas, existe en `matplotlib` una gran variedad de otros tipos de gráficas posibles; se refiere a la página principal `http://matplotlib.sourceforge.net` y a la página `http://www.scipy.org/Cookbook/Matplotlib` que contiene muchos ejemplos.

Por ejemplo, podemos visualizar matrices 2D como "heat-maps" (mapas en los cuales el color corresponde al valor de la matriz) con `pcolor` y `imshow`. Nótese que para crear una figura cuadrada, se puede utilizar

```
f = figure( figsize=(8,8) )
```

Otro tipo de gráfica disponible es un histograma. `pylab` puede calcular y dibujar la gráfica en un solo comando:

```
from pylab import randn, hist
x = randn(10000)
hist(x, 100, normed=True)
```

Hay una función `histogram` en `numpy` que calcula histogramas sin dibujarlos.

Aquí va una versión tomada de la documentación:

```
import numpy as np
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt

mu, sigma = 100, 15
x = mu + sigma*np.random.randn(10000)

# the histogram of the data
n, bins, patches = plt.hist(x, 50, normed=1, facecolor='green', alpha=0.75)

# add a 'best fit' line
y = mlab.normpdf( bins, mu, sigma)
l = plt.plot(bins, y, 'r--', linewidth=1)

plt.xlabel('Smarts')
plt.ylabel('Probability')
plt.title(r'$\mathrm{Histogram\ of\ IQ:}\ \mu=100,\ \sigma=15$')
plt.axis([40, 160, 0, 0.03])
```

```
plt.grid(True)

plt.show()
```

## 7.10. Uso avanzado de matplotlib

Para usos más avanzados de matplotlib, es necesario entender mejor la estructura interna del paquete, que es justamente lo que esconde pylab. Por ejemplo, para colocar una flecha en un dibujo, utilizamos "patches" ("parches"):

```
from pylab import *

x = arange(10)
y = x

plot(x, y)
arr = Arrow(2, 2, 1, 1, edgecolor='white')
ax = gca()   # obtener el objeto tipo "eje" ("axis")
ax.add_patch(arr)
arr.set_facecolor('g')
show()
```

## 7.11. Mallas con meshgrid

Por lo tanto, es necesario primero construir la malla donde se calculará el campo vectorial. Esto se hace con meshgrid, que acepta dos vectores 1D que dan las coordenadas $x$ y $y$ de los puntos de la malla. meshgrid construye una malla cuyos vértices son los puntos en el producto cartesiano de los dos conjuntos de puntos. Regresa una tupla de dos matrices, que dan las coordenadas $x$ y $y$ respectivamente de los puntos de la malla:

```
x = r_[-5:5:11j]
y = r_[-5:5:11j]

X, Y = meshgrid(x, y)
print X, Y
```

Así que meshgrid en algún sentido provee un mapa de las entradas de la matriz a sus coordenadas en el espacio real 2D.

El punto de esta construcción es que ahora las funciones se pueden evaluar en *cada punto de la red al mismo tiempo*. Por ejemplo, para tener una matriz que representa la suma de las componentes $x$ y $y$ en cada punto de la malla, hacemos

```
Z = X + Y
```

Para tener la distancia de la origen en cada punto, hacemos

```
R = sqrt(X*X + Y*Y)
```

Una abreviación del comando meshgrid, usando una notación indicial parecida a la que ocupa r_, es

```
X, Y = mgrid[-5:5:11j, -5:5:11j]
```

## 7.12. Campos vectoriales

Los campos vectoriales se pueden dibujar con la función `quiver`[2]. Esta función acepta 4 matrices, $X$ y $Y$ que dan las coordenadas $X$ y $Y$ de los puntos de la malla donde se especifica el campo, y $U$ y $V$, que son las componentes de los vectores que dibujar en cada punto.

