Introducción a la programación en Python

Versión original en inglés de J.R. Johansson (http://jrjohansson.github.io/) (robert@riken.jp).

Traducido/Adaptado por G.F. Rubilar (http://google.com/+GuillermoRubilar).

La última versión de estos Notebooks (http://ipython.org/notebook.html) está disponible en http://github.com/gfrubi/clases-python-cientifico (http://github.com/gfrubi/clases-python-cientifico).

La última versión del original (en inglés) está disponible en http://github.com/jrjohansson/scientific-python-lectures (http://github.com/jrjohansson/scientific-python-lectures). Los otros notebooks de esta serie están listados en http://jrjohansson.github.com (http://jrjohansson.github.com).

Archivos de programa en Python

- El código Python es usualmente almacenado en archivos de texto con extensión " `.py` " (un "script"):

```
miprograma.py
```

- Se asume que cada línea de un archivo de programa en Python es una sentencia Python, o parte de una sentencia.
  - La única excepción son las líneas de comentarios, que comienzan con el caracter `#` (opcionalmente precedida por un número arbitrario de caracteres de espacio en blanco, es decir, tabs y espacios. Las líneas de comentarios son usualmente ignoradas por el intérprete Python.

- Para ejecutar nuestro programa Python desde la línea de comando usamos:

```
$ python miprograma.py
```

- En sistemas UNIX es común definir la ruta al intérprete en la primera línea del programa (note que ésta es una línea de comentarios en lo que respecta al intérprete Python):

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

  Si hacemos esto, y adicionalmente configuramos el archivo para que sea ejecutable, podemos correr el programa usando:

```
$ miprograma.py
```

Codificación de caracteres

La codificación estándar de caracteres es la ASCII, pero podemos usar cualquier otra codificación, por ejemplo UTF-8. Para especificar que usamos UTF-8 incluimos la línea especial

```
# -*- coding: UTF-8 -*-
```

al comienzo del archivo.

Aparte de estas dos líneas *opcionales* al comienzo de un archivo Python, no se requiere de otro código adicional para inicializar un programa. Por otro lado, en la versión 3 de Python ya no es necesario agregar código extra alguno.

Jupyter Notebooks

Este archivo - un Jupyter/IPython notebook - no sigue el patrón estándar de código Python en un archivo de texto. En su lugar, un notebook IPython es almacenado como un archivo en el formato JSON (http://es.wikipedia.org/wiki/JSON). La ventaja es que podemos mezclar texto formateado, código Python, y código de salida. Esto requiere estar ejecutando un servidor de notebook IPython, y por eso este tipo de archivo no es un programa Python independiente como se describió antes. Aparte de eso, no hay diferencia entre el código Python en un archivo de programa o en un notebook IPython.

Variables y tipos

Nombres de símbolos

Los nombres de las variables en Python pueden contener los caracteres `a-z`, `A-Z`, `0-9` y algunos caracteres especiales como `_`. Los nombres de variables normales deben comenzar con una letra.

Por convención, los nombres de las variables comienzan con letra minúscula, mientras que los nombres de las clases comienzan con una letra mayúscula.

Además, existen algunos palabras claves Python que no pueden ser usados como nombres de variables. Éstas son:

```
and, as, 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, with, yield
```

Nota: Atención con la palabra `lambda`, que podría fácilmente ser un nombre de variable natural en un programa científico. Sin embargo, como es una palabra clave, no puede ser usado como nombre de una variable.

Asignaciones

El operador para asignar valores en Python es el signo igual (`=`). Python es un lenguage de *escritura dinámica*, de modo que no necesitamos especificar el tipo de una variable cuando la creamos.

Al asignar un valor a una variable nueva se crea esa variable:

In [1]:
```
# asignaciones de variables
x = 1.0
mi_variable = 12.2
```

Aunque no se especifique explícitamente, cada variable sí tiene un tipo asociada a ella. El tipo es extraido del valor que le fue asignado.

In [2]:
```
type(x)
```

Out[2]: `float`

Si asignamos un nuevo valor a una variable, su tipo puede cambiar.

In [3]:
```
x = 1
```

In [4]:
```
type(x)
```

Out[4]:
```
int
```

Si tratamos de usar una variable que no ha sido definida obtenemo un mensaje de error (`NameError`):

In [5]:
```
#print(y)
```

Tipos Fundamentales

In [6]:
```
# enteros
x = 1
type(x)
```

Out[6]:
```
int
```

In [7]:
```
# flotantes
x = 1.0
type(x)
```

Out[7]:
```
float
```

In [8]:
```
# booleanos
b1 = True
b2 = False

type(b1)
```

Out[8]:
```
bool
```

In [9]:
```
# números complejos: note que se usa `j` para especificar la parte imaginaria
x = 1.0 - 1.0j
type(x)
```

Out[9]:
```
complex
```

In [10]:
```
print(x)
```

```
(1-1j)
```

In [11]:
```
print(x.real, x.imag)
```

```
1.0 -1.0
```

In [12]:
```
x = 1.0

# verifica si la variable x es flotante
type(x) is float
```

