Notas del curso

# Programación en Python

Curso de actualización docente
Facultad de Ciencias

David P. Sanders

4 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)
```