Por ejemplo, podemos dibujar un campo radial al poner

```
X, Y = mgrid[-5:5:11j, -5:5:11j]
quiver(X, Y, X, Y)
```

ya que en el punto $(x,y)$ de la malla ponemos el vector $(x,y)$. Lo podemos colorear según el valor de la distancia del origen con

```
X, Y = mgrid[-5:5:11j, -5:5:11j]
R =  sqrt(X*X + Y*Y)
quiver(X, Y, X, Y, R)
```

Algunos campos más interesantes son

```
quiver(X, Y, Y, X)
quiver(X, Y, Y, -X)
```

Resulta que se ven un poco "deformados", ya que cada flecha se dibuja empezando en el punto dado de la malla. Se ve más ordenado al colocar el punto medio de la flecha en el punto de la malla:

```
quiver(X, Y, Y, X, R,  pivot='middle')
```

## 7.13. Gráficas en 3D

En versiones recientes de `matplotlib`, hay soporte limitado para gráficas en 3D. El módulo relevante es `mplot3d`. Un ejempo tomado de la documentación:

```
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
import matplotlib.pyplot as plt
import numpy as np

fig = plt.figure()
ax = Axes3D(fig)
X = np.arange(-5, 5, 0.25)
Y = np.arange(-5, 5, 0.25)
X, Y = np.meshgrid(X, Y)
R = np.sqrt(X**2 + Y**2)
Z = np.sin(R)
ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap=cm.jet)

plt.show()
```

## 7.14. Interacción con el mouse y el teclado

Se puede interactuar con el mouse y con el teclado de una manera relativamente sencilla.

Para interactuar con el mouse, creamos una función que se llamará cuando `pylab` note que hay un evento que involucre el mouse. Esta función toma un solo argumento, que suele llamarse `event`. `pylab` manda a la

[2]Eso es una broma nerd: "quiver" quiere decir "carcaj" (un objeto donde se guardan las flechas).

función un objeto que representa el evento, incluyendo cuál botón se presionó y en que posición de la gráfica. La función se registra ante `pylab` con la función `connect`.

Lo mismo tiene para el teclado. Así que el código más sencillo es el siguiente. Nótese que es necesario que haya una gráfica pre-existente antes de poder interactuar con ella.

```
from pylab import *

def teclado(event):
  print "Se tecleo %s en la posicion (%g, %g)" % event.key, event.xdata, event.ydata

def mouse(event):
  if event.button != 1:
    return
  x,y = event.xdata, event.ydata
  print "Se oprimio el boton izquierdo en (%g, %g)" % (x, y)

x = arange(10)
y = x*x
p, = plot(x, y)

show()

connect('button_press_event', mouse)
connect('key_press_event', teclado)
```

# Capítulo 8

# Ejemplos de cálculos y visualizaciones

## 8.1. Integración de ecuaciones diferenciales ordinarias

Veamos unas rutinas y herramientas para integrar ecuaciones diferenciales. Ocuparemos las técnicas básicas que se ven en el curso de Física Computacional –los métodos de Euler y Runge–Kutta.

La ecuación que resolveremos es

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}). \tag{8.1}$$

Recordemos que eso es una notación para decir que

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t)). \tag{8.2}$$

Es decir, si estamos en el punto $\mathbf{x}(t)$ al tiempo $t$, entonces la derivada instantánea en este momento y este punto del espacio fase está dada por $\mathbf{f}(\mathbf{x}(t))$.

Para resolver esta ecuación en la computadora, es necesario discretizarla de una manera u otra. La manera más sencilla para hacerlo es discretizando el tiempo en pasos iguales de tamaño $\delta t$ y aproximando la derivada por una diferencia finita, o sea expandiendo en una serie de Taylor, lo cual da

$$\mathbf{x}(t+\delta t) \simeq \mathbf{x}(t) + \delta t \dot{\mathbf{x}}(t) + O((\delta t)^2) = \mathbf{x}(t) + h\mathbf{f}(\mathbf{x}(t)) + O((\delta t)^2), \tag{8.3}$$

donde $h = \delta t$ es el paso de tiempo. Eso nos da el *método de Euler*. Lo podemos implementar como una función que regresa el valor nuevo de la variable:

```
def euler(t, x, h, f):
  return x + h*f(x)
```

Nótese que la función `euler` toma *el nombre de la función `f` que integrar* como argumento.