Out[12]:
```
True
```

In [13]:
```
# verifica si la variable x es un entero
type(x) is int
```

Out[13]:
```
False
```

Podemos también usar el método `isinstance` para testear tipos de variables:

In [14]:
```
isinstance(x, float)
```

Out[14]:
```
True
```

Conversión de Tipo

In [15]:
```
x = 1.5

print(x, type(x))
```

```
1.5 <class 'float'>
```

In [16]:
```
x = int(x)

print(x, type(x))
```

```
1 <class 'int'>
```

In [17]:
```
z = complex(x)

print(z, type(z))
```

```
(1+0j) <class 'complex'>
```

In [18]:
```
#x = float(z)
```

Un número complejo no puede ser convertido a un número flotante o a un entero. Necesitamos usar `z.real` , o bien `z.imag` , para extraer la parte que deseamos del número complejo z:

In [19]:
```
y = bool(z.real)

print(z.real, " -> ", y, type(y))

y = bool(z.imag)

print(z.imag, " -> ", y, type(y))
```

```
1.0  ->  True <class 'bool'>
0.0  ->  False <class 'bool'>
```

Operadores y comparaciones

La mayoría de los operadores y las comparaciones en Python funcionan como se esperaría:

- Operadores aritméticos `+`, `-`, `*`, `/`, `//` (división entera), '**' potencia

In [20]:
```
1 + 2, 1 - 2, 1 * 2, 1 / 2
```

Out[20]: (3, -1, 2, 0.5)

In [21]:
```
1.0 + 2.0, 1.0 - 2.0, 1.0 * 2.0, 1.0 / 2.0
```

Out[21]: (3.0, -1.0, 2.0, 0.5)

In [22]:
```
# División entera de dos númenos flotantes
3.0 // 2.0
```

Out[22]: 1.0

In [23]:
```
# Atención! El operador de potencia en Python no es ^, sino **
2**2
```

Out[23]: 4

- Los operadores booleanos se escriben como palabras: `and`, `not`, `or`.

In [24]:
```
True and False
```

Out[24]: False

In [25]:
```
not False
```

Out[25]: True

In [26]:
```
True or False
```

Out[26]: True

- Operadores de comparación `>`, `<`, `>=` (mayor o igual), `<=` (menor o igual), `==` igualdad, `es` idéntico.

In [27]:
```
2 > 1, 2 < 1
```

Out[27]: (True, False)

In [28]:
```
2 > 2, 2 < 2
```

Out[28]:
```
(False, False)
```

In [29]:
```
2 >= 2, 2 <= 2
```

Out[29]:
```
(True, True)
```

In [30]:
```
# igualdad
[1,2] == [1,2]
```

Out[30]:
```
True
```

In [31]:
```
# ¿objetos idénticos?
l1 = l2 = [1,2]

l1 is l2
```

Out[31]:
```
True
```

Tipos compuestos: Cadenas, listas y diccionarios

Cadenas

Las cadenas son el tipo de variables que es usado para almacenar mensajes de texto.

In [32]:
```
s = "Hola mundo"
type(s)
```

Out[32]:
```
str
```

In [33]:
```
# longitud de la cadena: el número de caracteres que contiene
len(s)
```

Out[33]:
```
10
```

In [34]:
```
# reemplaza una subcadena de una cadena por cadena
s2 = s.replace("mundo", "universo")
print(s2)
```

```
Hola universo
```

Podemos aislar un carácter en una cadena usando `[]` :

In [35]:
```
s[0]
```

Out[35]:
```
'H'
```

Atención usuarios de MATLAB: el indexado comienza en 0!

Podemos extraer una parte de una cadena usando la sintaxis `[desde:hasta]`, que extrae caracteres entre los índices `desde` y `hasta` **sin incluir el elemento con índice `hasta'**:

```
In [36]: s[0:5]
Out[36]: 'Hola '
```

Si omitimos `desde` o bien `hasta` de `[desde:hasta]`, por defecto se entiende que se refiere al comienzo y/o al fin de la cadena, respectivamente:

```
In [37]: s[:5]
Out[37]: 'Hola '
```

```
In [38]: s[6:]
Out[38]: 'undo'
```

```
In [39]: s[:]
Out[39]: 'Hola mundo'
```

Podemos también definir el tamaño del paso usando la sintaxis `[desde:hasta:paso]` (el valor por defecto de `paso` es 1, como ya vimos):

```
In [40]: s[::1]
Out[40]: 'Hola mundo'
```

```
In [41]: s[::2]
Out[41]: 'Hl ud'
```

Esta técnica es llamada *slicing* ("rebanado"). Puede leer más sobre la sintaxis aquí (http://pyspanishdoc.sourceforge.net/lib/built-in-funcs.html) y aquí (http://docs.python.org/release/2.7.3/library/functions.html?highlight=slice#slice) (en inglés).

Python tiene un rico conjunto de funciones para procesar texto. Ver por ejemplo la documentación en este link (http://docs.python.org/2/library/string.html) (en inglés) para más información.

Ejemplos de formateo de cadenas

In [42]:
```
print("uno", "dos", "tres")  # El comando print puede desplegar varias cadenas
```

```
uno dos tres
```

In [43]:
```
print("uno", 1.0, False, -1j)  # El comando print convierte todos los argument
os a cadenas
```

```
uno 1.0 False (-0-1j)
```

In [44]:
```
print("uno" + "dos" + "tres") # cadenas "sumadas" con + son contatenadas sin e
spacio entre ellas
```

```
unodostres
```

In [45]:
```
print("valor = "+str(1.0)) # podemos transformar un float a string y concatena
rlos en la salida
```

```
valor = 1.0
```

In [46]:
```
print("valor = %f" % 1.0) # podemos usar formateo de cadenas en el estilo del
 lenguaje C
```

```
valor = 1.000000
```

In [47]:
```
# este formateo crea una cadena
s2 = "valor1 = %.2f. valor2 = %d" % (3.1415, 1.5)

print(s2)
```

```
valor1 = 3.14. valor2 = 1
```

In [48]:
```
# forma alternativa, más intuitiva para formatear una cadena
s3 = 'valor1 = {0}, valor2 = {1}'.format(3.1415, 1.5)

print(s3)
```

```
valor1 = 3.1415, valor2 = 1.5
```

Listas

Listas son muy similares a las cadenas, excepto que cada elemento puede ser de un tipo diferente.

La sintaxis para crear listas en Python es `[..., ..., ...]` :

In [49]:
```
l = [1,2,3,4]

print(type(l))
print(l)
```

```
<class 'list'>
[1, 2, 3, 4]
```

Podemos usar las mismas técnicas de "rebanado" que usamos en el caso de cadenas para manipular listas:

In [50]:
```
print(l)

print(l[1:3])

print(l[::2])
```

```
[1, 2, 3, 4]
[2, 3]
[1, 3]
```

Atención usuarios de MATLAB: el indexado comienza en 0!

In [51]:
```
l[0]
```

Out[51]:
```
1
```

Los elementos en una lista no requieren ser del mismo tipo:

In [52]:
```
l = [1, 'a', 1.0, 1-1j]

print(l)
```

```
[1, 'a', 1.0, (1-1j)]
```

Las listas en Python pueden ser *inhomogéneas* y *arbitrariamente anidadas*:

In [53]:
```
lista_anidada = [1, [2, [3, [4, [5]]]]]

lista_anidada
```

Out[53]:
```
[1, [2, [3, [4, [5]]]]]
```

Las listas juegan un rol muy importante en Python y son, por ejemplo, usadas en bucles y otras estructuras de control de flujo (discutidas más abajo). Existen muchas funciones convenientes para generar listas de varios tipos, por ejemplo la función `range` :

In [54]:
```
desde = 10
hasta = 30
paso = 2

range(desde, hasta, paso)
```

Out[54]:
```
range(10, 30, 2)
```

In [55]:
```
# en Python 3 range genera un interador, que puede ser convertido a una lista
 usando 'list(...)'. Esto no tiene efecto en Python 2
list(range(desde, hasta, paso))
```

Out[55]:
```
[10, 12, 14, 16, 18, 20, 22, 24, 26, 28]
```

In [56]:
```
list(range(-10, 10))
```

Out[56]:
```
[-10, -9, -8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [57]:
```
s
```

Out[57]:
```
'Hola mundo'
```

In [58]:
```
# convierte una cadena a una lista, por conversión de tipo:

s2 = list(s)

s2
```

Out[58]:
```
['H', 'o', 'l', 'a', ' ', 'm', 'u', 'n', 'd', 'o']
```

In [59]:
```
# ordenando listas
s2.sort()

print(s2)
```

```
[' ', 'H', 'a', 'd', 'l', 'm', 'n', 'o', 'o', 'u']
```

Agregando, insertando, modificando, y removiendo elementos de listas

In [60]:
```
# crea una nueva lista vacía
l = []

# agrega un elemento a la lista, usando `append`
l.append("A")
l.append("d")
l.append("d")

print(l)
```

```
['A', 'd', 'd']
```

Podemos modificar listas asignando nuevos valores a los elementos de la lista. En lenguaje técnico se dice que la lista es *mutable*.

In [61]:
```
l[1] = "p"
l[2] = "p"

print(l)
```

```
['A', 'p', 'p']
```

In [62]:
```
l[1:3] = ["d", "d"]

print(l)
```

```
['A', 'd', 'd']
```

Insertar un elemento en una posición específica `insert`

In [63]:
```
l.insert(0, "i")
l.insert(1, "n")
l.insert(2, "s")
l.insert(3, "e")
l.insert(4, "r")
l.insert(5, "t")

print(l)
```

```
['i', 'n', 's', 'e', 'r', 't', 'A', 'd', 'd']
```

Eliminar el primer elemento con un valor específico usando 'remove'

In [64]:
```
l.remove("A")