Ahora para integrar la ecuación entre un tiempo initial $t_0$ y un tiempo final $t_f$, repetimos este paso muchas veces:

```
def integrar(t0, tf, h, x0, f):
  lista_t = []
  lista_x = []

  x = x0
  for t in arange(t0, tf+h/2., h):
    x = euler(t, x, h, f)

    lista_t.append(t)
    lista_x.append(x)

  return lista_t, lista_x
```

Para integrar la ecuación $\dot{x} = -x$ ponemos entonces

```
def f(x):
  return x

t, x = integrar(0, 10, 0.1, 10, f)
```

Nótese que el *mismo código* funciona para ecuaciones vectoriales $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x})$, siempre y cuando la función `f` acepte un vector y regresa un vector.

Si tenemos varios métodos, entonces cambiamos la definición de `integrar` a

```
def integrar(t0, tf, h, x0, f, metodo):
```

y la línea que hace el trabajo a

```
x = metodo(t, x, h, f)
```

Ahora para integrar con respecto a distintos métodos, podemos hacer

```
metodos = [euler, rk2, rk4]

for metodo in metodos:
  t, x = integrar(0, 10, 0.1, 10, f, metodo)
```

# Capítulo 9

# Comunicándose con otros programas

En este capítulo, veremos cómo un programa de Python puede interactuar con otros programas.

## 9.1. Redirección

En esta sección veremos algunas técnicas útiles del shell `bash` que no están restringidos a `python`.

Ya hemos visto que la manera más fácil de guardar información en un archivo es con la redirección de la salida estándar del programa: corriendo desde la terminal, ponemos

```
python prog.py > salida.dat
```

y así creamos un archivo `salida.dat`, donde se capta la información que anteriormente se mandaba a la terminal.

Lo mismo se puede hacer con la entrada estándar: si tenemos un programa que lee información que teclea el usuario:

```
a = raw_input("Dame a: ")
b = raw_input("Dame b: ")
```

Entonces podemos también mandarle valores de un archivo, al redirigir la entrada estándar:

```
python prog.py < datos.dat
```

donde `datos.dat` contiene la información deseada. A su vez, la salida de este programa se puede mandar a otro archivo con `>`.

Finalmente, si queremos conectar la salida de un programa con la entrada de otro, utilizamos un "tubo" ("pipe"):

```
python prog1.py | python prog2.py
```

Eso es equivalente a hacer

```
python prog1.py > temp.dat
python prog2.py < temp.dat
```

pero sin la necesidad de crear el archivo temporal.

Los pipes son muy útiles en unix, y se pueden encadenar.

## 9.2. Argumentos de la línea de comandos

Una manera más flexible y más útil para mandar información a un programa es a través de argumentos que se ponen en la mera línea de comandos. A menudo queremos mandar a un programa los valores de un parámetro, para después poder hacer barridos del mismo.

Para hacerlo, Python provee una variable llamada `argv`, que viene definida en el módulo `sys`. Podemos ver qué es lo que contiene esta variable al correr un programa sencillo:

```
from sys import argv

print "Argumentos: argv"
```

Si ahora llamamos a nuestro programa con

```
python prog.py 10 0.5 ising
```

entonces vemos que `argv` es una lista de cadenas, cada una representando uno de los argumentos, empezando por el nombre del script. Por lo tanto, podemos extraer la información deseada con las herramientas normales de Python, por ejemplo

```
from sys import argv
T, h = map(float, argv[1:3])
modelo = argv[3]
```

Sin embargo, si no damos suficientes argumentos, entonces el programa fracasará. Podemos atrapar una *excepción* de este tipo con un bloque `try`...`except`:

```
from sys import argv, exit
try:
  T, h = map(float, argv[1:3])
  modelo = argv[3]
except:
  print "Sintaxis: python prog.py T h modelo"
  exit(1)
```

Es útil proveerle al usuario información acerca de la razón por la cual fracasó el programa. Aquí hemos utilizado la función `exit` que viene también en el módulo `sys`, para que salga del programa al encontrar un problema, pero eso no es obligatorio –podría poner valores por defecto de los parámetros en este caso, por ejemplo.

## 9.3. Llamando a otros programas

El módulo `os` provee unas herramientas para mandar llamar a otros programas "hijos" desde un programa "padre"[1].