print(l)
```

```
['i', 'n', 's', 'e', 'r', 't', 'd', 'd']
```

Eliminar un elemento en una posición específica usando `del` :

Puede introducir `help(list)` para más detalles, o leer la documentación en la red

Tuplas

Tuplas son similares a las listas, excepto que ellas no pueden ser modificadas una vez creadas, es decir, son *inmutables*.

En Python, las tuplas son creadas usando la sintaxis `(..., ..., ...)` , o incluso `..., ...` :

In [65]:
```
punto = (10, 20)

print(punto, type(punto))
```

```
(10, 20) <class 'tuple'>
```

In [66]:
```
punto = 10, 20

print(punto, type(punto))
```

```
(10, 20) <class 'tuple'>
```

Podemos separar una tupla asignándola a una lista de variables separadas por coma:

In [67]:
```
x, y = punto

print("x =", x)
print("y =", y)
```

```
x = 10
y = 20
```

Si intentamos asignar un nuevo valor a un elemento de una tupla obtenemos un error:

In [68]:
```
#punto[0] = 20
```

Diccionarios

Los diccionarios son también como listas, excepto que cada elemento es un par clave-valor. La sintaxis de los diccionarios es `{clave1 : valor1, ...}` :

In [69]:
```
parametros = {"clave1" : 1.0, "clave2" : True, "clave3" : "hola"}

print(type(parametros))
print(parametros)
```

```
<class 'dict'>
{'clave1': 1.0, 'clave2': True, 'clave3': 'hola'}
```

In [70]:
```
parametros["clave1"]
```

Out[70]:
```
1.0
```

In [71]:
```
print("clave1 --> " + str(parametros["clave1"]))
print("clave2 --> " + str(parametros["clave2"]))
print("clave3 --> " + str(parametros["clave3"]))
```

```
clave1 --> 1.0
clave2 --> True
clave3 --> hola
```

In [72]:
```
parametros["clave1"] = "A"
parametros["clave2"] = (4<2)

# agrega una nueva entrada
parametros["clave4"] = "D"

print("clave1 = " + str(parametros["clave1"]))
print("clave2 = " + str(parametros["clave2"]))
print("clave3 = " + str(parametros["clave3"]))
print("clave4 = " + str(parametros["clave4"]))
```

```
clave1 = A
clave2 = False
clave3 = hola
clave4 = D
```

Control de flujo

Sentencias condicionales: if, elif, else

La sintaxis Python para la ejecución condicional de código usa las palabras clave `if`, `elif` (else if), `else`:

In [73]:
```
afirmacion1 = False
afirmacion2 = False

if afirmacion1:
    print("afirmacion1 es verdadera")

elif afirmacion2:
    print("afirmacion2 es verdadera")

else:
    print("afirmacion1 y afirmacion2 son falsas")
```

```
afirmacion1 y afirmacion2 son falsas
```

Aquí encontramos por primera vez un aspecto pecular e inusual del lenguaje Python: Los bloques del programa son definidos por su nivel de indentación (la cantidad de espacio antes de cada linea).

Compare con el código equivalente en C:

```
if (afirmacion1)
{
    printf("afirmacion1 es verdadera\n");
}
else if (afirmacion2)
{
    printf("afirmacion1 es verdadera\n");
}
else
{
    printf("afirmacion1 y afirmacion2 son falsas\n");
}
```

En C los bloques son definidos por los paréntesis llaves `{` y `}`. El nivel de indentación (espacio en blanco antes del código) no importa (es completamente opcional).

En Python, la extensión de un bloque de código es definido por el nivel de indentación (usualmente un tab o cuatro espacios en blanco). Esto significa que debemos ser cuidados@s de indentar nuestro código correctamente, de lo contrario tendremos errores de sintaxis. Además, pueden presentarse errores en la ejecución si se es inconsistente en la forma que se realiza el indentado en un mismo programa.

Ejemplos:

In [74]:
```
afirmacion1 = afirmacion2 = True

if afirmacion1:
    if afirmacion2:
        print("tanto afirmacion1 como afirmacion2 son verdaderas")
```

```
tanto afirmacion1 como afirmacion2 son verdaderas
```

In [75]:
```
# Mala indentación!
#if afirmacion1:
#    if afirmacion2:
#    print("tanto afirmacion1 como afirmacion2 son verdaderas")  # esta línea
 está mal indentada
```

In [76]:
```
afirmacion1 = False

if afirmacion1:
    print("afirmacion1 es verdadera")

    print("aun estamos dentro del bloque if")
```

In [77]:
```
if afirmacion1:
    print("afirmacion1 es verdadera")

print("ahora estamos fuera del bloque")
```

```
ahora estamos fuera del bloque
```

Ciclos

En Python, los ciclos (loops) puede ser programados de varias maneras diferentes. La forma más común es usando un ciclo `for`, que se usa junto con objetos iterables, como por ejemplos las listas. La sintaxis básica es:

Ciclos `for`:

In [78]:
```
for x in [1,2,3]:
    print(x)
```

```
1
2
3
```

El ciclo `for` itera sobre los elementos de la lista suministrada y ejecuta el bloque suministrado una vez para cada elemento. Cualquier tipo de lista puede ser usada para un ciclo `for`. Por ejemplo:

In [79]:
```
for x in range(4): # por defecto range comienza con 0
    print(x)
```

```
0
1
2
3
```

Nota: `range(4)` no incluye el 4 !

In [80]:
```
for x in range(-3,3):
    print(x)
```

```
-3
-2
-1
0
1
2
```

In [81]:
```
for palabra in ["computación", "científica", "con", "Python"]:
    print(palabra)
```

```
computación
científica
con
Python
```

Para iterar sobre pares clave-valor en un diccionario:

In [82]:
```
for clave, valor in parametros.items():
    print(clave + " = " + str(valor))
```

```
clave1 = A
clave2 = False
clave3 = hola
clave4 = D
```

Algunas veces es útil tener acceso a los índices de los valores mientras se itera sobre una lista. Podemos usar la función `enumerate` para esto:

In [83]:
```
for idx, x in enumerate(range(-3,3)):
    print(idx, x)
```

```
0 -3
1 -2
2 -1
3 0
4 1
5 2
```

Listas: Creando listas usando ciclos `for`:

Una forma conveniente y compacta de inicializar listas:

In [84]:
```
l1 = [x**2 for x in range(0,5)]

print(l1)
```

```
[0, 1, 4, 9, 16]
```

Ciclos `while`:

In [85]:
```
i = 0

while i < 5:
    print(i)
    i = i + 1

print("listo")
```

```
0
1
2
3
4
listo
```

Note que el comando `print("listo")` no es parte del cuerpo del ciclo `while`, debido a su indentación.

Funciones

En Python una función es definida usando la palabra clave `def`, seguida de un nombre para la función, una variable entre paréntesis `()`, y el símbolo de dos puntos `:`. El siguiente código, con un nivel adicional de indentación, es el cuerpo de la función.

In [86]:
```
def func0():
    print("test")
```

In [87]:
```
func0()
```

```
test
```

En forma opcional, pero muy recomendada, podemos definir un "docstring", que es una descripción del propósito y comportamiento de la función. El docstring debería ser incluido directamente después de la definición de la función, antes del código en el cuerpo de la función.

In [88]:
```
def func1(s):
    """
    Imprime la cadena 's' y cuántos caracteres tiene
    """

    print(s + " tiene " + str(len(s)) + " caracteres")
```

In [89]:
```
help(func1)
```

```
Help on function func1 in module __main__:

func1(s)
    Imprime la cadena 's' y cuántos caracteres tiene
```

In [90]:
```
func1("test")
```

```
test tiene 4 caracteres
```

Funciones que retornan un valor usan la palabra clave `return` :

In [91]:
```
def cuadrado(x):
    """
    Calcula el cuadrado de x.
    """
    return x**2
```

In [92]:
```
cuadrado(4)
```

Out[92]:
```
16
```

Podemos retornar múltiples valores desde una función usando las tuplas (ver más arriba):

In [93]:
```
def potencias(x):
    """
    Calcula algunas potencias de x.
    """
    return x**2, x**3, x**4
```

In [94]:
```
potencias(3)
```

Out[94]:
```
(9, 27, 81)
```

In [95]:
```
x2, x3, x4 = potencias(3)

print(x3)
```

```
27
```

Argumentos por defecto y argumentos de palabra clave

En la definición de una función, podemos asignar valores por defecto a los argumentos de la función:

In [96]:
```
def mifunc(x, p=2, debug=False):
    if debug:
        print("evaluando mifunc para x = " + str(x) + " usando el exponente p
 = " + str(p))
    return x**p
```

Si no suministramos un valor para el argumento `debug` al llamar a la función `mifunc` se considera el valor definido por defecto:

In [97]:
```
mifunc(5)
```

Out[97]:
```
25
```

In [98]:
```
mifunc(5, debug=True)
```

```
evaluando mifunc para x = 5 usando el exponente p = 2
```

Out[98]:
```
25
```

Si listamos explícitamente el nombre de los argumentos al llamar una función, ellos no necesitan estar en el mismo orden usado en la definición de la función. Esto es llamado argumentos *de palabra clave* (keyword), y son a menudo muy útiles en funciones que requieren muchos argumentos opcionales.

In [99]:
```
mifunc(p=3, debug=True, x=7)
```

```
evaluando mifunc para x = 7 usando el exponente p = 3
```

Out[99]:
```
343
```

Funciones sin nombre (funciones lambda)

En Python podemos también crear funciones sin nombre, usando la palabra clave `lambda` :

In [100]:
```
f1 = lambda x: x**2