La función más útil es `system`, que permite correr un comando a través de un nuevo `bash`; podemos enviar cualquier comando que funciona en `bash`, en particular podemos correr otros programas. La función acepta una cadena:

```
from os import system
system("ls")
```

Ahora podemos empezar a hacer cosas interesantes al construir los comandos con la sustitución de variables. Por ejemplo, si tenemos un programa `prog.py` que acepta un argumento de la línea de comandos, podemos correrlo de manera consecutiva con distintos valores del parámetro con

[1] Hoy en día, se recomienda más bien el paquete `subprocess` para esta funcionalidad, pero es más complicado.

```
for T in range(10):
  comando = "python prog.py %g" % T
  system(comando)
```

Si queremos abrir un programa que se conectará al nuestro y recibirá comandos, utilizamos `popen` para abrir un *pipe*:

```
from os import popen
gp = popen("gnuplot -persist", "w")
gp.write("plot sin(x)\n")
gp.close()
```

## 9.4. Ejemplos

- Cartas de aceptación en LaTeX
- GIF animado con gnuplot
- Figuras para distintos parámetros

# Capítulo 10

# Programación orientada a objetos: clases

En este capítulo, veremos uno de los temas más revolucionarios en materia de programación: la *programación orientada a objetos*. Aunque hemos estado utilizando los objetos de manera implícita a través del curso, ahora veremos cómo nosotros podemos definir nuestros propios tipos de objetos nuevos, y para qué sirve.

## 10.1. La programación orientada a objetos: las clases

Consideremos una situación clásica en el cómputo científico: un sistema que consiste en cierto número de partículas en 2 dimensiones, que tienen *propiedades internas*, como una posición, una velocidad y una masa, y que se pueden mover con una regla tipo Euler.

Para una partícula, podríamos definir sus variables simplemente como sigue:

```
x = y = 0.0     # posicion
vx =  vy = 1.0  # velocidad
```

Un paso de Euler de tamaño `dt` se puede implementar con

```
dt = 0.1
x += dt * vx;
y += dt * vy;
```

Pero ahora, ¿qué hacemos si necesitamos otra partícula más? Podríamos poner

```
x1 = y1 = 0.0
x2 = y2 = 0.0
vx1 = vy1 = 1.0
vx1 = vy1 = 1.0
```

Si las partículas también tienen masas y colores, entonces se vuelve muy tedioso, ya que tenemos que actualizar todo dos veces para cada propiedad nueva:

```
m1 = m2 = 0.0;
c1 = c2 = 0.0;
```

Para muchas particulas podríamos emplear arreglos (listas, en Python), que reduciría el trabajo.

Pero ahora podríamos imaginarnos que por alguna razón queremos agregar otra partícula, o incluso duplicar toda la simulación. Entonces tendríamos que duplicar a mano todas las variables, cambiando a la vez sus nombres, lo cual seguramente conducirá a introducir errores.

Sin embargo, *conceptualmente* tenemos muchas variables que están relacionadas: todas *pertenecen* a una partícula. Hasta ahora, no hay manera de expresar esto en el programa.

## 10.2. La solución: objetos, a través de clases

Lo que queremos hacer, entonces, es reunir todo lo que corresponde a una partícula en un *nuevo tipo de objeto*, llamado `Particula`. Luego podremos decir que `p1` y `p2` son `Particula`s, o incluso hacer un arreglo (lista) de `Particula`s.

Toda la información que le corresponde a una partícula dada estará contenida adentro del objeto; esta información formará parte del objeto no sólo conceptualmente, sino también en la representación en el programa.

Podemos pensar en un objeto, entonces, como un tipo de "caja negra", que tiene propiedades internas, y que puede interactuar de alguna manera con el mundo externo. No es necesario saber o entender qué es lo que hay adentro del objeto para entender cómo funciona. Se puede pensar que corresponde a una caja con palancas, botones y luces: las palancas y los botones proveen una manera de darle información o instrucciones a la caja, y las luces dan información de regreso al mundo externo.

Para implementar eso en Python, se declara una *clase* llamada `Particula` y se crea una *instancia* de esta clase, llamada `p`.

```
class Particula:
  x = 0.0

p = Particula()
p.x
```

Nótese que `p` tiene *adentro* una propiedad, que se llama `x`. Podemos interpretar `p.x` como una `x` que le *pertenece* a `p`.

Si ahora hacemos

```
p2 = Particula()
p2.x
```

entonces tenemos una variable completamente *distinta* que también se llama `x`, pero que ahora le pertenece a `p2`, que es otro objeto de tipo `Particula`. De hecho, más bien podemos pensar que `p2.x` es una manera de escribir `p2_x`, que es como lo hubiéramos podido escribir antes, que tiene la bondad de que ahora también tiene sentido pensar en el conjunto `p` como un todo.

Remarquemos que ya estamos acostumbrados a pensar en cajas de este tipo en matemáticas, al tratar con vectores, matrices, funciones, etc.

## 10.3. Métodos de clases

Hasta ahora, una clase actúa simplemente como una caja que contiene datos. Pero objetos no sólo tienen información, sino también pueden *hacer cosas*. Para hacerlo, también se pueden definir *funciones* –llamadas *métodos*– que le pertenecen al objeto. Simplemente se definen las funciones adentro de la declaración de la clase:

```
class Particula:
  x = 0.0
  v = 1.0

  def mover(self, dt):
    self.x += self.v*dt

p = Particula()
print p.x
p.mover(0.1)
print p.x
```

Nótese que los métodos de las clases siempre llevan un argumento extra, llamado `self`, que quiere decir "sí mismo". Las variables que pertenecen a la clase, y que se utilizan adentro de estas funciones, llevan `self.`, para indicar que son variables que forman parte de la clase, y no variables globales. Podemos pensar en las variables internas a la clase como variables "pseudo-globales", ya que están accesibles desde cualquier lugar *adentro* de la clase.

## 10.4. Funciones inicializadoras

Cuando creamos una instancia de un objeto, muchas veces queremos *inicializar* el objeto al mismo tiempo, es decir pasarle información sobre su estado inicial. En Python, esto se hace a través de una función inicializadora, como sigue:

```
class Particula:
  def __init__(self, xx=0.0, vv=1.0):
    self.x = xx
    self.v = vv

  def mover(self, dt):
    self.x += self.v*dt

p1 = Particula()
print p1.x
p2 = Particula(2.5, 3.7)
print p2.x
p1.mover(0.1)
p2.mover(0.1)
print p1.x, p2.x
```

Nótese que la función inicializadora debe llamarse `__init__`, con dos guiones bajos de cada lado del nombre `init`. Puede tomar argumentos que se utilizan para inicializar las variables de la clase.

## 10.5. Métodos internos de las clases

Las clases pueden ocupar varios métodos para proveer un interfaz más limpio para el usuario. Por ejemplo, al poner `print p1` en el último ejemplo, sale algo así como

```
<__main__.Particula instance at 0x2849cb0>
```

Podemos hacer que salga algo más útil al proveer adentro de la clase una función `__str__`:

```
class Particula:
  def __init__(self, xx=0.0, vv=1.0):
    self.x = xx
    self.v = vv

  def __str__(self):
    return "Particula(%g, %g)" % (self.x, self.v)

p1 = Particula()
print p1
```

También podemos definir funciones que permiten que podamos llevar a cabo operaciones aritméticas etc. con objetos de este tipo, es decir, podemos *sobrecargar* los operadores para que funcionen con nuestro nuevo tipo. Eso es lo que pasa con los `array` de `numpy`, por ejemplo.

**Ejercicio:** Haz un objeto para representar a una partícula en 2D. Haz una nube de tales partículas.

## 10.6. Herencia

Las clases también pueden formar jerarquías al utilizar la *herencia*, que quiere decir que una clase es un "tipo de", o "subtipo de" otro.

# Capítulo 11

# Paquetes para el cómputo científico en Python

Este capítulo propone dar un breve resumen de algunos de los muchos paquetes disponibles para llevar a cabo distintas tareas de cómputo científico en Python.

## 11.1. Cálculos numéricos con `scipy`

El paquete `scipy` provee una colección de herramientas para llevar a cabo tareas numéricas, como son los siguientes submódulos: funciones especiales (`special`); integración de funciones y de ecuaciones diferenciales ordinarias (`integrate`), optimización y raíces de funciones (`optimize`), álgebra lineal (`linalg`), incluyendo para matrices escasas (`sparse`), y estadísticas (`stats`).

Veamos algunos ejemplos. Para utilizar las funciones especiales, hacemos

```
from scipy import special
for i in range(5):


special.gamma(3)
special.gamma(3.5)

x = arange(-2, 2, 0.05)

for i in range(5):
  plot(x, map(special.hermite(i), x))
```

Para integrar la ecuación de Airy (un ejemplo de la documentación de `scipy`)

$$\frac{d^2w}{dz^2} - zw(z) = 0, \tag{11.1}$$

podemos poner

```
from scipy.integrate import odeint
from scipy.special import gamma, airy
y1_0 = 1.0/3**(2.0/3.0)/gamma(2.0/3.0)
y0_0 = -1.0/3**(1.0/3.0)/gamma(1.0/3.0)
y0 = [y0_0, y1_0]

def func(y, t):
  return [t*y[1],y[0]]

def gradiente(y,t):
```

```
  return [[0,t],[1,0]]

x = arange(0,4.0, 0.01)
t = x
ychk = airy(x)[0]
y = odeint(func, y0, t)
y2 = odeint(func, y0, t, Dfun=gradient)

print ychk[:36:6]
print y[:36:6,1]
print y2[:36:6,1]
```

En la segunda llamada a `odeint`, mandamos explícitamente la función que calcula la derivada (gradiente) de `f`; en la primera llamada, no fue necesario contar con una función que calculara eso.

Encontrar raíces de funciones es más o menos sencillo:

```
def f(x):
  return x + 2*cos(x)

def g(x):
  out = [x[0]*cos(x[1]) - 4]
  out.append(x[1]*x[0] - x[1] - 5)
  return out

from scipy.optimize import fsolve
x0 = fsolve(func, 0.3)
x02 = fsolve(func2, [1, 1])
```

## 11.2. Cálculos simbólicos con `sympy`

El módulo `sympy` provee una colección de rutinas para hacer cálculos simbólicos. Las variables se tienen que declarar como tal, y luego se pueden utilizar en expresiones simbólicas:

```
from sympy import *
x, y, z = symbols('xyz')
k, m, n = symbols('kmn', integer=True)
f = Function("f")

x + x
(x + y) ** 2
z = _
z.expand()

z.subs(x, 1)

limit(sin(x) / x, x, 0)
limit(1 - 1/x, x, oo)

diff(sin(2*x), x)
diff(sin(2*x), x, 3)

cos(x).series(x, 0, 10)

integrate(log(x), x)
integrate(sin(x), (x, 0, pi/2))
```

```
f(x).diff(x, x) + f(x)
dsolve(f(x).diff(x, x) + f(x), f(x))

solve(x**4 - 1, x)

pi.evalf(50)
```

## 11.3. mayavi2

El paquete `mayavi2` provee una manera de visualizar conjuntos de datos complejos en 3D. Para utilizarlo de manera interactiva, comenzamos con

```
ipython -wthread

import numpy as np

def V(x, y, z):
    """ A 3D sinusoidal lattice with a parabolic confinement. """
    return np.cos(10*x) + np.cos(10*y) + np.cos(10*z) + 2*(x**2 + y**2 + z**2)

X, Y, Z = np.mgrid[-2:2:100j, -2:2:100j, -2:2:100j]
V(X, Y, Z)

from numpy import *
from enthought.mayavi import mlab

x, y, z = random.rand(3, 10)
mlab.points3d(x, y, z)
c = random.rand(10)
mlab.points3d(x, y, z, color=c)

mlab.clf()

mlab.contour3d(X, Y, Z, V)
```

## 11.4. Entorno completo: sage

El paquete `sage`, disponible libremente de la página `http://www.sagemath.org`, pretende proveer una manera de reemplazar a programas del estilo de Mathematica, al proveer un entorno completo para hacer cálculos matemáticos. Es un entorno que provee un interfaz tipo Python a muchos paquetes libres para hacer matemáticas. Además, provee un interfaz disponible por el internet para interactuar con los “notebooks”.