# es equivalente a

def f2(x):
    return x**2
```

In [101]:
```
f1(2), f2(2)
```

Out[101]:
```
(4, 4)
```

Esta técnica es útil, por ejemplo, cuando queremos pasar una función simple como argumento de otra función, como en este caso:

In [102]:
```
# map es una función predefinida en Python
map(lambda x: x**2, range(-3,4))
```

Out[102]:
```
<map at 0x2129ee73668>
```

In [103]:
```
# en Python 3 podemos usar `list(...)` para convertir la iteración a una lista explícita
list(map(lambda x: x**2, range(-3,4)))
```

Out[103]:
```
[9, 4, 1, 0, 1, 4, 9]
```

Clases

Las clases son una característica clave de la programación orientada al objeto. Una clase es una estructura para representar un objeto y las operaciones que pueden ser realizadas sobre el objeto.

En Python una clase puede contener *atributos* (variables) y *métodos* (funciones).

En Python una clase es definida en forma similar a una función, pero usando la palabra clave `class`, y la definición de la clase usualmente contiene algunas definiciones de métodos (una función en una clase).

- Cada método de una clase debería tener un argumento `self` como su primer argumento. Este objeto es una autoreferencia.
- Algunos nombres de métodos de clases tienen un significado especial, por ejemplo:
  - `__init__`: El nombre del método que es invocado cuando el objeto es creado por primera vez.
  - `__str__`: Un método que es invocado cuando se necesita una simple representación de cadena de la clase, como por ejemplo cuando se imprime.
  - Existen muchos otros métodos especiales, ver http://docs.python.org/2/reference/datamodel.html#special-method-names (http://docs.python.org/2/reference/datamodel.html#special-method-names)

Ejemplo: Puntos en 2D

In [104]:
```
class Punto:
    """
    Clase simple para representar un punto en un sistema de coordenadas cartesiano bidimensional.
    """

    def __init__(self, x, y):
        """
        Crea un nuevo punto en x, y.
        """
        self.x = x
        self.y = y

    def traslada(self, dx, dy):
        """
        Traslada el punto en dx y dy en las direcciones x e y respectivamente.
        """
        self.x += dx
        self.y += dy

    def __str__(self):
        return "Punto en [%f, %f]" % (self.x, self.y)
```

Ahora creamos un "Punto", es decir, una instancia de la clase "Punto":

In [105]:
```
p1 = Punto(0,0) # esto invoca el método __init__ en la clase Punto

print(p1)         # esto invoca el método __str__
```

```
Punto en [0.000000, 0.000000]
```

In [106]:
```
type(p1)
```

Out[106]:
```
__main__.Punto
```

Si ahora ejecutamos

In [107]:
```
p2 = Punto(1,1)

p1.traslada(0.25,1.5)

print(p1)
print(p2)
```

```
Punto en [0.250000, 1.500000]
Punto en [1.000000, 1.000000]
```

Estamos definiendo una nueva instancia (un nuevo "punto", asignado a la variable p2), y luego hemos "invocado el método `traslada` en la instancia `p1` de la clase punto" lo que representa la acción de realizar una traslación del punto p1.

Note que llamar a métodos de clases puede modificar el estado de esa instancia de clase particular, pero no afecta otras instancias de la clase o alguna otra variable global.

Esto es una de las cosas buenas de un diseño orientado al objeto: código como las funciones y variables relacionadas son agrupadas en entidades separadas e independientes.

Módulos

La mayoría de la funcionalidad en Python es provista por *módulos*. La *Librería Estándar* (https://docs.python.org/2/library/) de Python es una gran colección de módulos que proveen implementaciones *multiplataforma* de recursos tales como el acceso al sistema operativo, entrada/salido de archivos (file I/O), manejo de cadenas, comunicación en redes, y mucho más.

Para usar un módulo en un programa Python éste debe primero ser **importado**, para lo cual se usa el comando `import`. Por ejemplo, para importar el módulo `math`, que contiene muchas funciones matemáticas estándar, podemos usar:

In [108]:
```
import math
```

Esto importa el módulo completo y lo deja disponible para su uso en el programa. Por ejemplo, podemos escribir:

In [109]:
```
import math

x = math.cos(2*math.pi)

print(x)
```

```
1.0
```

Alternativamente, podemos elegir importar todos los símbolos (funciones y variables) en un módulo al espacio de nombres (namespace) actual (de modo que no necesitemos usar el prefijo "`math.`" cada vez que usemos algo del módulo `math`:

In [110]:
```
from math import *

x = cos(2*pi)

print(x)
```

```
1.0
```

Esta forma de proceder puede ser muy conveniente, pero en programas largos que incluyen muchos módulos es a menudo una buena idea mantener los símbolos de cada módulo en sus propios espacios de nombres, usando `import math` . Esto elimina potenciales confusiones con eventuales colisiones de nombres, ya que no es poco común encontrar funciones o variables definidas con el mismo nombre en módulos distintos.

Como alternativa intermedia, podemos importar un módulo con un *alias* o nombre abreviado:

In [111]:
```
import math as m

x = m.cos(2*m.pi)

print(x)
```

```
1.0
```

Finalmente, podemos importar sólo algunos símbolos seleccionados desde un módulo listándolos explícitamente, en lugar de usar el carácter comodín `*` :

In [112]:
```
from math import cos, pi

x = cos(2*pi)

print(x)
```

```
1.0
```

Mirando qué contiene un módulo, y su documentación

Luego que se ha cargado un módulo, podemos listar los símbolos que éste provee usando la función `dir` :

In [113]:
```
import math

dir(math)
```

Out[113]:
```
['__doc__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'acos',
 'acosh',
 'asin',
 'asinh',
 'atan',
 'atan2',
 'atanh',
 'ceil',
 'copysign',
 'cos',
 'cosh',
 'degrees',
 'e',
 'erf',
 'erfc',
 'exp',
 'expm1',
 'fabs',
 'factorial',
 'floor',
 'fmod',
 'frexp',
 'fsum',
 'gamma',
 'gcd',
 'hypot',
 'inf',
 'isclose',
 'isfinite',
 'isinf',
 'isnan',
 'ldexp',
 'lgamma',
 'log',
 'log10',
 'log1p',
 'log2',
 'modf',
 'nan',
 'pi',
 'pow',
 'radians',
 'remainder',
 'sin',
 'sinh',
 'sqrt',
 'tan',
 'tanh',
 'tau',
 'trunc']
```

Usando la función `help` podemos obtener una descripción de cada función (casi... no todas las funciones tienen *docstrings*, como se les llama técnicamente. Sin embargo, la mayoría de las funciones están documentadas de esta forma).

In [114]:
```
help(math.log)
```

```
Help on built-in function log in module math:

log(...)
    log(x, [base=math.e])
    Return the logarithm of x to the given base.

    If the base not specified, returns the natural logarithm (base e) of x.
```

In [115]:
```
log(10) # calcula el logaritmo de 10 en base e
```

Out[115]:
```
2.302585092994046
```

In [116]:
```
log(10, 2) # calcula el logaritmo de 10 en base 2
```

Out[116]:
```
3.3219280948873626
```

También podemos usar la función `help` directamente sobre los módulos:

```
help(math)
```

Algunos módulos muy útiles de la librería estándar de Python son `os` (interfaz con el sistema operativo), `sys` (Parámetros y funciones específicas del sistema), `math` (funciones matemáticas), `shutil` (operaciones con archivos), `subprocess` , `multiprocessing` , `threading` .

Una lista completa de los módulos estándar para Python 2 y Python 3 está disponible (en inglés) en http://docs.python.org/2/library/ (http://docs.python.org/2/library/) y http://docs.python.org/3/library/ (http://docs.python.org/3/library/), respectivamente. Una versión en español está disponible en http://pyspanishdoc.sourceforge.net/lib/lib.html (http://pyspanishdoc.sourceforge.net/lib/lib.html).

Existen muchos otros módulos (paquetes) desarrollados para Python que implementan distintas funcionalidades, herramientas y algoritmos. Muchos de ellos son constantemente desarrollados en forma abierta por comunidades de usuari@s interesad@s.

Aquí listamos algunos módulos generales útiles en el ámbito de las ciencias (Físicas):

- Numpy (http://www.numpy.org/): Implementa el uso eficiente de arreglos numéricos multidimensionales (vectores, matrices, etc.).
- Scipy (http://www.scipy.org/): Implementa múltiples funciones especiales, algoritmos de integración numérica, optimización, interpolación, transformada de Fourier, procesamiento de señales, álgebra lineal, estadística, procesamiento de imágenes, entre otras. Este módulo hace uso de Numpy.
- Matplotlib (http://matplotlib.org/): Suministra herramientas para crear gráficos bidimensionales en diversos formatos, y en una calidad adecuada para incluirlos en publicaciones científicas.
- Sympy (http://sympy.org/): Paquete que implementa algoritmos de matemática simbólica.

Además de estos paquetes principales (más usados) cabe la pena mencionar a:

- Sunpy (http://sunpy.org/): Módulo para el análisis de datos relacionados con la física solar.
- EMpy (http://sunpy.org/): Suministra algoritmos numéricos usados en electromagnetismo.
- Mpmath (http://mpmath.org/): Módulo con herramientas para cálculos con valores reales y complejos con precisión arbitraria.
- Poliastro (http://poliastro.readthedocs.io/en/latest/): Conjunto de rutinas Python útiles en astrodinámica y mecánica orbital.
- Fatiando a Terra (http://fatiando.org/): Conjunto de herramientas para la modelación de fenómenos geofísicos.
- Qutip2 (http://qutip.org/): Paquete de herramientas para simular la dinámica de sistemas cuánticos abiertos.
- Yt (http://yt-project.org/): Paquete para el análisis y visualización de datos volumétricos.
- Fipy (http://www.ctcms.nist.gov/fipy/): Implementa algoritmos para resolver ecuaciones diferenciales parciales (EDP) por medio de métodos de volúmenes finitos.
- Holopy (http://manoharan.seas.harvard.edu/holopy/): Herramientas para el trabajo con hologramas digitales y scattering de luz.
- Astropy (http://www.astropy.org/): Módulo que implementa herramientas de uso común en Astronomía.
- Librosa (http://librosa.github.io/librosa/): Módulo para análisis de audio y música.
- Scikit-learn (http://scikit-learn.org/): Implementa funciones de "Machine Learning" (Clasificación, Regresión, Clustering, reducción dimensional, Selección de Modelos, etc.)

Creación de Módulos

Uno de los conceptos más importantes en programación es el de reusar código para evitar repeticiones.

La idea es escribir funciones y clases con un propósito y extensión bien definidos, y reusarlas en lugar de repetir código similar en diferentes partes del programa (programación modular). Usualmente el resultado es que se mejora ostensiblemente la facilidad de lectura y de mantención de un programa. En la práctica, esto significa que nuestro programa tendrá menos errores, y serán más fáciles de extender y corregir.

Python permite programación modular en diferentes niveles. Las funciones y las clases son ejemplos de herramientas para programación modular de bajo nivel. Los módulos Python son construcciones de programación modular de más alto nivel, donde podemos colectar variables relacionadas, funciones y clases. Un módulo Python es definido en un archivo Python (con extensión `.py`), y puede ser accequible a otros módulos Python y a programas usando el comendo `import`.

Considere el siguiente ejemplo: el archivo `mimodulo.py` contiene una implementación simple de una variable, una función y una clase:

Podemos importar el módulo `mimodulo` a un programa Python usando `import`,

```
In [117]: import mimodulo
```

y usar `help(module)` para obtener un resumen de lo que suministra el módulo:

In [118]:
```
help(mimodulo)
```

```
Help on module mimodulo:

NAME
    mimodulo

DESCRIPTION
    Ejemplo de un módulo Python. Contiene una variable llamada mi_variable,
    una función llamada mi_function, y una clase llamada MiClase.

CLASSES
    builtins.object
        MiClase

    class MiClase(builtins.object)
     |  Clase ejemplo.
     |
     |  Methods defined here:
     |
     |  __init__(self)
     |      Initialize self.  See help(type(self)) for accurate signature.
     |
     |  get_variable(self)
     |
     |  set_variable(self, nuevo_valor)
     |      Asigna self.variable a un nuevo valor
     |
     |  ----------------------------------------------------------------------
-
     |  Data descriptors defined here:
     |
     |  __dict__
     |      dictionary for instance variables (if defined)
     |
     |  __weakref__
     |      list of weak references to the object (if defined)

FUNCTIONS
    mi_function()
        Función ejemplo

DATA
    mi_variable = 0

FILE
    c:\users\fsanmartin\data science - juan gabriel gomila\mimodulo.py
```

In [119]:
```
mimodulo.mi_variable
```

Out[119]:
```
0
```

In [120]:
```
mimodulo.mi_function()
```

Out[120]: 0

In [121]:
```
mi_clase = mimodulo.MiClase()
mi_clase.set_variable(10)
mi_clase.get_variable()
```

Out[121]: 10

Excepciones

En Python los errores son manejados con una construcción especial de lenguaje llamada "Exceptions" (excepciones). Cuando ocurre un error, una excepción puede ser hecha, que interrumpe el flujo normal del programa y retorna a algún otro lugar del código donde se definan los comandos try-except más cercanos.

Para generar una excepción podemos usar el comando `raise`, que toma un argumento que debe ser una instancia de la clase `BaseExpection` o una clase derivada de ella.

Un uso típico de las excepciones es para abortar funciones cuando ocurre algún error, por ejemplo:

```
def mi_funcion(argumentos):

    if not verify(argumentos):
        raise Expection("Argumentos invalidos")

    # el resto del código sigue aquí
```

Para capturar los errores que son generados por funciones y métodos de clases, o por el mismo intérprete Python, use los comandos `try` y `except`:

```
try:
    # aquí va el código normal
except:
    # el código para manejar el error va aquí
    # Este código no se ejecuta a menos que
    # el código de arriba genere un error
```

Por ejemplo:

In [122]:
```
try:
    print("test")
    # genera un error: ya que la variable test no está definida
    print(test)
except:
    print("Encontré una excepción")
```

```
test
Encontré una excepción
```

Para obtener información sobre un error, podemos accesar la instancia de clase `Exception` que describe la excepción usando por ejemplo:

```
except Exception as e:
```

In [123]:
```
try:
    print("test")
    # genera un error: ya que la variable test no está definida
    print(test)
except Exception as e:
    print("Encontré una excepción:" + str(e))
```

```
test
Encontré una excepción:name 'test' is not defined
```

Lectura adicional

- http://www.python.org (http://www.python.org) - The official web page of the Python programming language.
- http://www.python.org/dev/peps/pep-0008 (http://www.python.org/dev/peps/pep-0008) - Guía de estilo para la programación en Python. Altamente recomendada (en inglés).
- http://www.greenteapress.com/thinkpython/ (http://www.greenteapress.com/thinkpython/) - Un libro gratuito sobre Python.
- Python Essential Reference (http://www.amazon.com/Python-Essential-Reference-4th-Edition/dp/0672329786) - Un buen libro de referencia sobre programación en Python.

Versiones

In [124]:
```
import sys
import IPython
```

In [125]:
```
print("Este notebook fue evaluado con: Python %s y IPython %s." % (sys.version
, IPython.__version__))
```

```
Este notebook fue evaluado con: Python 3.7.3 (default, Apr 24 2019, 15:29:51)
[MSC v.1915 64 bit (AMD64)] y IPython 7.6.1.
```

In [ ]: