

notebook1

June 11, 2017

1 Introducción a Python para ciencias e ingenierías (notebook 1)

Docente: Ing. Martín Gaitán

Links útiles

Descarga de la suite "Anaconda" (Python 3.6)

1.0.1 <http://continuum.io/downloads>

Repositorio de "notebooks" (material de clase)

1.0.2 <http://bit.ly/cursopy>

Python "temporal" online:

1.0.3 <http://try.jupyter.org>

1.1 ¿Empecemos!

Python es un lenguaje de programación:

- Interpretado e Interactivo
- Fácil de aprender, programar y **leer** (menos *bugs*)
- De *muy alto nivel*
- Multiparadigma
- Orientado a objetos
- Libre y con licencia permisiva
- Eficiente
- Versátil y potente!
- Con gran documentación
- Y una gran comunidad de usuarios

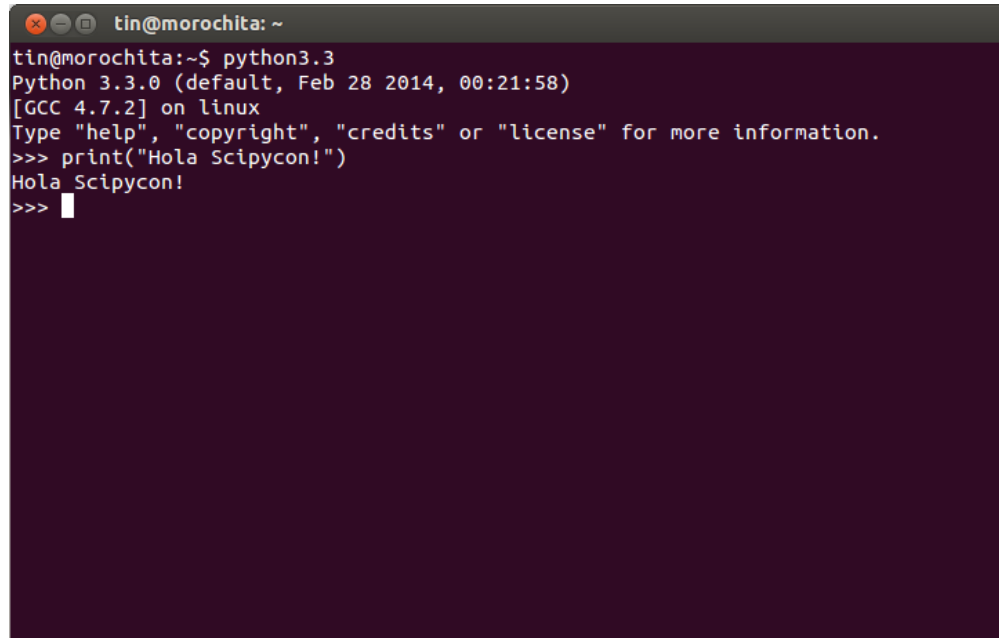
1.1.1 Instalación

- En Windows o mac: recomendación [Anaconda](#). **Instalá la versión basada en Python 3** que corresponda a tu Sistema Operativo
- En linux directamente puedes instalar todo lo necesario desde tus repositorios. Por ejemplo en Ubuntu:

```
sudo apt-get install ipython3-notebook python3-matplotlib python3-numpy python3-scipy`
```

1.1.2 ¿Cómo se usa Python?

Consolas interactivas Hay muchas maneras de usar el lenguaje Python. Dijimos que es un lenguaje **interpretado** e **interactivo**. Si ejecutamos la consola (En windows cmd.exe) y luego python, se abrirá la consola interactiva

A screenshot of a terminal window with a dark background. The window title is 'tin@morochita: ~'. The prompt is 'tin@morochita:~\$'. The user has entered 'python3.3'. The output shows 'Python 3.3.0 (default, Feb 28 2014, 00:21:58)', '[GCC 4.7.2] on linux', and a message to type 'help', 'copyright', 'credits' or 'license' for more information. The user has entered '>>> print("Hola Scipycon!")' and the output is 'Hola Scipycon!'. The prompt '>>>' is followed by a cursor.

En la consola interactiva podemos escribir sentencias o pequeños bloques de código que son ejecutados inmediatamente. Pero *la consola interactiva* estándar es **limitada**. Mucho mejor es usar **IPython**.

La consola IPython supera a la estándar en muchos sentidos. Podemos autocompletar (<TAB>), ver ayuda rápida de cualquier objeto (?) y muchas cosas más.

IPython Notebook (Jupyter) Y otra forma muy útil es usar los *Notebooks*. Jupyter es un entorno web para computación interactiva.

Si bien nació como parte del proyecto IPython, el mismo entorno visual se puede conectar a "*kernels*" de distintos lenguajes. Se puede usar Jupyter con Python, Julia, R, Octave y decenas de lenguajes más.

Podemos crear y editar "celdas" de código Python que podés editar y volver a ejecutar, podés intercalar celdas de texto, fórmulas matemáticas, y hacer que gráficos se muestren inscrutados en la misma pantalla. Estos archivos se guardan con extensión *.ipynb*, que pueden exportarse a diversos formatos estáticos como html o como código python puro. (.py)

Los notebooks son muy útiles para la "**programación exploratoria**", muy frecuente en ciencia e ingeniería

Todo el material de estos cursos estarán en formato notebook.

Para ejecutar IPython Notebook, desde la consola tipear:

```
jupyter notebook
```

```
tin@morochita: ~/lab/curso-python-cientifico
(curso)tin@morochita:~/lab/curso-python-cientifico$ ipython
Python 3.3.0 (default, Feb 28 2014, 00:21:58)
Type "copyright", "credits" or "license" for more information.

IPython 2.3.0 -- An enhanced Interactive Python.
?      -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help    -> Python's own help system.
object? -> Details about 'object', use 'object??' for extra details.

In [1]: print("Hola")
Hola

In [2]: range?
Type:      type
String form: <class 'range'>
Namespace: Python builtin
Docstring:
range([start,] stop[, step]) -> range object

Returns a virtual sequence of numbers from start to stop by step.

In [3]:
```

Programas También podemos usar Python para hacer programas o scripts. Esto es, escribir nuestro código en un archivo con extensión .py y ejecutarlo con el intérprete de python. Por ejemplo, el archivo hello.py (al que se le llama módulo) tiene este contenido:

```
print("¡Hola curso!")
```

Si ejecutamos python scripts/hello.py se ejecutará en el interprete Python y obtendremos el resultado

```
In [1]: print('Hola curso')
```

```
Hola curso
```

```
In [2]: !python3 scripts/hello.py
```

```
¡Hola curso!
```

```
In [ ]:
```

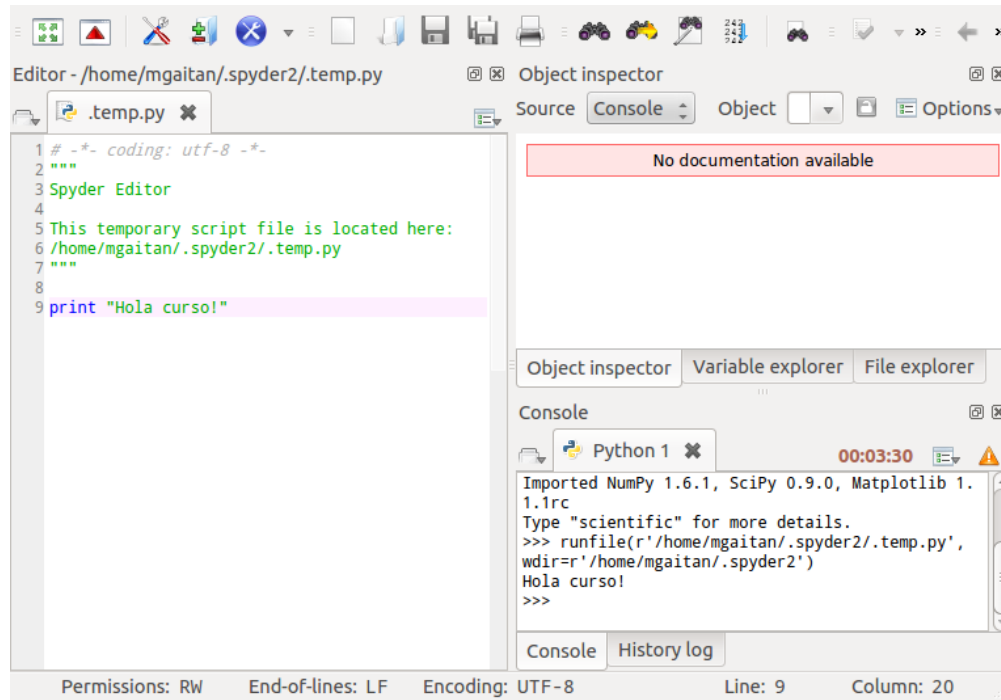
IPython agrega muchas funcionalidades complementarias que no son parte del lenguaje Python. Por ejemplo el signo ! que precede la línea anterior indica que se ejecutará un programa/comando del sistema en vez de código python

1.1.3 ¿Qué editor usar?

Python no exige un editor específico y hay muchos modos y maneras de programar.

Un buen editor orientado a Python científico es **Spyder**, que es un entorno integrado (editor + ayuda + consola interactiva)

También el entorno Jupyter trae un editor sencillo



1.1.4 ¿Python 2 o Python 3?

Hay dos versiones **actuales** de Python. La rama 2.7 (actualmente la version 2.7.9) y la rama 3 (actualmente 3.6.1). Todas las bibliotecas científicas de Python funcionan con ambas versiones. Pero Python 3 es aún más simple en muchos sentidos y es el que permanecerá a futuro!

2 ¿Queremos programar!

2.0.1 En el principio: Números

Python es un lenguaje de muy alto nivel y por lo tanto trae muchos *tipos* de datos incluidos.

```
In [4]: 1 + 1.4 - 12
```

```
Out[4]: -9.6
```

Ejecuten su consola y aa practicar!

```
In [5]: 29348575847598437598437598347598435**3
```

```
Out[5]: 2527907016281460289241633290916723098915975007982679438262423904713108863758862408610840
```

```
In [8]: 5 % 3
```

```
Out[8]: 2
```

Los tipos numéricos básicos son *int* (enteros sin limite), *float* (reales,) y *complex* (complejos)

```
In [6]: (3.2 + 12j) * 2
```

```
Out[6]: (6.4+24j)
```

```
In [ ]: 0.1 + 0.3
```

```
In [ ]: 3 // 2
```

```
In [ ]: 3 % 2
```

Las operaciones aritméticas básicas son:

- adición: +
- sustracción: -
- multiplicación: *
- división: /
- módulo (resto de división): %

- potencia: **
- división entera: //

Las operaciones se pueden agrupar con paréntesis y tienen precedencia estándar

```
In [ ]: x = 1.32
        resultado = ((21.2 + 4.5)**0.2 / x) + 1j
        print(resultado)
        resultado + 2
```

Outs vs prints

- La función `print` *imprime* (muestra) el resultado por salida estándar (pantalla) pero **no devuelve un valor** (estrictamente devuelve `None`). Quiere decir que el valor mostrado no queda disponible para seguir computando.
- Si la última sentencia de una celda tiene un resultado distinto a `None`, se guarda y se muestra en `Out [x]`
- Las últimas ejecuciones se guardan en variables automáticas `_`, `__` (última y anteúltima) o en general `_x` o `Out [x]`

```
In [15]: int(1.4)
```

```
Out[15]: 1
```

In []: 1 + 2J

```
In [ ]: Out[6]      # que es Out (sin corchetes)? Pronto lo veremos
```

Más funciones matemáticas Hay muchas más *funciones* matemáticas y algunas constantes extras definidas en el *módulo* `math`

```
In [7]: import math      # se importa el modulo para poder usar sus funciones
```

```
In [8]: math.sin(2*math.pi)
```

Out[8]: -2.4492935982947064e-16

```
In [9]: # round es una función built-in
round(5.6)
```

Out[9]: 6

```
In [10]: round(math.pi, 4)
```

```
Out[10]: 3.1416
```

```
In [11]: math.ceil(5.4)
```

```
Out[11]: 6
```

```
In [12]: math.trunc(5.8)
```

Out[12]: 5

```
In [14]: math.factorial(1e4)
```

```
In [15]: math.sqrt(-1)    # Epa!!
```

```
-----
ValueError                                Traceback (most recent call last)

<ipython-input-15-a8152a31b172> in <module>()
----> 1 math.sqrt(-1) # Epa!!

ValueError: math domain error
```

Pero existe un módulo equivalente para operaciones sobre el dominio complejo

```
In [22]: import cmath
        cmath.sqrt(-1)
```

```
Out[22]: 1j
```

Y también, sabiendo por propiedad de la potencia, podríamos directamente hacer:

```
In [23]: (-1)**0.5
```

```
Out[23]: (6.123233995736766e-17+1j)
```

2.0.2 Todo es un "objeto"

En Python todo es un *objeto*, es decir, una *instancia* de un clase o tipo de datos. Los objetos no solo *guardan* valores (atributos) sino que tienen acceso a *métodos*, es decir, traen acciones (funciones) que podemos ejecutar sobre esos valores, a veces requiriendo/permitiendo parámetros adicionales.

Jupyter/IPython facilita conocer todos los atributos y métodos de un objeto mediante **introspección**. Prueben escribir resultado. y apretar <TAB>. Por ejemplo:

```
In [16]: resultado = 1 + 2j
```

```
In [17]: R = 4.2
```

```
In [19]: R.is_integer()
```

```
Out[19]: False
```

Además del TAB, en una sesión interactiva de Jupyter, se puede obtener ayuda contextual para cualquier objeto (cualquier cosa!) haciendo Shift + TAB una o más veces, o agregando un signo de interrogación al final (blah?) y ejecutando

```
In [ ]:
```

En python "puro", estos comportamientos se logran con las funciones `dir()` y `help()`

Para conocer la clase/tipo de cualquier objeto se usa `type`

En muchos casos, se puede convertir explícitamente (o "castear") tipos de datos. En particular, entre números:

Ejercicios

- 1) Crear una variable llamada *magnitud* con un valor real. Definir otra variable compleja *intensidad* cuya parte real sea 1.5 veces *magnitud* y la parte imaginaria 0.3 veces *magnitud* + 1j. Encontrar la raíz cuadrada de *intensidad*.
- 2) Para calcular un **interés compuesto** se utiliza la fórmula

$$C_F = C_I(1 + r)^n$$

Donde:

- \$ C_F \$ es el capital al final del enésimo período
- \$ C_I \$ es el capital inicial
- \$ r \$ es la tasa de interés expresada en tanto por uno (v.g., 4 % = 0,04)
- \$ n \$ es el número de períodos

Codifique la fórmula en una celda y calcule el capital final para un depósito inicial de 10 mil pesos a una tasa del 1.5% mensual en 18 meses.

- 3) Investigue, a través de la ayuda interactiva, el parámetro opcional de `int` y las funciones `bin`, `oct` y `hex`. Basado en esto exprese en base 2 la operación `1 << 2` y en base hexadecimal `FA1 * 017`

2.0.3 Texto

Una cadena o *string* es una **secuencia** de caracteres (letras, números, símbolos). Python 3 utiliza el estándar **unicode**.

```
In [32]: print("Hola mundo!")
```

Hola mundo!

```
In [ ]: chinito = ""
```

```
In [ ]: type(chinito)
```

De paso, `unicode` se aplica a todo el lenguaje, de manera que el propio código puede usar caracteres "no ascii"

```
In [25]: años = 13
```

Las cadenas se pueden definir con apóstrofes, comillas, o triple comillas, de manera que es menos frecuente la necesidad de "escapar" caracteres

```
In [28]: calle = "O'Higgins"
         metáfora = 'Los "patitos" en fila'
```

Las triples comillas permiten crear cadenas multilínea


```
In [29]: V = """Me gustas cuando "callas"
          porque estás como ausente..."""
          V
```

```
Out[29]: 'Me gustas cuando "callas"\nporque estás como ausente...'
```

```
In [49]: print(poema)
```

```
Me gustas cuando "callas"
porque estás como ausente...
```

Las cadenas tienen sus propios **métodos**: pasar a mayúsculas, capitalizar, reemplazar una subcadena, etc.

```
In [55]: v = "hola amigos"
          v.capitalize()
```

```
Out[55]: 'Hola amigos'
```

Las cadenas se pueden concatenar

```
In [30]: a = " fue un soldado de San Martín"
          calle + a
```

```
Out[30]: "0'Higgings fue un soldado de San Martín"
```

y repetir

```
In [26]: "*" * 10
```

```
Out[26]: '*****'
```

Para separar una cadena se usa el método split

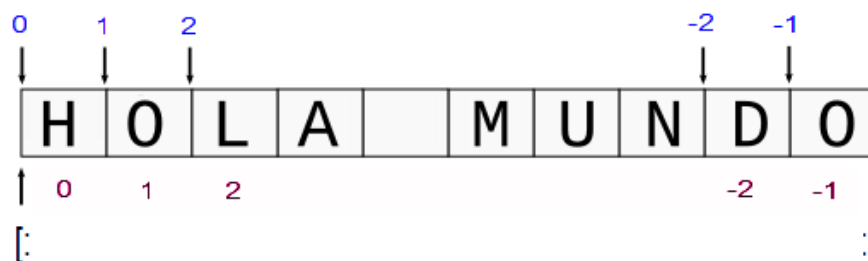
```
In [73]: a = "hola,amigos,como"
          a.split(',')
```

```
Out[73]: ['hola', 'amigos', 'como']
```

Y el método inverso es join, para unir muchas cadenas intercalandolas con otra

```
In [54]: " ".join(['y', 'jugando', 'al', 'amor', 'nos', 'encontró'])
```

```
Out[54]: 'y jugando al amor nos encontró'
```



Indizado y rebanado Las cadenas son **secuencias**. O sea, conjuntos ordenados que se pueden indizar, recortar, reordenar, etc.

```
In [57]: cadena = "HOLA MUNDO"
         cadena[0:4]
```

```
Out[57]: 'HOLA'
```

```
In [ ]:
```

```
In [87]: cadena[::-1]    #wow!
```

```
Out[87]: 'ODNUM ALOH'
```

```
In [81]: cadena[0:2]
```

```
Out[81]: 'HO'
```

El tipo str en python es **immutable**, lo que quiere decir que, una vez definido un objeto tipo cadena no podemos modificarlo.

```
In [90]: cadena[0] = 'B'
```

```
-----
TypeError                                Traceback (most recent call last)

<ipython-input-90-3dc4dfe33a69> in <module>()
----> 1 cadena[0] = 'B'

TypeError: 'str' object does not support item assignment
```

Pero si podemos basarnos en un string para **crear otro**

```
In [92]: 'B' + cadena[1:]
```

```
Out[92]: 'BOLA MUNDO'
```

```
In [91]: cadena = cadena.replace('H', 'B')
         cadena
```

```
Out[91]: 'BOLA MUNDO'
```

```
In [93]: cadena[:4] + " SUB" + cadena[5:]
```

```
Out[93]: 'BOLA SUBMUNDO'
```

longitud de una secuencia La función `len` (de *length*) devuelve la cantidad de elementos de cualquier secuencia

```
In [ ]: len(cadena)
```

interpolación Se puede crear un string a partir de una "plantilla" con un formato predeterminado. La forma más poderosa es a través del método `format`

```
In [37]: "{} es {}".format('Messi', 'crack')    # por posición, implícito
```

```
Out[37]: 'Messi es crack'
```

```
In [38]: "{1} es {0}".format('Messi', 'crack')    # por posición, explícito
```

```
Out[38]: 'crack es Messi'
```

```
In [39]: "{saludo} {planeta}".format(saludo='Hola', planeta='Mundo')    # por nombre de argument
```

```
Out[39]: 'Hola Mundo'
```

```
In [40]: "La parte real es {numero.real:.5f} y la imaginaria es {numero.imag} {numero}".format(
```

```
Out[40]: 'La parte real es 1.00000 y la imaginaria es 2.0 (1+2j)'
```

Casting de tipos Python es dinámico pero de **tipado es fuerte**. Quiere decir que no intenta adivinar y nos exige ser explícitos.

```
In [44]: "2" + "2"
```

```
Out[44]: '22'
```

```
In [ ]: int("2") + int("2")
```

```
In [ ]: float('2.34545') ** 2
```

Ejercicios

1. Dado un texto cualquiera, crear uno equivalente con "subrayado" con el caracter "=" en toda su longitud . Por ejemplo, "Beso a Beso", se debe imprimir por pantalla

```
=====
Beso a Beso
=====
```

2. Dada una cadena de palabras separadas por coma, generar otra cadena multilinea de "items" que comienzan por "* ". Por ejemplo "manzanas, naranjas, bananas" debe imprimir:

```
* manzanas
* naranjas
* bananas
```

2.0.4 Listas y tuplas: contenedores universales

```
In [47]: nombres = ["Melisa", "Nadia", "Daniel"]
```

```
In [48]: type(nombres)
```

```
Out[48]: list
```

Las listas tambien son secuencias, por lo que el indizado y rebanado funciona igual

```
In [49]: nombres[-1]
```

```
Out[49]: 'Daniel'
```

```
In [50]: nombres[-2:]
```

```
Out[50]: ['Nadia', 'Daniel']
```

Y pueden contener cualquier tipo de objetos

```
In [51]: mezclanza = [1.2, "Jairo", 12e6, calle, nombres[1]]
```

```
In [52]: print(mezcolanza)
```

```
[1.2, 'Jairo', 12000000.0, "0'Higgings", 'Nadia']
```

Hasta acá son iguales a las **tuplas**

```
In [53]: una_tupla = ("Martín", 1.2, (1j, nombres[0]))
         print(type(una_tupla))
         print(una_tupla[1:3])
         una_tupla
```

```
<class 'tuple'>
(1.2, (1j, 'Melisa'))
```

```
Out[53]: ('Martín', 1.2, (1j, 'Melisa'))
```

```
In [124]: list(una_tupla)
```

```
Out[124]: ['Martín', 1.2, (1j, 'Melisa')]
```

LA DIFERENCIA es que las **listas son mutables**. Es decir, es un objeto que puede cambiar: extenderse con otra secuencia, agregar o quitar elementos, cambiar un elemento o una porción por otra, reordenarse *in place*, etc.

```
In [119]: mezclanza.extend([1,2])
mezcol
```

```
In [120]: mezclanza
```

```
Out[120]: [1.2,
           'Jairo',
           12000000.0,
           "0'Higgings",
           'Nadia',
           'otro elemento',
           'otro elemento',
           'otro elemento',
           [1, 2],
           1,
           2]
```

```
In [ ]: una_tupla.append('a')
```

```
In [121]: mezclanza[0] = "otra cosa"
```

```
In [122]: mezclanza
```

```
Out[122]: ['otra cosa',
           'Jairo',
           12000000.0,
           "0'Higgings",
           'Nadia',
           'otro elemento',
           'otro elemento',
           'otro elemento',
           [1, 2],
           1,
           2]
```

Incluso se pueden hacer asignaciones de secuencias sobres "slices"

```
In [ ]: mezclanza[0:2] = ['A', 'B']      # notar que no hace falta que el valor tenga el mismo t
mezclanza
```

Como las tuplas son inmutables (como las cadenas), no podemos hacer asignaciones

```
In [123]: una_tupla [-1] = "osooo"
```

```
-----

TypeError                                Traceback (most recent call last)

<ipython-input-123-a4cb4abacc8c> in <module>()
----> 1 una_tupla[-1] = "osooo"

TypeError: 'tuple' object does not support item assignment
```

Las **tuplas** son mucho más eficientes (y seguras) si sólo vamos a **leer** elementos. Pero muchas operaciones son comunes.

```
In [ ]: l = [1, 3, 4, 1]
        t = (1, 3, 1, 4)
        print(l.count(3))
        print(t.count(3))
        l.append('8')
        l
```

packing/unpacking Como toda secuencia, las listas y tuplas se pueden *desempacar*

```
In [125]: nombre, nota = ("Juan", 10)
          print("{} se sacó un {}".format(nombre, nota))      # igual a "{0} se sacó un {1}"
```

Juan se sacó un 10

```
In [126]: a, b = 1, 2.0
          a, b = b, a
          print (a)
```

2.0

Python 3 permite un desempacado extendido

```
In [59]: a, b, *c = (1, 2, 3, 4, 5)      # c captura 3 elementos
          a, b, c
```

```
Out[59]: (1, 2, [3, 4, 5])
```

```
In [60]: a, *b, c = [1, 2, 3, 4, 5]      # b captura 3 elementos del medio
        print((a, b, c))
```

```
(1, [2, 3, 4], 5)
```

```
In [61]: # antes era más complicado
```

```
v = [1, 2, 3, 4, 5]
a, b, c = (v[0], v[1:-1], v[-1])
a, b, c
```

```
Out[61]: (1, [2, 3, 4], 5)
```

Como con los números, se pueden convertir las secuencias de un tipo de dato a otro. Por ejemplo:

```
In [ ]: a = (1, 3, 4)
        list(('A', 1))
```

Una función *builtin* muy útil es `range`

```
In [135]: list(range(3, 10, 2))
```

```
Out[135]: [3, 5, 7, 9]
```

```
In [138]: range(6)
```

```
Out[138]: range(0, 6)
```

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

```
In [ ]: range(0, 25, 5)
```

También hay una función estándar que da la sumatoria

```
In [64]: help(sum)
```

Help on built-in function sum in module builtins:

```
sum(iterable, start=0, /)
```

Return the sum of a 'start' value (default: 0) plus an iterable of numbers

When the iterable is empty, return the start value.

This function is intended specifically for use with numeric values and may reject non-numeric types.

```
In [63]: sum([1, 5.36, 5, 10])
```

```
Out[63]: 21.36
```

En toda secuencia tenemos el método `index`, que devuelve la posición en la que se encuentra un elemento

```
In [65]: a = [1, 'hola', []]
         a.index('hola')
```

```
Out[65]: 1
```

y como las listas son *mutables* también se pueden reordenar *in place* (no se devuelve un valor, se cambia internamente el objeto)

```
In [66]: a = [1, 2, 3]
         a.reverse()
```

```
In [147]: a
```

```
Out[147]: [3, 2, 1]
```

```
In [149]: list(reversed(a))
```

```
Out[149]: [1, 2, 3]
```

La forma alternativa es usando una función, que **devuelve** un valor

```
In [ ]: b = list(reversed(a))
         b
```

Nota: se fuerza la conversión de tipo con `list()` porque `reversed`, al igual que `range`, no devuelve estrictamente una lista. Ya veremos más sobre esto.

Una función útil es `zip()`, que agrupa elementos de distintas secuencias

```
In [67]: nombres = ['Juan', 'Martín', 'María']
         pasiones = ['cerveza', 'boca juniors', 'lechuga']
         nacionalidad = ('arg', 'arg', 'uru')
         list(zip(nombres, pasiones, nacionalidad))
```

```
Out[67]: [('Juan', 'cerveza', 'arg'),
          ('Martín', 'boca juniors', 'arg'),
          ('María', 'lechuga', 'uru')]
```

Ejercicios

1. Resuelva la siguiente operación

$$\frac{(\sum_{k=0}^{100} k)^3}{2}$$

2. Dada cualquier secuencia, devolver una tupla con sus elementos concatenados en a la misma secuencia en orden inverso. Por ejemplo para "ABCD" devuelve ('A', 'B', 'C', 'D', 'D', 'C', 'B', 'A')
3. Generar dos listas a partir de la función `range` de 10 elementos, la primera con los primeros múltiplos de 2 a partir de 0 y la segunda los primeros múltiplos de 3 a partir de 30 (inclusive). Devolver como una lista de tuplas [(0, 30), (2, 33), ...]

2.0.5 Estructuras de control de flujos

if/elif/else En todo lenguaje necesitamos controlar el flujo de una ejecución según una condición Verdadero/Falso (booleana). *Si (condicion) es verdadero hacé (bloque A); Sino hacé (Bloque B)*. En pseudo código:

```
Si (condicion):  
    bloque A  
sino:  
    bloque B
```

y en Python es muy parecido!

```
In [70]: edad = int(input('edad: '))  
        if edad < 18:  
  
            print("Hola pibe")  
        else:  
            print("Bienvenido señor")
```

```
edad: 12  
Hola pibe
```

```
In [159]:
```

```
Out[159]: True
```

Los operadores lógicos en Python son muy explicitos.

```
A == B  
A > B  
A < B  
A >= B  
A <= B  
A != B  
A in B
```

- A todos los podemos combinar con not, que niega la condición
- Podemos combinar condiciones con AND y OR, las funciones all y any y paréntesis

Podemos tener múltiples condiciones en una estructura. Se ejecutará el primer bloque cuya condición sea verdadera, o en su defecto el bloque else. Esto es equivalente a la sentencia switch o select case de otros lenguajes

```
In [71]: if edad < 12:  
        print("Feliz día del niño")  
        elif 13 < edad < 18:  
            print("Qué problema los granitos, no?")  
        elif edad in range(19, 90):  
            print("En mis épocas...")  
        else:  
            print("Y eso es todo amigos!")
```

Y eso es todo amigos!

```
In [162]: all([True, False, True])
```

```
Out[162]: False
```

En un `if`, la conversión a tipo *boolean* es implícita. El tipo `None` (vacío), el `0`, una secuencia (lista, tupla, string) (o conjunto o diccionario, que ya veremos) vacía siempre evalúa a `False`. Cualquier otro objeto evalúa a `True`.

```
In [163]: a = 5 - 5
```

```
if a:
    a = "No es cero"
else:
    a = "Dio cero"
print(a)
```

```
Dio cero
```

Para hacer asignaciones condicionales se puede usar la *estructura ternaria* del `if`: A si (condicion) sino B

```
In [165]: b = 5 - 6
          a = "No es cero" if b else "dio cero"
          print(a)
```

```
No es cero
```

Ejercicio dados **valores** numéricos para a , b y c , implementar la fórmula $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ donde a , b y c son los coeficientes de la ecuación $ax^2 + bx + c = 0$, para $a \neq 0$

For Otro control es **iterar** sobre una secuencia (o "*iterador*"). Obtener cada elemento para hacer algo. En Python se logra con la sentencia `for`

```
In [166]: sumatoria = 0
          for elemento in [1, 2, 3.6]:
              sumatoria = sumatoria + elemento
          sumatoria
```

```
Out[166]: 6.6
```

```
In [167]: list(enumerate(['a', 'b', 'c']))
```

```
Out[167]: [(0, 'a'), (1, 'b'), (2, 'c')]
```

Notar que no iteramos sobre el índice de cada elemento, sino sobre los elementos mismos. Basta de *i*, *j* y esas variables innecesarias! . Si por alguna razón son necesarias, tenemos la función `enumerate`

```
In [168]: for (posicion, valor) in enumerate([4, 3, 19]):  
           print("El valor de la posicion %s es %d" % (posicion, valor))
```

```
El valor de la posicion 0 es 4  
El valor de la posicion 1 es 3  
El valor de la posicion 2 es 19
```

```
In [169]: for i in range(10):  
           print(i)
```

```
0  
1  
2  
3  
4  
5  
6  
7  
8  
9
```

El bloque `for` se corre hasta el final del *iterador* o hasta encontrar una sentencia `break`

```
In [170]: sumatoria = 0  
           for elemento in range(1000):  
               if elemento > 100:  
                   break  
               sumatoria = sumatoria + elemento  
           sumatoria, elemento
```

```
Out[170]: (5050, 101)
```

También podemos usar `continue` para omitir la ejecución de "una iteración"

```
In [171]: sumatoria = 0  
           for elemento in range(20):  
               if elemento % 2:  
                   continue  
               print(elemento)  
               sumatoria = sumatoria + elemento  
           sumatoria
```

0
2
4
6
8
10
12
14
16
18

Out[171]: 90

Muchas veces queremos iterar una lista para obtener otra, con sus elementos modificados. Por ejemplo, obtener una lista con los cuadrados de los primeros 10 enteros.

```
In [172]: cuadrados = []
          for i in range(-3,15,1):
              cuadrados.append(i**2)
          print (cuadrados)
```

[9, 4, 1, 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196]

Una forma compacta y elegante (apythónica!) de escribir esta estructura muy frecuente son las **listas por comprensión**:

```
In [173]: [n*2 for n in range(5)]
```

Out[173]: [0, 2, 4, 6, 8]

Se lee: "Obtener el cuadrado de cada elemento i de la secuencia (rango 0 a 9)".

Pero además podemos filtrar: usar sólo los elementos que cumplen una condición.

```
In [81]: [i**2 for i in range(-2, 6) if i % 2 == 1]
```

Out[81]: [1, 1, 9, 25]

Ejercicios

- 1) Obtener la sumatoria de los cubos de los numeros impares menores a 100.

$$\sum_{a=0}^{100} a^3 \mid a \text{ impar}$$

- 2) Obtener la productoria de los primeros 12 digitos decimales de PI
- 3) Encuentre el mínimo de

$$f(x) = (x - 4)^2 - 3 \mid x \in [-100, 100]$$

- 4) Encuentre el promedio de los números reales de la cadena "3,4 1,2 -6 0 9,7"

```
In [ ]:
```

Expresiones generadores Al crear una lista por comprensión, se calculan todos los valores y se agregan uno a uno a la lista, que una vez completa se "devuelve" como un objeto nuevo.

Cuando no necesitamos todos los valores *al mismo tiempo*, porque por ejemplo podemos consumirlos de 1 en 1, es mejor crear *generadores*, que son tipos de datos **iterables pero no indizables** (es el mismo tipo de objeto que devuelve reversed, que ya vimos).

```
In [ ]: sum(a**2 for a in range(10))
```

While Otro tipo de sentencia de control es *while*: iterar mientras se cumpla una condición

```
In [180]: a = int(input('ingrese un numero'))
          while a < 10:
              print (a)
              a += 1
```

```
ingrese un numero3
3
4
5
6
7
8
9
```

Como en la iteración con for se puede utilizar la sentencia break para "romper" el bucle. Entonces puede modificarse para que la condición esté en una posición arbitraria

```
In [181]: n = 1
          while True:
              n = n + 1
              print('{} elefantes se balanceaban sobre la tela de una araña'.format(n))
              continuar = input('Desea invitar a otro elefante?')
              if continuar == 'no':
                  break
```

```
2 elefantes se balanceaban sobre la tela de una araña
Desea invitar a otro elefante?si
3 elefantes se balanceaban sobre la tela de una araña
Desea invitar a otro elefante?si
4 elefantes se balanceaban sobre la tela de una araña
Desea invitar a otro elefante?s
5 elefantes se balanceaban sobre la tela de una araña
Desea invitar a otro elefante?no
```

2.0.6 Diccionarios

Los diccionarios son otro tipo de estructuras de alto nivel que ya vienen incorporados. A diferencia de las secuencias, los valores **no están en una posición** sino bajo **una clave**: son asociaciones clave:valor

```
In [ ]: camisetas = {'Orión': 1, 'Carlitos': 10, 'Gago': 5, 'Gaitán': 'Jugador n.º 12'}
```

Accedemos al valor a través de una clave

```
In [ ]: camisetas['Perez'] = 8
```

```
In [ ]: camisetas
```

Las claves pueden ser cualquier objeto inmutable (cadenas, números, tuplas) y los valores pueden ser cualquier tipo de objeto. Las claves no se pueden repetir pero los valores sí.

Importante: los diccionarios **no tienen un orden definido**. Si por alguna razón necesitamos un orden, debemos obtener las claves, ordenarlas e iterar por esa secuencia de claves ordenadas.

```
In [ ]: sorted(camisetas.keys(), reverse=True)
```

Los diccionarios **son mutables**. Es decir, podemos cambiar el valor de una clave, agregar o quitar.

```
In [ ]: list(camisetas.items())
```

Hay muchos *métodos* útiles

```
In [ ]: for jugador, camiseta in camisetas.items():
    if jugador == 'Gaitán':
        continue
    print("%s lleva la %d" % (jugador, camiseta))
```

Se puede crear un diccionario a partir de tuplas (clave, valor) a través de la propia clase dict()

```
In [ ]: dict([('Yo', 'gaitan@gmail.com'), ('Melisa', 'mgomez@phasety.com'), ('Cismondi', 'cismondi@gmail.com')])
```

Que es muy útil usar con la función zip() que ya vimos

```
In [ ]: nombres = ("Martin", "Mariano")
        emails = ("tin@email.com", "nano@email.com")

        dict(zip(nombres, emails))
```

Ejercicio

1. Dados la lista de precios por kilo:

```
precios = {  
    "banana": 12,  
    "manzana": 8.5,  
    "naranja": 6,  
    "pera": 18  
}
```

Y la siguiente lista de compras

```
compras = {  
    "banana": 1,  
    "naranja": 3,  
    "pera": 0,  
    "manzana": 1  
}
```

Calcule el costo total de la compra.

1. Ejecute `import this` y luego analice el código del módulo con `this??` . ¿comprende el algoritmo? Cree el algoritmo inverso, es decir, codificador de [rot13](#)

```
In [73]: import this
```

The Zen of Python, by Tim Peters

```
Beautiful is better than ugly.  
Explicit is better than implicit.  
Simple is better than complex.  
Complex is better than complicated.  
Flat is better than nested.  
Sparse is better than dense.  
Readability counts.  
Special cases aren't special enough to break the rules.  
Although practicality beats purity.  
Errors should never pass silently.  
Unless explicitly silenced.  
In the face of ambiguity, refuse the temptation to guess.  
There should be one-- and preferably only one --obvious way to do it.  
Although that way may not be obvious at first unless you're Dutch.  
Now is better than never.  
Although never is often better than *right* now.  
If the implementation is hard to explain, it's a bad idea.  
If the implementation is easy to explain, it may be a good idea.  
Namespaces are one honking great idea -- let's do more of those!
```

```
In [74]: this??
```

2.0.7 Conjuntos

Los conjuntos (`set()` o `{}`) son grupos de elementos únicos. Al igual que los diccionarios, no están necesariamente ordenados

```
In [ ]: mamiferos = set(['perro', 'gato', 'leon'])
        domesticos = {'perro', 'gato', 'gallina'}
        aves = {'gallina', 'halcón', 'colibrí'}
```

```
In [ ]: mamiferos
```

Los conjuntos tienen métodos para cada una de las operaciones del [álgebra de conjuntos](#)

```
In [ ]: mamiferos.intersection(domesticos)    # mamiferos & domesticos
```

```
In [ ]: mamiferos.union(domesticos)           # mamiferos | domesticos
```

```
In [ ]: aves.difference(domesticos)           # mamiferos - domesticos
```

```
In [ ]: mamiferos.symmetric_difference(domesticos) # mamiferos ^ domesticos
```

Se puede comparar pertenencia de elementos y subconjuntos

```
In [ ]: 'gato' in mamiferos
```

```
In [ ]: domesticos.issubset(mamiferos)
```

Además, tienen métodos para agregar o extraer elementos

```
In [ ]: mamiferos.add('elefante')
        mamiferos
```

Por supuesto, se puede crear un conjunto a partir de cualquier iterador

```
In [ ]: set([1, 2, 3, 2, 1, 3])
```

Existe también una **versión inmutable** de los diccionarios, llamado `frozenset`

```
In [75]: frozenset
```

```
Out[75]: frozenset
```

Ejercicio La función `dir()` devuelve una lista con los nombre de todos los métodos y atributos de un objeto. Obtener una lista ordenada de los métodos en común entre `list`, `tuple` y `str` y los que son exclusivos para cada uno

```
In [ ]:
```


notebook2

June 11, 2017

1 Introducción a Python para ciencias e ingenierías (notebook 2)

Ing. Martín Gaitán

Links útiles

Repositorio del curso:

1.0.1 <http://bit.ly/cursopy>

Python "temporal" online:

1.0.2 <http://try.jupyter.org>

- Descarga de [Python "Anaconda"](#)
- Resumen de [sintaxis markdown](#)

1.1 Funciones

Hasta ahora hemos definido código en una celda: declaramos parámetros en variables, luego hacemos alguna operación e imprimimos y/o devolvemos un resultado.

Para generalizar esto podemos declarar **funciones**, de manera de que no sea necesario redefinir variables en el código para calcular/realizar nuestra operación con diferentes parámetros. En Python las funciones se definen con la sentencia `def` y con `return` se devuelve un valor

```
In [ ]: def cuadrado(numero):  
        """Dado un escalar, devuelve su potencia cuadrada"""  
        resulta = numero**2  
        return resulta
```

```
In [ ]: cuadrado(3)
```

```
In [ ]: cuadrado(2e10)
```

```
In [ ]: cuadrado(5-1j)
```

```
In [ ]: cuadrado(3 + 2)
```

notebook3

June 11, 2017

1 Introducción a Python para ciencias e ingenierías (notebook 3)

Ing. Martín Gaitán

Links útiles

Repositorio del curso:

1.0.1 <http://bit.ly/cursopy>

Python "temporal" online:

1.0.2 <http://try.jupyter.org>

- Descarga de [Python "Anaconda"](#)
- Resumen de [sintaxis markdown](#)

1.1 Programación Orientada a Objetos, una brevísima introducción

No vamos a entrar en detalles sobre qué es la famosísima "programación orientada a objetos". Simplemente es una buena manera de estructurar código para repetirse menos y "encapsular" datos (conservar estados) y sus comportamientos relacionados. Por ejemplo, cuando muchas funciones que reciben el mismo conjunto de parámetros, es un buen candidato a reescribirlo como una clase.

```
In [ ]: class Rectangulo:
    def __init__(self, base, altura):
        self.base = base
        self.altura = altura

    # el primer parámetro de cada metodo de la clase se llama self
    # y hace referencia "al propio" objeto, desde donde se puede consultar/modificar
    # el estado de sus atributos o llamar a otros métodos
    def area(self):
        return self.base * self.altura

    def perimetro(self):
        return (self.base + self.altura) * 2
```

```
In [22]: c = 1 + 2j
         c.conjugate()
```

```
Out[22]: (1-2j)
```

Listo, ya definimos una clase Rectangulo con la que podemos "fabricar" (instanciar) objetos este tipo, dando los parámetros que son requeridos por el método inicializador `__init__`. Se hace directamente "llamando a la clase"

```
In [7]: rectangulo2x4 = Rectangulo(2, 4)
```

```
In [8]: rectangulo2x4.base, rectangulo2x4.altura
```

```
Out[8]: (2, 4)
```

```
In [15]: rectangulo2x4.base = 2
         rectangulo2x4.area()
```

```
Out[15]: 8
```

```
In [23]: type(rectangulo2x4)
```

```
Out[23]: __main__.Rectangulo
```

Una vez definido la instancia podemos utilizar sus métodos.

```
In [17]: rectangulo2x4.area()
```

```
Out[17]: 8
```

```
In [18]: rectangulo2x4.perimetro()
```

```
Out[18]: 12
```

Siempre es bueno definir el método especial `__str__`, que es el que dice como 'castear' nuestro objeto a un string. De otra manera, la función `print` o la representación de salida interactiva no son muy útiles

```
In [24]: print(rectangulo2x4)
         rectangulo2x4
```

```
<__main__.Rectangulo object at 0x7f0efda4efd0>
```

```
Out[24]: <__main__.Rectangulo at 0x7f0efda4efd0>
```

Redefinamos la clase Rectángulo

```
In [35]: class Rectangulo:
    def __init__(self, base, altura):
        self.base = base
        self.altura = altura

    def area(self):
        return self.base * self.altura

    def perimetro(self):
        return (self.base + self.altura) * 2

    def volumen(self, profundidad):
        return self.area() * profundidad

    def __str__(self):
        return "{} de {} x {}".format(type(self).__name__, self.base, self.altura)

In [36]: r = Rectangulo(3, 4)
        print(r)
```

Rectangulo de 3 x 4

```
In [39]: r.volumen(1.5)
```

```
Out[39]: 18.0
```

1.1.1 Herencia

Un cuadrado es un tipo especial de rectángulo, donde $\text{base} == \text{altura}$. En vez de escribir una clase casi igual, hacemos una **herencia**

```
In [31]: class Cuadrado(Rectangulo): # cuadrado es una subclase de rectángulo.
    def __init__(self, lado):

        # igualito a papá rectángulo
        super(Cuadrado, self).__init__(base=lado, altura=lado)
        #self.base = lado
        #self.altura = lado
```

Lo que hicimos en este caso es hacer una "subclase" que hereda todo los métodos de la "clase padre" y le redefinimos el método inicializador

Una subclase no sólo puede redefinir el inicializador, sino cualquier método.

```
In [32]: cuadrado = Cuadrado(2)
        cuadrado.perimetro()
```

```
Out[32]: 8
```

```
In [33]: cuadrado.area()
```

```
Out[33]: 4
```

```
In [34]: print(cuadrado)
```

```
Cuadrado de 2 x 2
```

En python no sólo podemos reusar código "verticalmente" (herencia simple). Podemos incorporar "comportamientos" de multiples clases padre (herencia múltiple). Esto es lo que se conoce como el patrón "Mixin".

```
In [41]: class OVNI: # acá fue cuando me fumé
        def volar(self):
            print("Soy un %s y estoy volando!" % self)
            return 42
```

```
class CuadradoVolador(Cuadrado, OVNI):
    pass
```

```
In [42]: mi_cuadrado_volador = CuadradoVolador(4)
        mi_cuadrado_volador.area()
```

```
Out[42]: 16
```

```
In [43]: mi_cuadrado_volador.volar()
```

```
Soy un CuadradoVolador de 4 x 4 y estoy volando!
```

```
Out[43]: 42
```

Podríamos necesitar, por algun motivo esotérico digno de Capilla del Monte, un tipo especial CuadradoVolador que "vuele" de otra manera. Simplemente heredamos y reescribimos (o extendemos usando la función `super()`)

```
In [44]: class CuadradoVoladorSupersonico(CuadradoVolador):

        def volar(self): # esto se llama "sobrecargar método"
            valor = super(CuadradoVoladorSupersonico, self).volar()
            print("y estoy volando supersónicamente a {} metros!".format(valor))
```

```
        def volar_distinto(self):
            print('vuelo dando aletazos')
```

```
supersonico = CuadradoVoladorSupersonico(2)
```

```
In [45]: supersónico.volar()
```

Soy un CuadradoVoladorSupersonico de 2 x 2 y estoy volando!
y estoy volando supersónicamente a 42 metros!

```
In [46]: supersonico.volar_distinto()
```

vuelo dando aletazos

1.1.2 Ejercicios

1. Defina una clase `Estudiante` con atributos `nombre` y `año_ingreso`. Defina un método que devuelva la cantidad de años que lleva de cursado hasta el 2015. Luego redefina el método como `__len__` y utilice la función `len()` con la instancia.
2. Defina una clase `Linear` que recibe parámetros `a`, `b` y un método que calcula ecuación $ax + b$. Redefina ese método como el método especial `__call__` e intente llamar a la instancia como una función.
3. Herede de la clase `Linear` una subclase `Exponencial` que calcula ax^b .

1.2 Módulos y paquetes

Buenísimo estos *notebooks* pero ¿qué pasa si quiero reusar código?

Hay que crear **módulos**.

Por ejemplo, creemos un módulo para guardar la función que encuentra raíces de segundo grado.

Podemos abrir cualquier editor (incluido el que trae el propio jupyter), o alternativamente podemos preceder la celda con la "función magic" (que aporta Jupyter y se denotan por empezar con `%` o `%%`), en este caso `%%writefile`

El resultado es dejar en un archivo llamado `cuadratica.py` con el código de nuestra función en el mismo directorio donde tenemos el notebook (el archivo `.ipynb`)

```
In [60]: %%writefile cuadratica.py
```

```
def raices(a, b=0, c=0):
    """dados los coeficientes, encuentra los valores de x tal que  $ax^2 + bx + c = 0$ """

    discriminante = (b**2 - 4*a*c)**0.5
    x1 = (-b + discriminante)/(2*a)
    x2 = (-b - discriminante)/(2*a)
    return (x1, x2)
```

Overwriting `cuadratica.py`

Lo hemos guardado en un archivo `cuadratica.py` en el directorio donde estamos corriendo esta consola (notebook), entonces directamente podemos **importar** ese modulo.

```
In [61]: import cuadratica
```

El módulo `cuadratica` importado funciona como “*espacio de nombres*”, donde todos los objetos definidos dentro son atributos

```
In [ ]: cuadratica.raices
```

```
In [62]: cuadratica.raices(3, 2, -1)
```

```
Out[62]: (0.3333333333333333, -1.0)
```

```
In [ ]: cuadratica.raices(3, 2, 1)
```

Importar un módulo es importar un “espacio de nombres”, donde todo lo que el módulo contenga (funciones, clases, constantes, etc.) se accederá como un atributo del módulo de la forma `módulo.<objeto>`

Cuando el nombre del espacio de nombres es muy largo, podemos ponerle un alias

```
In [63]: import cuadratica as cuad    # igual que la primera forma pero poniendole un alias (mas
```

```
      cuad.raices(24,6,-5)
```

```
Out[63]: (0.3482423621500228, -0.5982423621500228)
```

Si **sólo queremos alguna unidad de código y no todo el módulo**, entonces podemos hacer una importación selectiva

```
In [64]: from cuadratica import raices    # sólo importa el "objeto" que indequemos y lo deja
                                             # en el espacio de nombres desde el que estamos importan
```

```
In [65]: raices?
```

```
In [ ]: %
```

Si, como sucede en general, el módulo definiera más de una unidad de código (una función, clase, constantes, etc.) podemos usar una tupla para importar varias cosas al espacio de nombres actual. Por ejemplo:

```
from cuadratica import raices, integral, diferencial
```

Por último, si queremos importar todo pero no usar el prefijo, podemos usar el `*`. **Esto no es recomendado**

```
from cuadratica import *
```

1.2.1 Ejercicios

1. Cree un módulo `circunferencia.py` que defina una constante `PI` y una función `area(r)` que calcula el área para una circunferencia de radio `r`.
2. Desde una celda de la sesión interactiva, importe todo el módulo como un alias `circle` y verifique `circle.PI` y `circle.area()`. Luego importe utilizando la forma `from circunferencia import ...` que importe también la función y la constante
3. verifique que `circle.area` y `area` son el mismo objeto

1.2.2 Paquetes: módulos de módulos

Cuando tenemos muchos módulos que están relacionados es bueno armar un **paquete**. Un paquete de módulos es un simple directorio con un módulo especial llamado `__init__.py` (que puede estar vacío) y tantos módulos y subpaquetes como queramos.

Los paquetes se usan igual que un módulo. Por ejemplo, supongamos que tenemos una estructura

```
paquete/  
  __init__.py  
  modulo.py
```

Puedo importar la función `funcion_loca` definida en `modulo.py` así

```
from paquete.modulo import funcion_loca
```

```
In [2]: %mkdir paquete          # creamos un directorio "paquete"
```

```
In [3]: %%writefile paquete/__init__.py
```

Writing `paquete/__init__.py`

```
In [6]: %%writefile paquete/modulo.py
```

```
def funcion_loca(w=300,h=200):  
    -                                     = (  
                                     255,  
                                     lambda  
                                     V      ,B,c  
    :c      and Y(V*V+B,B, c  
      -1)if(abs(V)<6)else  
    (      2+c-4*abs(V)**-.4)/i  
      ) ;v,      x=w,h; C = range(-1,v*x  
      +1);import struct; P = struct.pack;M, \  
      j =b'<IIHHHH',open('M.bmp','wb').write; k= v,x,1,24  
    for X in C or 'Mandelbrot. Adapted to Python3 by @tin_nqn_':  
      j(b'BM' + P(M, v*x*3+26, 26, 12,*k)) if X== -1 else 0; i,\  
      Y=_;j(P(b'BBB',*(lambda T: map(int, (T*80+T**9  
      *i-950*T **99,T*70-880*T**18 + 701*  
      T **9      ,T*i**(1-T**45*2))))(sum(  
      [      Y(0,(A%3/3.+X%v+(X/v+  
      A/3/3.-x/2)/1j)*2.5  
      /x      -2.7,i)**2 for \  
      A      in C  
      [:9]))  
      /9)  
      ) )
```


notebook4

June 11, 2017

1 Introducción a Python para ciencias e ingenierías (notebook 4)

Ing. Martín Gaitán

Repositorio del curso:

1.0.1 <http://bit.ly/cursopy>

Python "temporal" online:

1.0.2 <http://try.jupyter.org>

- Descarga de [Python "Anaconda"](#)
- Resumen de [sintaxis markdown](#)

1.1 Matplotlib, un gráfico vale más que mil palabras

Python es un lenguaje muy completo pero, aunque es muy grande, su librería estándar no es infinita. Por suerte hay [miles y miles de bibliotecas extras](#) para complementar casi cualquier aspecto en el que queramos aplicar Python. En algunos ámbitos, con soluciones muy destacadas.

Para hacer gráficos la opción canónica es Matplotlib <http://matplotlib.org/> . Ya viene instalado con la versión completa de Anaconda.

```
In [1]: %matplotlib inline
```

```
In [2]: from matplotlib import pyplot    # generalmente importado "as plt"
```

```
In [3]: x = [0.1*i for i in range(-50, 51)]
        x
```

```
Out[3]: [-5.0,
         -4.9,
         -4.8000000000000001,
         -4.7,
         -4.6000000000000005,
         -4.5,
         -4.4,
         -4.3,
         -4.2,
```

-4.1000000000000005,
-4.0,
-3.9000000000000004,
-3.8000000000000003,
-3.7,
-3.6,
-3.5,
-3.4000000000000004,
-3.3000000000000003,
-3.2,
-3.1,
-3.0,
-2.9000000000000004,
-2.8000000000000003,
-2.7,
-2.6,
-2.5,
-2.4000000000000004,
-2.3000000000000003,
-2.2,
-2.1,
-2.0,
-1.9000000000000001,
-1.8,
-1.7000000000000002,
-1.6,
-1.5,
-1.4000000000000001,
-1.3,
-1.2000000000000002,
-1.1,
-1.0,
-0.9,
-0.8,
-0.7000000000000001,
-0.6000000000000001,
-0.5,
-0.4,
-0.3000000000000004,
-0.2,
-0.1,
0.0,
0.1,
0.2,
0.3000000000000004,
0.4,
0.5,
0.6000000000000001,

```
0.7000000000000001,  
0.8,  
0.9,  
1.0,  
1.1,  
1.2000000000000002,  
1.3,  
1.4000000000000001,  
1.5,  
1.6,  
1.7000000000000002,  
1.8,  
1.9000000000000001,  
2.0,  
2.1,  
2.2,  
2.3000000000000003,  
2.4000000000000004,  
2.5,  
2.6,  
2.7,  
2.8000000000000003,  
2.9000000000000004,  
3.0,  
3.1,  
3.2,  
3.3000000000000003,  
3.4000000000000004,  
3.5,  
3.6,  
3.7,  
3.8000000000000003,  
3.9000000000000004,  
4.0,  
4.1000000000000005,  
4.2,  
4.3,  
4.4,  
4.5,  
4.6000000000000005,  
4.7,  
4.8000000000000001,  
4.9,  
5.0]
```

```
In [4]: y = [x_i**2 for x_i in x]  
y
```

```
Out[4]: [25.0,
```

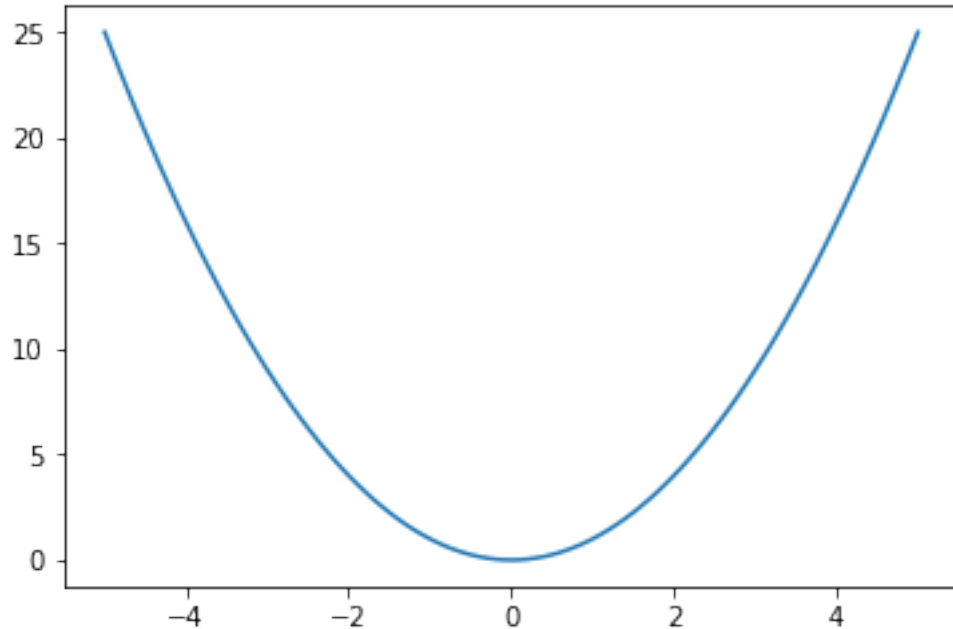
24.010000000000005,
23.040000000000006,
22.090000000000003,
21.160000000000004,
20.25,
19.360000000000003,
18.49,
17.64,
16.810000000000006,
16.0,
15.210000000000003,
14.440000000000001,
13.690000000000001,
12.96,
12.25,
11.560000000000002,
10.890000000000002,
10.240000000000002,
9.610000000000001,
9.0,
8.410000000000002,
7.840000000000002,
7.290000000000001,
6.760000000000001,
6.25,
5.760000000000002,
5.290000000000001,
4.840000000000001,
4.41,
4.0,
3.610000000000003,
3.24,
2.890000000000006,
2.560000000000005,
2.25,
1.960000000000004,
1.690000000000002,
1.440000000000004,
1.210000000000002,
1.0,
0.81,
0.640000000000001,
0.490000000000001,
0.360000000000001,
0.25,
0.160000000000003,
0.090000000000002,
0.040000000000001,

0.010000000000000002,
0.0,
0.010000000000000002,
0.040000000000000001,
0.090000000000000002,
0.160000000000000003,
0.25,
0.360000000000000001,
0.490000000000000001,
0.640000000000000001,
0.81,
1.0,
1.210000000000000002,
1.440000000000000004,
1.690000000000000002,
1.960000000000000004,
2.25,
2.560000000000000005,
2.890000000000000006,
3.24,
3.610000000000000003,
4.0,
4.41,
4.840000000000000001,
5.290000000000000001,
5.760000000000000002,
6.25,
6.760000000000000001,
7.290000000000000001,
7.840000000000000002,
8.410000000000000002,
9.0,
9.610000000000000001,
10.240000000000000002,
10.890000000000000002,
11.560000000000000002,
12.25,
12.96,
13.690000000000000001,
14.440000000000000001,
15.210000000000000003,
16.0,
16.810000000000000006,
17.64,
18.49,
19.360000000000000003,
20.25,
21.160000000000000004,

```
22.090000000000003,  
23.040000000000006,  
24.010000000000005,  
25.0]
```

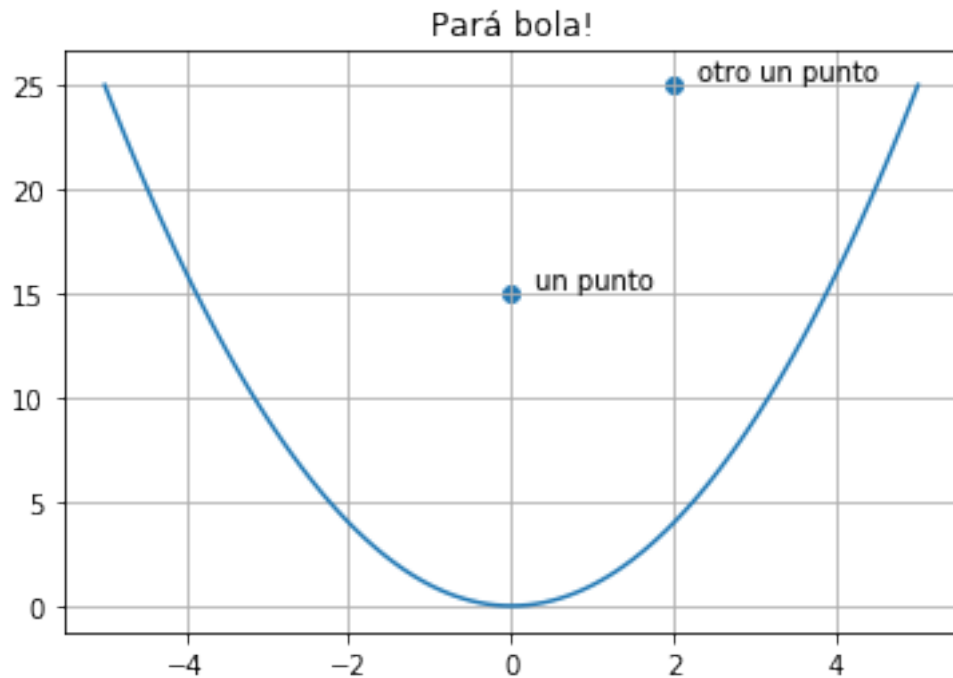
```
In [5]: pyplot.plot(x,y)
```

```
Out[5]: [<matplotlib.lines.Line2D at 0x7f8356575c50>]
```



Los gráficos emergentes son buenos porque tiene la barra de herramientas (interactividad) y podemos guardarlos en excelente calidad a golpe de mouse. Pero en los notebooks podemos poner los gráficos directamente incrustados

```
In [8]: pyplot.plot(x,y)  
        pyplot.title('Pará bola!')  
        pyplot.scatter([0, 2], [15, 25])  
        pyplot.annotate(s='un punto', xy=(0, 15), xytext=(0.3, 15.2))  
        pyplot.annotate(s='otro un punto', xy=(2, 25), xytext=(2.3, 25.2))  
        pyplot.grid()
```



Matplotlib sabe hacer muchísimos tipos de gráficos!

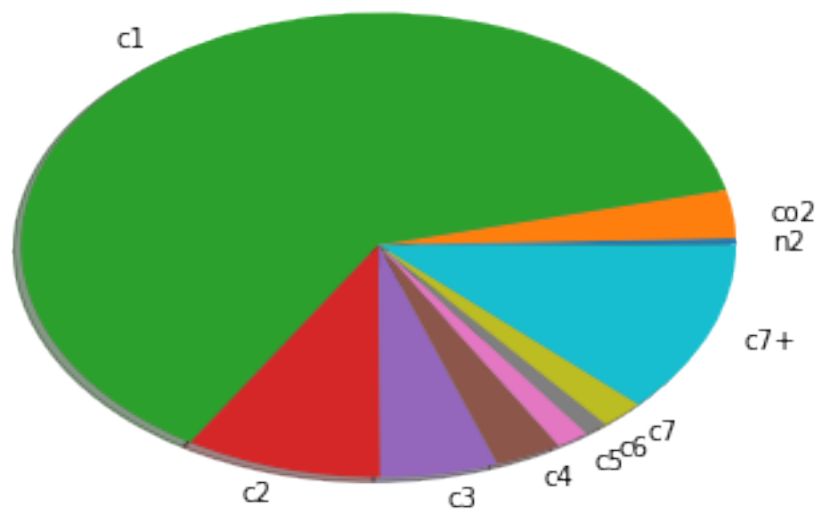
```
In [9]: !cat data/near_critical_oil.csv
```

```
Component,Mol fraction
n2,0.46
co2,3.36
c1,62.36
c2,8.9
c3,5.31
c4,3.01
c5,1.5
c6,1.05
c7,2.0
c7+,12.049
```

```
In [10]: import csv
```

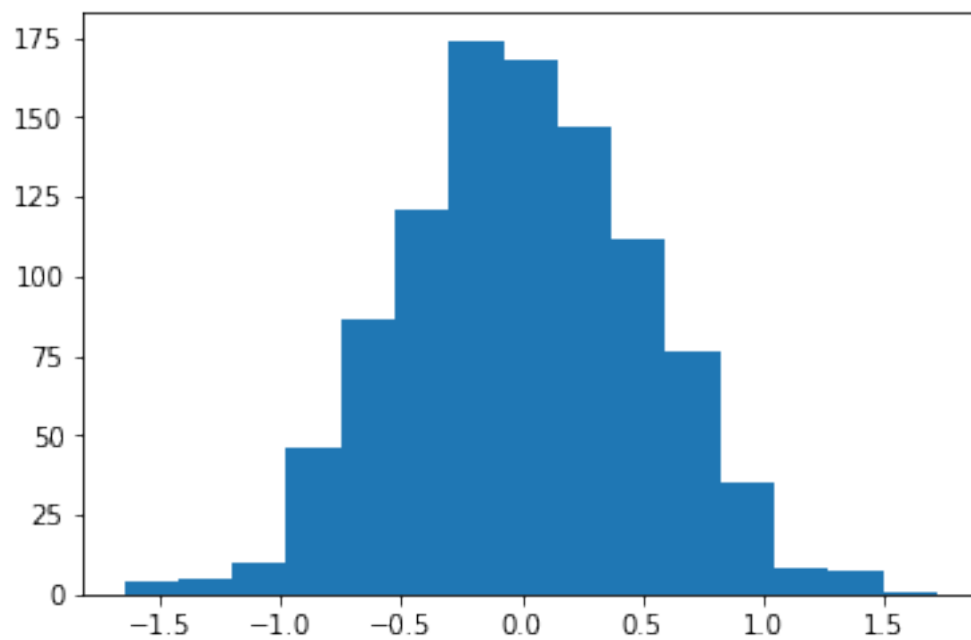
```
with open('data/near_critical_oil.csv') as csv_file:
    reader = csv.reader(csv_file)
    critical_oil = [line for line in reader]    #o list(reader)

components = [c for (c, f) in critical_oil[1:]]
fraction = [float(f) for (c, f) in critical_oil[1:]]
# el ; evita el output de la celda
pyplot.pie(fraction, labels=components, shadow=True);
```



```
In [11]: import random
          campana = [random.gauss(0, 0.5) for i in range(1000)]

In [12]: pyplot.hist(campana, bins=15);
```



1.1.1 La "papa" de matplotlib

Este es el algoritmo más importante para graficar con matplotlib

1. Ir a <http://matplotlib.org/gallery>
2. Elegir el gráfico de ejemplo que más se parezca a lo que queremos lograr
3. Copiar el código del ejemplo y adaptarlo a nuestros datos y gustos



Lo importante
no es saber,
sino tener el telefono
del que sabe.

```
In [13]: import numpy as np  
import matplotlib.pyplot as plt
```

```
N = 20  
theta = np.linspace(0.0, 2 * np.pi, N, endpoint=False)
```

```

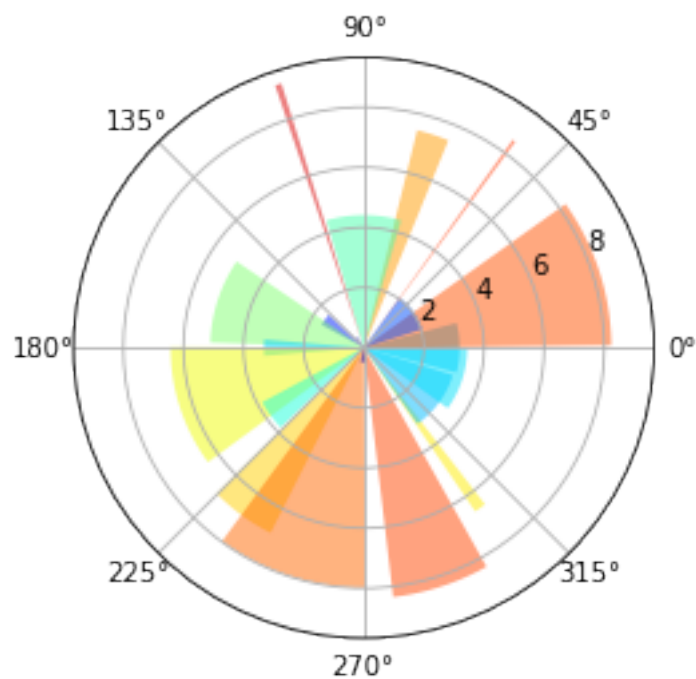
radii = 10 * np.random.rand(N)
width = np.pi / 4 * np.random.rand(N)

ax = plt.subplot(111, projection='polar')
bars = ax.bar(theta, radii, width=width, bottom=0.0)

# Use custom colors and opacity
for r, bar in zip(radii, bars):
    bar.set_facecolor(plt.cm.jet(r / 10.))
    bar.set_alpha(0.5)

plt.show()

```



1.1.2 Ejercicios

1. Dada la función para el cálculo de raíces de una ecuación de segundo grado [implementada](#) en clases anteriores, crear una función que dados los coeficientes grafique la parábola y denote las raíces con puntos rojos y el valor de X en cada una.
2. Basado en el [ejemplo de grafico de torta](#) de la galería, adaptar el ejemplo que grafica "near_critical_oil.csv" para que no se vea "ovoide" y la porción correspondiente a "C2" quede separada. Agregar un título al gráfico

Antes de seguir con Matplotlib debemos aprender el corazón del Python Científico: **Numpy**

1.2 Numpy, todo es un array

El paquete **numpy** es usado en casi todos los cálculos numéricos usando Python. Es un paquete que provee a Python de estructuras de datos vectoriales, matriciales y de rango mayor, de alto rendimiento. Está implementado en C y Fortran, de modo que cuando los cálculos son vectorizados (formulados con vectores y matrices), el rendimiento es muy bueno.

```
In [18]: import numpy as np
```

El pilar de numpy (y toda la computación científica basada en Python) es el tipo de datos `ndarray`, o sea arreglos de datos multidimensionales.

¿Otra secuencia más? ¿pero que tenían de malo las listas?

Las listas son geniales para guardar **cualquier tipo de objeto**, pero esa flexibilidad las vuelve ineficientes cuando lo que queremos es almacenar datos homogéneos

```
In [14]: %timeit [0.1*i for i in range(10000)]    # %timeit es otra magia de ipython
```

814 μ s \pm 12.2 μ s per loop (mean \pm std. dev. of 7 runs, 1000 loops each)

```
In [15]: %timeit np.arange(0, 1000, .1)    # arange es igual a range, pero soporta paso de tipo
```

38 μ s \pm 267 ns per loop (mean \pm std. dev. of 7 runs, 10000 loops each)

```
In [16]: %%timeit -o
          X = range(10000000)
          Y = range(10000000)
          Z = [(x + y) for x,y in zip(X,Y)]
```

944 ms \pm 35.6 ms per loop (mean \pm std. dev. of 7 runs, 1 loop each)

```
Out[16]: <TimeitResult : 944 ms  $\pm$  35.6 ms per loop (mean  $\pm$  std. dev. of 7 runs, 1 loop each)>
```

```
In [17]: %%timeit -o
          X = np.arange(10000000)
          Y = np.arange(10000000)
          Z = X + Y
```

141 ms \pm 1.14 ms per loop (mean \pm std. dev. of 7 runs, 1 loop each)

```
Out[17]: <TimeitResult : 141 ms  $\pm$  1.14 ms per loop (mean  $\pm$  std. dev. of 7 runs, 1 loop each)>
```

```
In [18]: __.best / _.best
```

```
Out[18]: 6.499575090405748
```

Existen varias formas para inicializar nuevos arrays de numpy, por ejemplo desde

- Listas o tuplas
- Usando funciones dedicadas a generar arreglos numpy, como `arange`, `linspace`, `ones`, `zeros` etc.
- Leyendo datos desde archivos

```
In [24]: v = np.array([1,2,3,4])
         v
```

```
Out[24]: array([1, 2, 3, 4])
```

```
In [32]: # una matriz: el argumento de la función array function es una lista anidada de Python
         M = np.array([[1, 2],
                       [3, 4.0]])
         M
```

```
Out[32]: array([[ 1.,  2.],
                [ 3.,  4.]])
```

```
In [26]: type(v), type(M)
```

```
Out[26]: (numpy.ndarray, numpy.ndarray)
```

1.2.1 Dimensiones, tamaño, tipo, forma

Los `ndarrays` tienen distintos atributos. Por ejemplo

```
In [27]: v.ndim, M.ndim    # cantidad de dimensiones
```

```
Out[27]: (1, 2)
```

```
In [28]: v.shape, M.shape # tupla de "forma". len(v.shape) == v.ndim
```

```
Out[28]: ((4,), (2, 2))
```

```
In [29]: v.size, M.size   # cantidad de elementos.
```

```
Out[29]: (4, 4)
```

```
In [30]: M.T    # transpuesta!
```

```
Out[30]: array([[1, 3],
                [2, 4.]])
```

A diferencia de las listas, los *arrays* también **tienen un tipo homogéneo**

```
In [33]: v.dtype, M.dtype    #
```

```
Out[33]: (dtype('int64'), dtype('float64'))
```

Se puede definir explícitamente el tipo de datos del array

```
In [36]: np.array([[1, 2], [3, 4]], dtype=complex)
```

```
-----  
TypeError                                Traceback (most recent call last)  
  
  <ipython-input-36-737d9df13f43> in <module>()  
----> 1 np.array([[1, 2], [3, 'f']], dtype=complex)  
  
TypeError: a float is required
```

Una gran ventaja del atributo `shape` es que podemos cambiarlo. Es decir, reacomodar la distribución de los elementos (por supuesto, sin perderlos en el camino)

```
In [38]: A = np.arange(0, 12)  
        A.shape
```

```
Out[38]: (12,)
```

```
In [42]: A.shape = 3, 4  
        A
```

```
Out[42]: array([[ 0,  1,  2,  3],  
                [ 4,  5,  6,  7],  
                [ 8,  9, 10, 11]])
```

El método `reshape` es otra manera de definir la forma de un array, generando uno nuevo array (a diferencia de `A.shape` que simplemente es otra vista del mismo array)

```
In [43]: A = np.arange(12).reshape((3,4))  
        A
```

```
Out[43]: array([[ 0,  1,  2,  3],  
                [ 4,  5,  6,  7],  
                [ 8,  9, 10, 11]])
```

1.2.2 Vistas

Esto es porque numpy en general no mueve los elementos de la memoria y en cambio usa **vistas** para mostrar los elementos de distinta forma. Es importante entender esto porque incluso los slicings son vistas.

```
In [48]: a = np.arange(10)  
        a
```

```
Out[48]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [49]: a = np.arange(10)
        b = a[::2] # todo de 2 en 2
        b
```

```
Out[49]: array([0, 2, 4, 6, 8])
```

```
In [50]: b[0] = 12
        a # chan!!!
```

```
Out[50]: array([12,  1,  2,  3,  4,  5,  6,  7,  8,  9])
```

En cambio

```
In [51]: c = np.arange(10)
        d = c[::2].copy()
        d[0] = 12
        c
```

```
Out[51]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Una forma de saber si un array es "base" o hereda los datos de otro array (es una vista), es verificar el atributo base

```
In [52]: b.base is a and a.base is None
```

```
Out[52]: True
```

1.2.3 Otras funciones constructoras de arrays

Además de arange hay otras funciones que devuelven arrays. Por ejemplo linspace, que a diferencia de arange no se da el tamaño del paso, sino la cantidad de puntos que queremos en el rango

```
In [53]: np.linspace(0, 2 * np.pi, 100) # por defecto, incluye el limite.
```

```
Out[53]: array([ 0.          ,  0.06346652,  0.12693304,  0.19039955,  0.25386607,
                0.31733259,  0.38079911,  0.44426563,  0.50773215,  0.57119866,
                0.63466518,  0.6981317 ,  0.76159822,  0.82506474,  0.88853126,
                0.95199777,  1.01546429,  1.07893081,  1.14239733,  1.20586385,
                1.26933037,  1.33279688,  1.3962634 ,  1.45972992,  1.52319644,
                1.58666296,  1.65012947,  1.71359599,  1.77706251,  1.84052903,
                1.90399555,  1.96746207,  2.03092858,  2.0943951 ,  2.15786162,
                2.22132814,  2.28479466,  2.34826118,  2.41172769,  2.47519421,
                2.53866073,  2.60212725,  2.66559377,  2.72906028,  2.7925268 ,
                2.85599332,  2.91945984,  2.98292636,  3.04639288,  3.10985939,
                3.17332591,  3.23679243,  3.30025895,  3.36372547,  3.42719199,
                3.4906585 ,  3.55412502,  3.61759154,  3.68105806,  3.74452458,
                3.8079911 ,  3.87145761,  3.93492413,  3.99839065,  4.06185717,
                4.12532369,  4.1887902 ,  4.25225672,  4.31572324,  4.37918976,
                4.44265628,  4.5061228 ,  4.56958931,  4.63305583,  4.69652235,
                4.75998887,  4.82345539,  4.88692191,  4.95038842,  5.01385494,
```

```
5.07732146, 5.14078798, 5.2042545 , 5.26772102, 5.33118753,
5.39465405, 5.45812057, 5.52158709, 5.58505361, 5.64852012,
5.71198664, 5.77545316, 5.83891968, 5.9023862 , 5.96585272,
6.02931923, 6.09278575, 6.15625227, 6.21971879, 6.28318531])
```

```
In [54]: _.size # en cualquier consola, python guarda el ultimo output en la variable _
```

```
Out[54]: 100
```

```
In [55]: matriz_de_ceros = np.zeros((4,6))
matriz_de_ceros
```

```
Out[55]: array([[ 0.,  0.,  0.,  0.,  0.,  0.],
 [ 0.,  0.,  0.,  0.,  0.,  0.],
 [ 0.,  0.,  0.,  0.,  0.,  0.],
 [ 0.,  0.,  0.,  0.,  0.,  0.]])
```

```
In [56]: np.ones((2, 4))
```

```
Out[56]: array([[ 1.,  1.,  1.,  1.],
 [ 1.,  1.,  1.,  1.]])
```

Pero numpy no sólo nos brinda los arrays. Los conceptos claves que aporta son *vectorización* y *broadcasting*

1.2.4 Vectorización y funciones universales

La **vectorización** define que las operaciones aritméticas entre arrays de igual forma se realizan implícitamente **elemento a elemento**, y por lo tanto hay una **ausencia de iteraciones explícitas y de indización**. La vectorización tiene muchas ventajas:

- El código vectorizado es más conciso y fácil de leer.
- Menos líneas de código habitualmente implican menos errores.
- El código se parece más a la notación matemática estándar (por lo que es más fácil, por lo general, corregir código asociado a construcciones matemáticas)
- La vectorización redonda en un código más "pythónico"

```
In [19]: import numpy as np
a = np.array([3, 4.3, 1])
b = np.array([-1, 0, 3.4])
c = a * b
c
```

```
Out[19]: array([-3. ,  0. ,  3.4])
```

Para dar soporte a la vectorización, numpy reimplementa funciones matemáticas como "funciones universales", que son aquellas que funcionan tanto para escalares como para arrays

```
In [20]: import math
math.sin(a)
```

notebook5

June 11, 2017

1 Introducción a Python para ciencias e ingenierías (notebook 5)

Ing. Martín Gaitán

Email: gaitan@gmail.com

Links útiles

Repositorio del curso:

1.0.1 <http://bit.ly/cursopy>

Python "temporal" online:

1.0.2 <http://try.jupyter.org>

- Descarga de [Python "Anaconda"](#)
- Resumen de [sintaxis markdown](#)

1.1 Una yapa de Numpy: Vectorizar funciones

Sabemos que Numpy opera vectorialmente, es decir, ejecutando la operación aritmetica o la "función universal" (ufunc) a cada uno de los elementos de los operadores.

```
In [1]: import numpy as np
        x = np.array([1, 2., 3.])
        y = np.array([.3, -5, 10])
```

También sabemos que cuando operamos un vector con un escalar se aplica el broadcasting

```
In [2]: x >= 2
```

```
Out[2]: array([False,  True,  True], dtype=bool)
```

pero no podemos hacer un "casting" automático de un array tipo booleano a un sólo booleano

```
In [3]: bool(x >= 2)
```

ValueError

Traceback (most recent call last)


```
<ipython-input-3-e301f242580e> in <module>()
----> 1 bool(x >= 2)
```

ValueError: The truth value of an array with more than one element is ambiguous. Use a.a

Ese "cast" se puede hacer con funciones de agregación booleanas, como all, allclose, any, etc. pero muchas veces eso implica cambiar código que ya tenemos escrito

```
In [4]: np.all(x >= 2)
```

```
Out[4]: False
```

Por ejemplo, veamos este par de funciones

```
In [5]: def func1(a):
        return a ** 2
```

```
def func2(a):
    """Return the same value if it's negative, otherwise returns its square"""

    if a < 0:
        return a
    else:
        return a**2
```

obviamente ambas funcionan con escalares,

```
In [6]: func1(10), func2(-4)
```

```
Out[6]: (100, -4)
```

Y func1 funciona también para arrays, debido a que la operación que hace (potencia) admite broadcasting.

```
In [7]: func1(np.arange(10))
```

```
Out[7]: array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])
```

Sin embargo, como el casting de un array de booleanos a un sólo booleano no es automático, func2 no es "universal"

```
In [8]: func2(np.arange(10))
```

```

-----

ValueError                                Traceback (most recent call last)

<ipython-input-8-9251847b0339> in <module>()
----> 1 func2(np.arange(10))

<ipython-input-5-3886a6ae5a87> in func2(a)
      6     """Return the same value if it's negative, otherwise returns its square"""
      7
----> 8     if a < 0:
      9         return a
     10     else:

```

ValueError: The truth value of an array with more than one element is ambiguous. Use a.a

Lo que necesitamos es **vectorizar la función**. `vectorize` transforma una función cualquiera en una de tipo universal: esto es, cuando un parámetro es un array y la aplicación directa no funciona, automáticamente se realiza una iteración implícita elemento a elemento.

```

In [ ]: v_func2 = np.vectorize(func2)
        v_func2(np.arange(10))    # para cada elemento de x, aplica func2

In [ ]: type(v_func2), v_func2.__doc__

```

Y sigue funcionando para escalares

```

In [ ]: v_func2(2)

```

1.1.1 Ejercicio

1. Defina como función universal de numpy la siguiente función definida por partes y grafique entre -5 y 5

$$y = f(x) = \begin{cases} x + 3, & \text{si } x < -1 \\ x^2 + 1, & \text{si } -1 \leq x \leq 2 \\ 5, & \text{si } x > 2 \end{cases}$$

```

In [ ]:

```

1.2 Scipy: parado sobre hombros de gigantes

La biblioteca Scipy se basa en los arrays multidimensionales de Numpy e implementa cientos de algoritmos científicos de alto nivel, clasificados en subpaquetes temáticos.

- Integración y ecuaciones diferenciales ([scipy.integrate](#))
- Optimización ([scipy.optimize](#))
- Interpolación ([scipy.interpolate](#))
- Transformada de Fourier ([scipy.fftpack](#))
- Procesamiento de señales ([scipy.signal](#))
- Álgebra lineal (mejor que `numpy.linalg`) ([scipy.linalg](#))
- Algoritmos optimizados para matrices ralas ([scipy.sparse](#))
- Estadística ([scipy.stats](#))
- Procesamiento de imágenes ([scipy.ndimage](#))
- Entrada/salida de archivos ([scipy.io](#))
- Funciones especiales ([scipy.special](#))

y hay más! Toda la documentación en <http://docs.scipy.org/doc/scipy/reference/>
 Veamos algunas aplicaciones

1.3 Integrales

La evaluación numérica de integrales de la forma

$$\int_a^b f(x)dx$$

se llaman *cuadratura numérica*, o simplemente *cuadratura*. SciPy tiene las funciones `quad`, `dblquad` y `tplquad` para cuadraturas simples, dobles o triples.

```
In [ ]: quad
```

```
In [1]: from scipy.integrate import quad
```

```
# definimos f f(x)
def f(x):
    return 2*x
```

```
In [ ]: val, abserr = quad(f, 0, 3)
        print("integral value =", val, ", absolute error =", abserr)
```

Ya que estamos definiendo "funciones objetivo", hay otra manera de definir funciones simples en python, que sólo si queremos las asignamos a un nombre. Por ejemplo

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

se podría definir así

```
In [ ]: # otro ejemplo de funcion lambda
        f = lambda x: x+2      # -> def f(x): return x**2
```

```
In [ ]: f(2)
```

Lo útil es que esta forma de crear **funciones anónimas** permite definir las directamente al pasarla como argumento

```
In [ ]: quad(lambda x: x**2, 0, 3)
```

Integración de funciones con parámetros adicionales Si necesitáramos pasarle parámetros a nuestra función lo hacemos en un tupla en el argumento args. Por ejemplo, supongamos que tenemos una función de la forma $g(x) = x^n$ donde x es la variable independiente y n es un parámetro.

```
In [ ]: def g(x, n):  
        return x**n
```

Si quisieramos evaluar la integral de esta función de x fijando n en 3

```
In [ ]: # evalua la integral x**3  
quad(g, 0, 5, args=(3,))
```

```
In [ ]: from scipy import integrate
```

```
In [ ]: integrate.dblquad
```

1.3.1 Ecuaciones diferenciales ordinarias

El paquete de integración también tiene funciones para resolver [ecuaciones diferenciales ordinarias](#).

Por ejemplo, para resolver

$$\frac{dy}{dt} = -2y$$

con $t \in [-1, 5]$ y la condición inicial $y(0) = 1$

```
In [ ]: %matplotlib inline  
  
from matplotlib import pyplot as plt  
import numpy as np  
  
from scipy.integrate import odeint  
  
t = np.linspace(-1, 5, 100)  
y = odeint(lambda y,t: -2*y, 1, t)  
plt.plot(t,y)
```

Como vemos, [el resultado es correcto](#)

1.3.2 Ejercicios

1. Dado el ejemplo de [gráfico "fill"](#) calcular el área de toda la superficie sombreada en rojo

```
In [ ]: # otra manera: directamente aplicar el método de trapezoides (teniendo datos y no la fun  
np.trapz(np.abs(y), x)
```

1.4 Optimización y ajuste de curvas

1.4.1 encontrar mínimos

Supongamos que tenemos una función cualquiera de una o más variables

```
In [ ]: def fpol(x):  
        return x**4 + 5*x**3 + (x-2)**2
```

```
In [ ]: fig, ax = plt.subplots()  
        x = np.linspace(-5, 2, 100)  
        ax.plot(x, fpol(x));
```

Podemos encontrar un mínimo local con la función `minimize`, que recibe como parámetro la función objetivo y un *valor de estimación inicial* a partir del cual se comienza a "ajustar"

```
In [ ]: from scipy import optimize  
        result = optimize.minimize(fpol, -3, method='BFGS') # http://en.wikipedia.org/wiki/BFGS  
        result
```

```
In [ ]: result['x']
```

`minimize` es una "función paraguas" para diferentes algoritmos, que se pueden elegir mediante el parámetro `method` o bien utilizar la función explícita de minimización que lo implementa.

```
In [ ]: optimize.minimize(fpol, 0, method='hBFGS')['x']
```

Ojo que hay métodos que son muy sensibles al valor de estimación

```
In [ ]: optimize.minimize(fpol, 0, method='Powell')['x']
```

Ejercicio

1. Encuentre el mínimo y el máximo de la función

$$f(x, y) = xe^{-x^2-y^2}$$

- 2- Basado en los ejemplos de gráficos de [superficie](#), y [scatter 3d](#) grafique la superficie para $x, y \in [(-2, 2), (-2, 2)]$ con un punto triangular en los puntos encontrados anteriormente

```
In [ ]: # %load https://gist.githubusercontent.com/mgaitan/faa930a5874311a6f888/raw/ec7ff8268935  
        %matplotlib inline  
  
        import numpy as np  
        from matplotlib import pyplot as plt  
        from mpl_toolkits.mplot3d import Axes3D  
        from matplotlib import cm  
  
        x = y = np.linspace(2, -2, 100)  
        xx,yy = np.meshgrid(x, y)
```

```

f = lambda var: var[0]*np.exp(-var[1]**2 - var[0]**2)

z = f([xx,yy])

fig = plt.figure()
fig.set_size_inches(10,10)
ax = fig.gca(projection='3d')
ax.plot_surface(xx,yy,z, rstride=1, cstride=1, cmap=cm.coolwarm,
                linewidth=0)

In [ ]: # ----
        from scipy import optimize
        minimum = optimize.minimize(f, [-1, 1])
        maximum = optimize.minimize(lambda x: -f(x), [1, 1])
        minimum, maximum

In [ ]: # ----

        ax.scatter(minimum['x'][0], minimum['x'][1], minimum['fun'], marker='v', s=200)
        ax.scatter(maximum['x'][0], maximum['x'][1], f(maximum['x']), marker='^', s=200)
        fig

```

1.4.2 Curve fitting

Otra tarea frecuente es ajustar parámetros de una función objetivo a partir de un *dataset* de puntos conocidos.

Supongamos que tenemos una serie de datos medidos

```

In [ ]: dataset = np.array([[ 0.          ,  3.07127661],
                             [ 0.08163265,  2.55730445],
                             [ 0.16326531,  2.28438915],
                             [ 0.24489796,  1.91475822],
                             [ 0.32653061,  2.00380351],
                             [ 0.40816327,  1.89419135],
                             [ 0.48979592,  1.74713349],
                             [ 0.57142857,  1.68237822],
                             [ 0.65306122,  1.44749977],
                             [ 0.73469388,  1.67511522],
                             [ 0.81632653,  1.34023054],
                             [ 0.89795918,  1.1209472 ],
                             [ 0.97959184,  1.41692478],
                             [ 1.06122449,  0.88480583],
                             [ 1.14285714,  0.9939094 ],
                             [ 1.2244898 ,  1.02293629],
                             [ 1.30612245,  1.11983417],
                             [ 1.3877551 ,  0.77520734],
                             [ 1.46938776,  0.88371884],
                             [ 1.55102041,  1.24492445],

```

```

[ 1.63265306,  0.8275613 ],
[ 1.71428571,  0.60846983],
[ 1.79591837,  0.73019407],
[ 1.87755102,  0.75139707],
[ 1.95918367,  0.6496137 ],
[ 2.04081633,  0.59122461],
[ 2.12244898,  0.61734269],
[ 2.20408163,  0.61890166],
[ 2.28571429,  0.68647436],
[ 2.36734694,  0.47551378],
[ 2.44897959,  0.89850013],
[ 2.53061224,  0.53029377],
[ 2.6122449 ,  0.74853936],
[ 2.69387755,  0.371923  ],
[ 2.7755102 ,  0.43536233],
[ 2.85714286,  0.40515777],
[ 2.93877551,  0.52171142],
[ 3.02040816,  0.53069869],
[ 3.10204082,  0.71363042],
[ 3.18367347,  0.54962316],
[ 3.26530612,  0.7133841 ],
[ 3.34693878,  0.27204244],
[ 3.42857143,  0.56572211],
[ 3.51020408,  0.29310287],
[ 3.59183673,  0.50044492],
[ 3.67346939,  0.60938301],
[ 3.75510204,  0.55696286],
[ 3.83673469,  0.59409416],
[ 3.91836735,  0.30335525],
[ 4.         ,  0.24230362]]
)

```

```
In [ ]: x, y = dataset[:,0], dataset[:,1]
```

```

fig = plt.figure(figsize=(9,6))
ax = fig.add_subplot(1, 1, 1)
ax.scatter(dataset[:,0], dataset[:,1], marker='^');

```

Vemos que tiene forma de exponencial decreciente, es decir

$$f(x) = ae^{-bx} + c$$

El objetivo es encontrar cuales son los valores de los parámetros a, b y c que mejor ajustan la curva a los puntos dados.

```
In [ ]: def func(x, a, b, c):    # x es la variable independiente, a b y c los parametros a encontrar
    return a*np.exp(-b*x) + c
```

```
popt, pcov = optimize.curve_fit(func, x, y)
popt
```

Por lo tanto la curva que mejor ajusta a nuestros datos es

$$f(x) = 2.33e^{-1.18x} + 0.25$$

```
In [ ]: ax.plot(x, func(x, *popt), 'r', label='fitted')
fig
```

1.4.3 Procesamiento de imágenes

Una imagen digital tipo **raster** (matriz de pixeles) puede ser facilmente interpretable como un array multidimensional. Por ejemplo, una imagen de 512x512 pixeles a color puede almacenarse en una "matriz" de 3x512x512 donde cada una las 3 matrices representa una capa de color RGB (rojo, verde, azul)

Para una imagen monocromática, obviamente una sola matriz nos alcanza.

Scipy trae el array de una **imagen clásica** (fragmento de una foto de Playboy de 1972) para ejemplos de procesamiento de imágenes, en escala de grises.

```
In [ ]: from scipy.misc import lena
        image = lena()
        image.shape,

In [ ]: %matplotlib inline
        from matplotlib import pyplot as plt
        plt.gray()
        plt.imshow(image)
```

Podemos procesar imágenes simplemente aplicando máscaras

```
In [ ]: mascara = (image > image.max() * .75)
        plt.imshow(mascara)    # blanco == True
```

Podemos aplicar la máscara a la imagen original, por ejemplo,

```
In [ ]: plt.imshow(np.where(mascara, image, 255))

In [ ]: mask = np.tril(np.ones(image.shape)) #tril / triu seleccionan triangulos inf y sup
        plt.imshow(np.where(mask, image, 255))
```

Scipy trae un paquete para manipulación de **imágenes multidimensionales**

```
In [ ]: from scipy import ndimage

In [ ]: plt.imshow(ndimage.convolve(image, np.ones((12,12))))
```


1.4.4 Transformaciones

```
In [ ]: lena = image
        shifted_lena = ndimage.shift(lena, (50, 50))
        shifted_lena2 = ndimage.shift(lena, (50, 50), mode='nearest')
        rotated_lena = ndimage.rotate(lena, 30)
        cropped_lena = lena[50:-50, 50:-50]
        zoomed_lena = ndimage.zoom(lena, 5)
        imgs = (shifted_lena, shifted_lena2, rotated_lena, cropped_lena, zoomed_lena)
        fig, axes = plt.subplots(1, 5, figsize=(20,100))
        for ax, img in zip(axes, imgs):
            ax.tick_params(left=False, bottom=False, labelleft=False)
            ax.imshow(img)
```

1.4.5 Filtros

```
In [ ]: from scipy import signal

        noisy_lena = np.copy(lena).astype(np.float)
        noisy_lena += lena.std()*0.5*np.random.standard_normal(lena.shape)
        blurred_lena = ndimage.gaussian_filter(noisy_lena, sigma=3)
        median_lena = ndimage.median_filter(blurred_lena, size=5)
        wiener_lena = signal.wiener(blurred_lena, (5,5))
        imgs = (noisy_lena, blurred_lena, median_lena, wiener_lena)
        fig, axes = plt.subplots(1, 4, figsize=(20,80))
        for ax, img in zip(axes, imgs):
            ax.tick_params(left=False, bottom=False, labelleft=False)
            ax.imshow(img)
```

Si les interesa esta área deberían ver los paquetes especializados [Scikit-image](#), [Pillow](#) y [OpenCV](#)

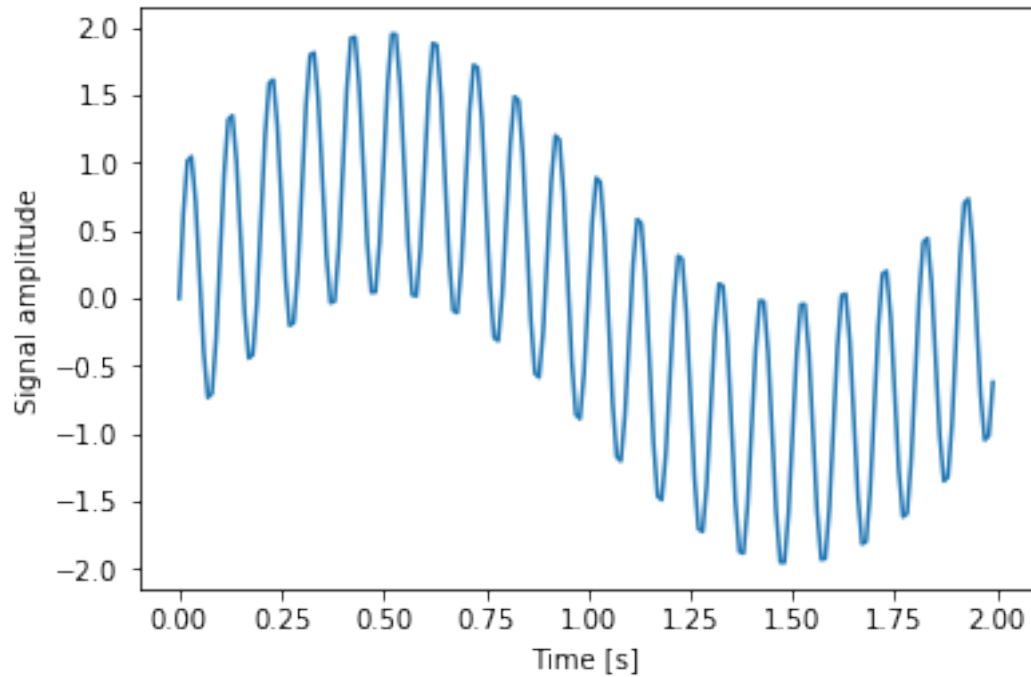
1.5 Transformada de Fourier

```
In [65]: %matplotlib inline

        f = 10      # Frequency, in cycles per second, or Hertz
        f_s = 100   # Sampling rate, or number of measurements per second

        t = np.linspace(0, 2, 2 * f_s, endpoint=False)
        x = np.sin(f * 2 * np.pi * t) + np.sin(np.pi * t)

        fig, ax = plt.subplots()
        ax.plot(t, x)
        ax.set_xlabel('Time [s]')
        ax.set_ylabel('Signal amplitude');
```



```
In [67]: from scipy import fftpack
```

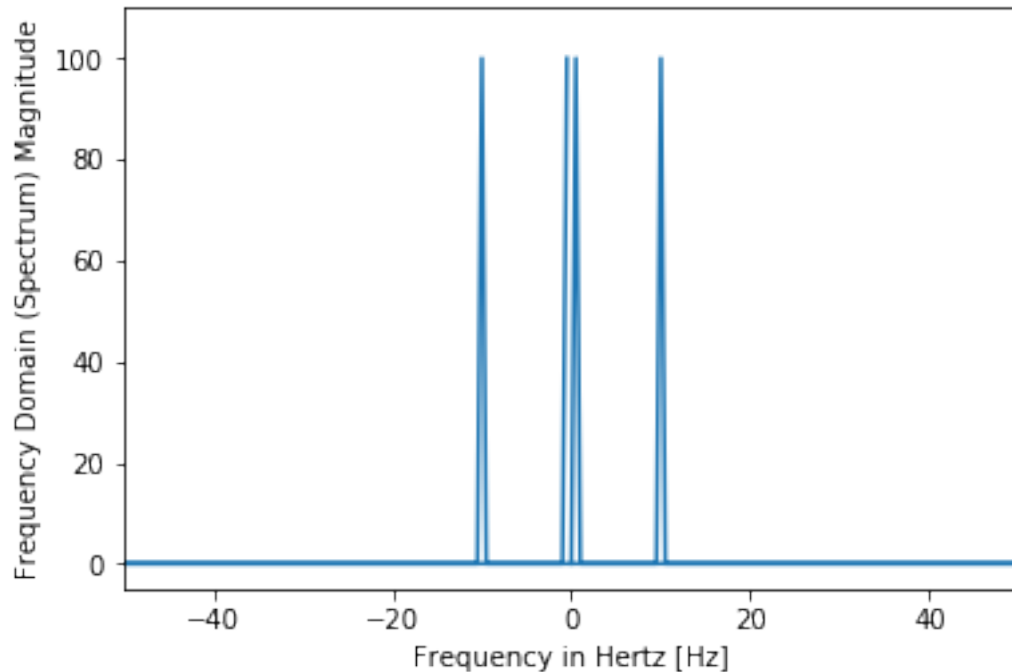
```
X = fftpack.fft(x)
freqs = fftpack.fftfreq(len(x)) * f_s

fig, ax = plt.subplots()

ax.plot(freqs, np.abs(X))
ax.set_xlabel('Frequency in Hertz [Hz]')
ax.set_ylabel('Frequency Domain (Spectrum) Magnitude')
ax.set_xlim(-f_s / 2, f_s / 2)
ax.set_ylim(-5, 110)
```

```
Out[67]:
```

```
(-5, 110)
```



2 A now, something completely different: SymPy!

([intertextualidad :D](#)) De paso, sabías que el Python se llama así porque el holandés que lo creo es fanático de Monty Python)

SymPy es una biblioteca Python para matemática simbólica, similar a software como Mathematica o Mathcad. La documentación se encuentra en <http://sympy.org/>.

En Anaconda (full) ya viene instalado, o se puede instalar via pip o conda con

```
$ pip install sympy
```

2.0.1 ¿que es la computación simbólica?

Comunmente, las computadoras usan una [coma flotante](#) para **representar** numeros reales (y complejos) y calcular operaciones matemáticas. Esto implica que la precisión es limitada

```
In [11]: import math
         math.sqrt(8)
```

```
Out[11]: 2.8284271247461903
```

Los sistemas de cálculo simbólico (o álgebra computacional, CAS), utilizan una **representación simbólica** (presentada de la manera más simplicifada posible)

```
In [12]: import sympy
```

```
raiz8 = sympy.sqrt(8)
raiz8
```

```
Out[12]: 2*sqrt(2)
```

Podemos, por supuesto, **evaluar** la expresión simbólica

```
In [13]: raiz8.evalf()
```

```
Out[13]: 2.82842712474619
```

```
In [14]: raiz8.evalf(n=150)
```

```
Out[14]: 2.8284271247461900976033774484193961571393437507538961463533594759814649569242140777007
```

Por defecto la evaluación numérica se lleva a cabo con una precisión de 15 decimales. Pero puede ajustarse la precisión al número de decimales que uno desee, enviando el número de decimales como argumento a `evalf()` o a la función `N()`

SymPy utiliza como background la biblioteca [mpmath](#) que le permite realizar cálculos con aritmética de **precisión arbitraria**, de forma tal que ciertas constantes especiales, como π , el número e , ∞ (infinito), son tratadas como símbolos y pueden ser evaluadas con aritmética de alta precisión:

2.0.2 Salida enriquecida

Para continuar, haremos que el "output matemático" se vea más bonito

```
In [15]: import sympy
         from sympy.interactive import printing
         printing.init_printing(use_latex='mathjax')
```

Esa función inicializa el output en el mejor modo disponible en el entorno. Como los notebooks saben mostrar LaTeX, lo hará así, mostrando *outputs* muy bonitos.

```
In [16]: sympy.sqrt(8)
```

```
Out[16]:
```

$$2\sqrt{2}$$

2.0.3 SymPy como una calculadora

SymPy tiene tres tipos de datos predefinidos: el real (Real), el racional (Rational) y el entero (Integer). El tipo Rational representa a un número racional como un par de números enteros: el numerador y el denominador. Por ejemplo: `Rational(1, 2)` representa la fracción $1/2$, `Rational(5, 3)` a $5/3$, etc.

```
In [17]: a = sympy.Rational(1, 2)
         a
```

Out[17]:

$$\frac{1}{2}$$

In [18]: a*2

Out[18]:

$$1$$

In [19]: sympy.pi**2

Out[19]:

$$\pi^2$$

In [20]: sympy.pi.evalf(n=200)

Out[20]:

3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211706

In [21]: (sympy.pi+sympy.exp(1)).evalf()

Out[21]:

5.85987448204884

También existe una clase para representar al infinito matemático, llamada oo:

In [22]: sympy.oo > 99999

Out[22]:

True

In [23]: sympy.oo + 1

Out[23]:

$$\infty$$

In [24]: 1 / sympy.oo

Out[24]:

$$0$$

otra manera de ejecutar evalf de una expresión es con la función sympy.N

In [25]: sympy.N(sympy.sqrt(2), 100)

Out[25]:

1.41421356237309504880168872420969807856967187537694807317667973799073247846210703885038753432764157

Ejercicios:

1. Calcular $1/2 + 1/3$ con aritmética racional.
2. Calcular 2^e con 200 decimales

2.0.4 Símbolos

El alma del Cálculo Simbólico son, naturalmente, las variables simbólicas, que en SymPy son instancias de la clase `Symbol`. Una explicación intuitiva es que, mientras que las variables ordinarias tienen un valor que puede ser un número, una cadena, un valor verdadero / falso, una secuencia, etc. las variables simbólicas juegan el papel de "contenedores": no sabemos a priori lo que pueden ser

```
In [26]: x = sympy.Symbol('x')
        y = sympy.Symbol('y')
        n = sympy.Symbol('n', integer=True)
```

Luego, pueden ser manipuladas:

```
In [27]: x + y + x - y
```

Out[27]:

$$2x$$

```
In [28]: (x + y)**2
```

Out[28]:

$$(x + y)^2$$

Una manera más compacta de definir variables simbólicas es con la función `symbols`, que permite pasar una cadena con muchas variables a generar separadas por espacios.

```
In [29]: a, b = sympy.symbols('a b')
        (a - b)**2
```

Out[29]:

$$(a - b)^2$$

2.0.5 Traducir expresiones a otros lenguajes

Podemos aprovechar toda la funcionalidad de "impresión" que trae SymPy para traducir cualquier expresión a otros formatos.

```
In [30]: from sympy import sin, exp, cos

        formula = sin(x)*exp(cos(x)**x)/x
        print(sympy.latex(formula, mode='equation', itex=True))
```

$$\frac{1}{x} e^{\cos(x)} \sin(x)$$

Sólo nos queda copiar y pegar esa expresión en markdown

$$\frac{1}{x} e^{\cos(x)} \sin(x)$$

```
In [31]: sympy.ccode(formula)    # código C ansi
```

```
Out[31]: 'exp(pow(cos(x), x))*sin(x)/x'
```

```
In [32]: sympy.fcode(formula)    # código fortran iso
```

```
Out[32]: '      exp(cos(x)**x)*sin(x)/x'
```

2.0.6 Manipulación algebraica

SymPy tiene una gran potencia para realizar cálculos en forma algebraica.

```
In [33]: sympy.expand((x+y)**2, deep=True)
```

```
Out[33]:
```

$$x^2 + 2xy + y^2$$

también se pueden hacer especificaciones adicionales:

```
In [34]: sympy.expand(x + y, complex=True)
```

```
Out[34]:
```

$$\Re x + \Re y + i\Im x + i\Im y$$

A veces es útil darle "pistas" al sistema para que sepa por qué "camino matemático" andar

```
In [35]: sympy.expand(sympy.cos(x + y))
```

```
Out[35]:
```

$$\cos(x + y)$$

```
In [36]: sympy.expand(sympy.cos(x + y), trig=True)
```

```
Out[36]:
```

$$-\sin(x) \sin(y) + \cos(x) \cos(y)$$

2.0.7 Simplificación

```
In [ ]: sympy.simplify(x**2 + 2*x*y + y**2)
```

La simplificación se realiza de la mejor manera que SymPy encuentra y a veces la respuesta puede no ser lo que uno espera (demasiado trivial o no en la forma que se necesita). Las pistas la estrategia de simplificación que debe utilizar:

- `powsimp`: simplifica exponentes.
- `trigsimp`: simplifica expresiones trigonométricas
- `logcombine`
- `randsimp`
- `together`

Ejercicios:

1. Calcular la forma expandida de $(x + y)^6$.
2. Simplificar la expresión trigonométrica $\frac{\sin(x)}{\cos(x)}$

2.1 Límites

Los límites se pueden calcular con mucha facilidad usando SymPy.

```
limit(f(x), x, 0)
```

```
In [ ]: from sympy import limit, sin, oo
        limit(sin(x)/x, x, 0), limit(sin(x)/x, x, oo)
```

2.2 Diferenciación

Se puede derivar cualquier expresión de SymPy

```
In [ ]: from sympy import diff, tan, exp, cos, sin

        diff(sin(x), x)
```

```
In [ ]: diff(exp(sin(x**3)**x), x)
```

Se pueden calcular derivadas de orden superior especificando el orden de derivación como tercer argumento de `diff`:

```
In [ ]: diff(sin(2*x), x, 2)
```

```
In [ ]: diff(sin(2*x), x, 3)
```

2.2.1 Expansión en serie de Taylor

SymPy puede expandir funciones en serie de Taylor mediante la función:

```
In [ ]: sympy.series(sympy.cos(x), x)
```


Ejercicios:

1. Calcular el límite de

$$\lim_{x \rightarrow 0} \operatorname{sen}(x) \frac{\exp(\cos(x)^x)}{x}$$

2. Calcular las tres primeras derivadas de

$$\log(x^2 - \tan(x))$$

2.2.2 Integración

SymPy es capaz de calcular integrales definidas e indefinidas para funciones elementales, trascendentes y especiales, mediante la herramienta `integrate()`.

Integración de funciones elementales:

```
In [ ]: from sympy import integrate, log
        integrate(6*x**5, x)
```

```
In [ ]: integrate(sin(x), x)
```

```
In [ ]: from sympy import sinh
        integrate(2*x + sinh(x), x)
```

Integración de funciones especiales:

```
In [ ]: integrate(exp(-x**2)*sympy.erf(x), x)
```

También es posible calcular integrales definidas. La función es la misma: `integrate(función, (variable, límite inferior, límite superior))`, sólo que como segundo argumento se utiliza una tupla cuyo primer elemento es la variable de integración, su segundo elemento es el límite inferior de integración y el último es el límite superior.

```
In [ ]: integrate(x**3, (x, -1, 1))
```

```
In [ ]: integrate(cos(x), (x, -sympy.pi/2, sympy.pi/2))
```

¡Incluso se pueden calcular integrales impropias!

```
In [ ]: integrate(exp(-x), (x, 0, sympy.oo))
```

```
In [ ]: integrate(exp(-x**2), (x, -sympy.oo, sympy.oo))
```

2.2.3 Resolución de ecuaciones

SymPy es capaz de resolver ecuaciones algebraicas de una y de varias variables. Para eso existe la función `solve(ecuación, variable)`, donde ecuación es una expresión que es igual a cero.

Por ejemplo, para resolver la ecuación $x^4 = 0$

```
In [ ]: sympy.solve(x**4 - 1, x)
```

También se pueden resolver sistemas de ecuaciones en varias variables. Ahora el primer argumento de solve es una lista cuyos elementos son las ecuaciones a resolver, y el segundo argumento es otra lista formada por las variables o incógnitas del sistema. Por ejemplo:

```
In [ ]: sympy.solve([x + 5*y - 2, -3*x + 6*y - 15], [x, y])
```

Puede resolver ecuaciones en las que intervengan funciones trascendentes, pero su capacidad es limitada y, a veces, no encuentra la solución.

```
In [ ]: sympy.solve(exp(x) + 1, x)
```

2.2.4 Factorizacion

Es posible factorizar polinomios en términos de factores irreducibles con la función factor(función):

```
In [ ]: from sympy import factor
        f = x**4 - 3*x**2 + 1
        factor(f)
```

2.2.5 Ecuaciones diferenciales

```
In [ ]: x, g = sympy.symbols('x g')
        expresion = g(x).diff(x, x) + g(x)
        expresion
```

```
In [ ]: sympy.dsolve(expresion, g(x))
```

Si uno quiere la ecuación diferencial sea resuelta de un modo en particular, puede incluir "pistas" como segundo argumento de dsolve para guiarlo en la resolución:

```
In [ ]: expresion = sin(x)*cos(g(x)) + cos(x)*sin(g(x))*g(x).diff(x)
        sympy.dsolve(expresion, g(x), hint='separable')
```

2.2.6 Sympy, numpy y otras yerbas

Sympy es capaz de evaluar numéricamente una expresión usando Numpy. Para esto se usa la función lambdify que sabe generar una función universal

```
In [ ]: expr = sin(x)/x
        f = sympy.lambdify(x, expr, "numpy")
```

```
In [ ]: a = np.linspace(0.0001, 10, 100)
        f(a)
```

Pero sympy va más lejos y sabe compilar una expresión, usando lenguajes como Fortran o Cython que son aún más rápidos que usar C.

```
In [ ]: from sympy.utilities.autowrap import ufuncify
        f2 = ufuncify([x], expr, backend='f2py')
```

```

In [ ]: f2(a)

In [ ]: # para los incrédulos
        np.all(f(a) == f2(a))

In [ ]: %timeit f(a)

In [ ]:

In [ ]: %timeit f2(a)

```

3 C'est fini

Pero hay mucho más en este curso para seguir por tu cuenta

- Gráficos más lindos con Seaborn y análisis de datos con Pandas [Clase](#)
- Interactivad y contenidos enriquecidos en el notebook [Clase](#)
- Integrar Python con código Fortran (puaj!) [[Clase](#)](/notebooks/Clase%205b.ipynb)
- Acelerando Python con Cython y Numba [Clase](#)
- Cómo instalar otros paquetes y distribuir los tuyos [Clase](#)

Además, hay cientos de bibliotecas científicas para temáticas específicas, que en general se basan en Numpy/Scipy

- Machine Learning: [Scikit-learn](#)
- Procesamiento de imágenes avanzado: [Scikit-Image](#) y [OpenCV](#)
- Paralelización/GPU: [Theano](#)
- Astronomía: [Astropy](#)
- Biología molecular: [Biopython](#)
- Química molecular: [Chemlab](#)
- Y [decenas](#), qué digo, [cientos](#), qué digo, [miles](#) paquetes más!

3.1 Tareas Importantes

- Sumate a la comunidad de Python más copada del mundo

```

<header class="list-group-item">
  <h4 class="list-group-item-heading">Suscribite a la lista <a href="http://python.org.a
</header>
<article class="list-group-item">
  <form id="newsletter-form" class="inline-form" action="http://listas.python.org.ar/mai
    <div class="form-group">
      <input class="form-control" name="email" placeholder="Email">
    </div>
    <input class="btn btn-primary" type="submit" value="Inscribirme a Pyar!">
  </form>
</article>
</section>

```

- Vení a la conferencia anual! es Gratis!

Y dame feedback (sincero) sobre el curso

```
In [10]: IFrame(src="https://docs.google.com/forms/d/1e2zHvV6ae3bQAjLsZuyZ_C5AJ2PGDkdpUD2pBH3mMR")
```

```
Out[10]: <IPython.lib.display.IFrame at 0x7f2c6065bb38>
```

```
In [ ]:
```

notebook6

June 11, 2017

1 Introducción a Python para ciencias e ingenierías (notebook 6)

Ing. Martín Gaitán

- Email: gaitan@gmail.com

1.1 Seaborn

Antes de empezar con pandas vamos a configurar [Seaborn](#), un paquete que, entre otras cosas, mejora la estética de gráficos de matplotlib

Para instalar, como casi siempre

```
conda install seaborn
```

o

```
pip install seaborn
```

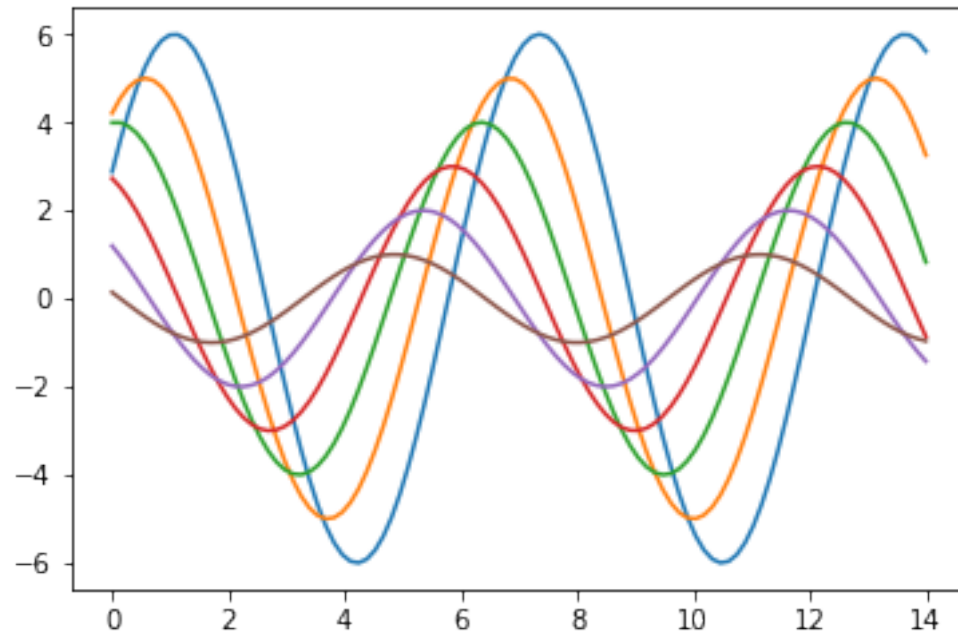
Veamos un gráfico default generado por matplotlib

```
In [1]: %matplotlib inline
```

```
In [2]: import matplotlib.pyplot as plt
import numpy as np
```

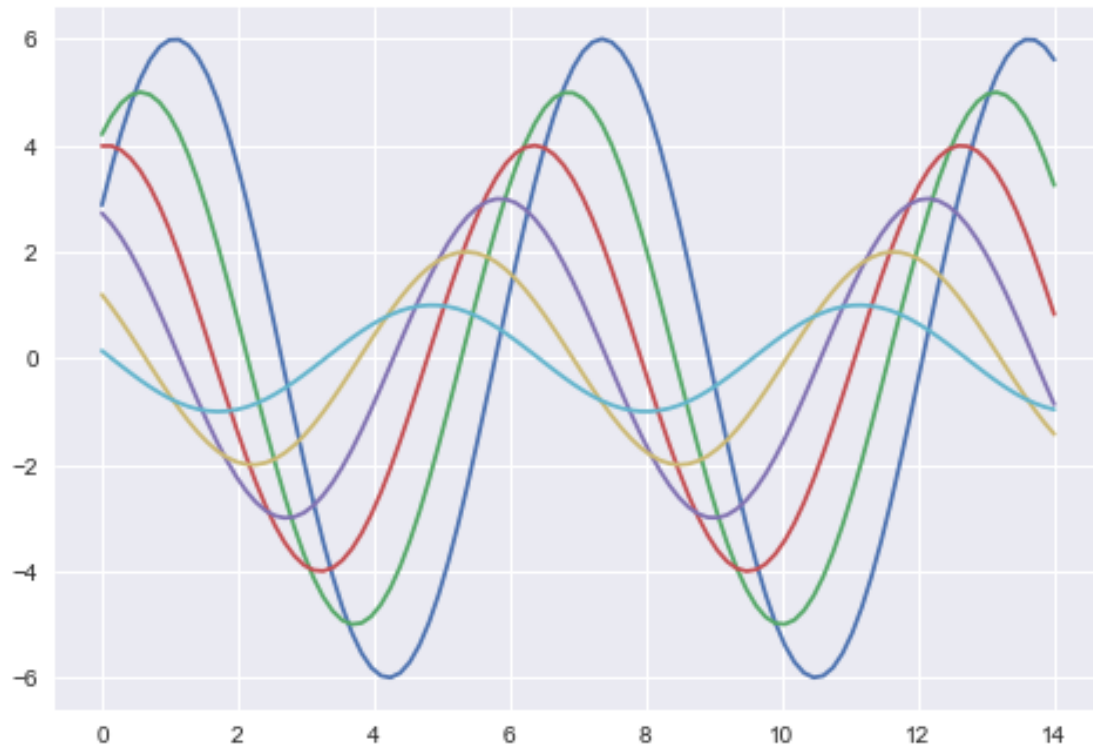
```
In [3]: def sinplot():
    x = np.linspace(0, 14, 100)
    for i in range(1, 7):
        plt.plot(x, np.sin(x + i * .5) * (7 - i))
```

```
sinplot()
```



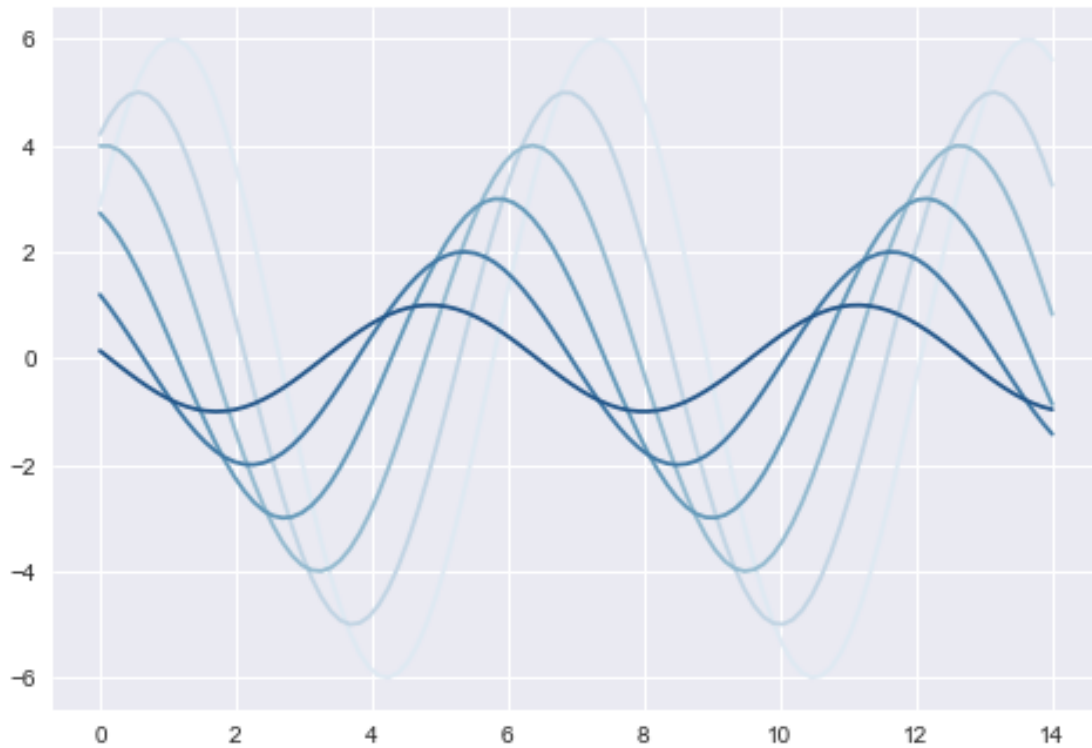
```
In [4]: import seaborn as sns    # magia !  
        sinplot();
```

```
/home/tin/.virtualenvs/curso/lib/python3.6/site-packages/IPython/html.py:14: ShimWarning: The `IPython.html.widgets` has moved to `ipywidgets`.", ShimWarning)
```



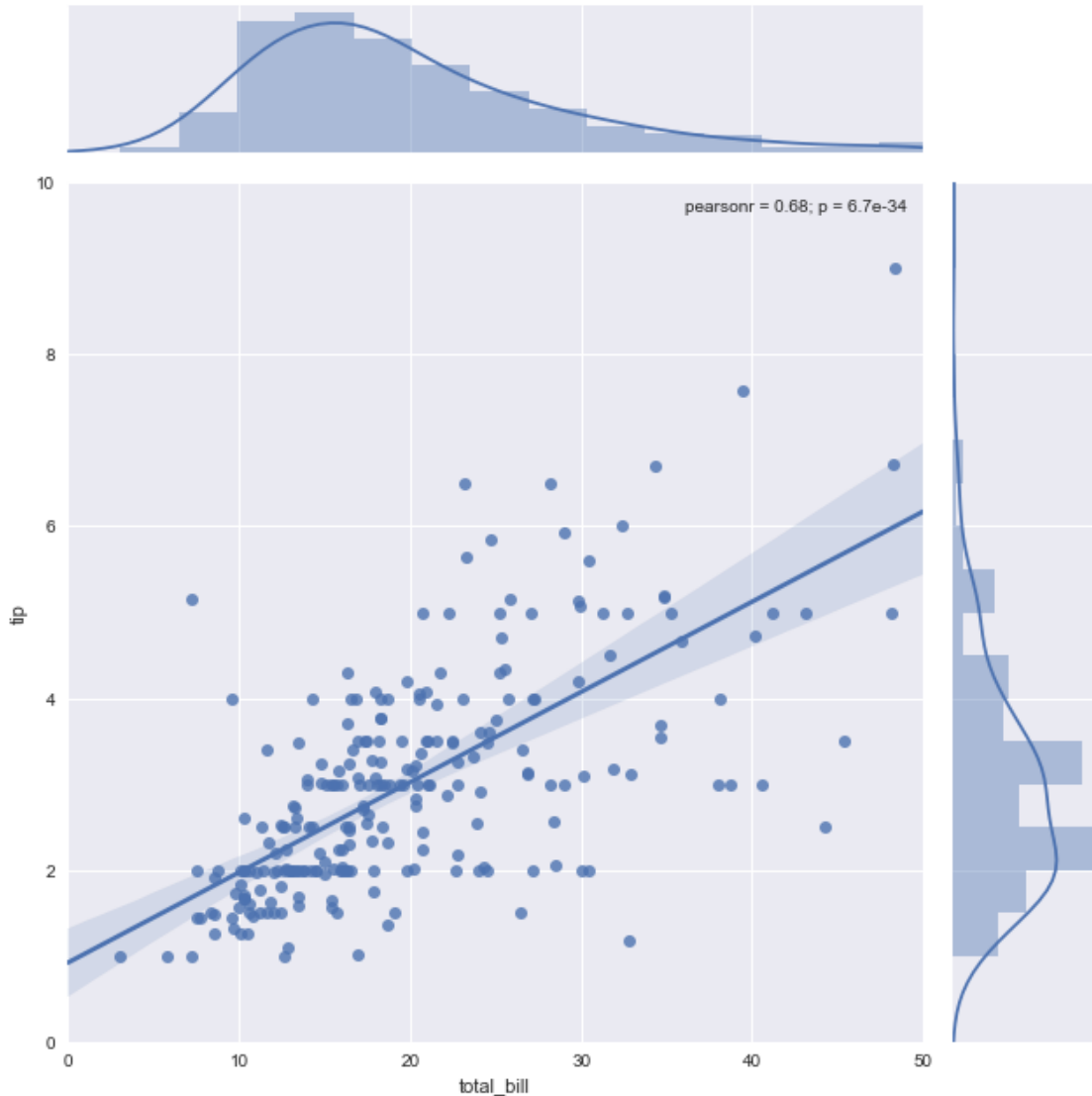
Seaborn permite configurar paletas, estilos de una manera muy fácil, globalmente o para un gráfico en particular

```
In [5]: with sns.color_palette("Blues", desat=.7):  
        sinplot()
```



Al ser un paquete especialmente diseñado para **estadística**, además de gráficos más lindos, tiene **gráficos especiales** e incorpora "datasets" de prueba

```
In [6]: tips = sns.load_dataset("tips")  
        g = sns.jointplot("total_bill", "tip", data=tips, kind="reg", xlim=(0, 50), ylim=(0, 10))
```

¿Y qué ese dataset? Veamos

```
In [7]: type(tips)
```

```
Out[7]: pandas.core.frame.DataFrame
```

1.2 Pandas

Pandas es una biblioteca para manipulación y análisis de datos basada en Numpy. Aporta nuevas estructuras de datos de alto nivel que extienden datos almacenados en arrays, aportando más semántica y nuevas operaciones.

Puede pensarse a Pandas como un **reemplazo pythonico a excel**

Pandas introduce dos estructuras de datos principales: Series y DataFrame.

```
In [8]: import pandas as pd
```

1.2.1 Series

Un Series es un objeto **unidimensional**, similar a un array, una lista o una columna en una tabla, que tiene **asociado una etiqueta** para cada elemento. Por defecto, esta etiqueta es un número de 0 a N

```
In [9]: s = pd.Series([1,3,5,np.nan,6,8]) # creamos una serie, analogo a un array de 1D
s
```

```
Out[9]: 0    1.0
        1    3.0
        2    5.0
        3    NaN
        4    6.0
        5    8.0
        dtype: float64
```

```
In [10]: pd.Series?
```

Como vemos, es simplemente un envoltorio más lindo: el verdadero contenedor es un array de numpy

```
In [11]: s.values
```

```
Out[11]: array([ 1.,  3.,  5., nan,  6.,  8.])
```

```
In [20]: s.values.ndim, s.values.shape
```

```
Out[20]: (1, (6,))
```

Si bien una serie se puede indizar directamente como una secuencia, la forma correcta y potente es utilizar los métodos especiales `loc` (basado en valor) e `iloc` (basado en posición).

```
In [17]: s.index
```

```
Out[17]: RangeIndex(start=0, stop=6, step=1)
```

```
In [21]: s.loc[0], s.iloc[0]
```

```
Out[21]: (1.0, 1.0)
```

¿Pero entonces una Serie es una secuencia?. En realidad es más parecido a un diccionario ordenado

```
In [22]: s[0]
```

```
Out[22]: 1.0
```

```
In [23]: s[-1]
```

```

-----

KeyError                                Traceback (most recent call last)

<ipython-input-23-0e2107f91cbd> in <module>()
----> 1 s[-1]

~/.virtualenvs/curso/lib/python3.6/site-packages/pandas/core/series.py in __getitem__(self, key)
    599         key = com._apply_if_callable(key, self)
    600         try:
--> 601             result = self.index.get_value(self, key)
    602
    603             if not is_scalar(result):

~/.virtualenvs/curso/lib/python3.6/site-packages/pandas/core/indexes/base.py in get_value(self, key, tz)
    2475         try:
    2476             return self._engine.get_value(s, k,
-> 2477                                         tz=getattr(series.dtype, 'tz', None))
    2478         except KeyError as e1:
    2479             if len(self) > 0 and self.inferred_type in ['integer', 'boolean']:

pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_value (pandas/_libs/index.c:5167)

pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_value (pandas/_libs/index.c:5167)

pandas/_libs/index.pyx in pandas._libs.index.IndexEngine.get_loc (pandas/_libs/index.c:5167)

pandas/_libs/hashtable_class_helper.pxi in pandas._libs.hashtable.Int64HashTable.get_item (pandas/_libs/hashtable_class_helper.pxi:1324)

pandas/_libs/hashtable_class_helper.pxi in pandas._libs.hashtable.Int64HashTable.get_item (pandas/_libs/hashtable_class_helper.pxi:1324)

KeyError: -1

```

```
In [ ]: s.ix[-1]
```

Hay muchas operaciones que podemos hacer sobre una serie.

```
In [ ]: s.mean(), s.sum(), s.std()
```

E incluso ver un resumen general de los datos

```
In [ ]: s.describe()
```

Como vemos, es un poco más permisivo que Numpy con los datos faltantes (ese nan que hay por allí)

```
In [ ]: s.values.mean(), s.values.sum(), s.values.std()
```

Y hay operaciones que no están disponibles en numpy o se comportan distinto

```
In [ ]: s.median()
```

```
In [ ]: s.values.median() # np.median(s.values)
```

Pandas es una herramienta nacida en la industria de las finanzas, y por ello intenta hacer fácil las cosas típicas. Una de ellas es **graficar datos**. Utiliza **matplotlib** por default

```
In [ ]: s.plot()
```

Como dijimos, los valores de una serie tienen una etiqueta y se pueden definir explícitamente

```
In [ ]: goles = pd.Series([10, 54, 31, 0], index=['gaitan', 'messi', 'suarez', 'ronaldo'])
        goles
```

Cuando el índice no es de enteros, se puede buscar un valor tanto por clave como por posición (con reglas de slicing)

```
In [ ]: goles['messi']
```

```
In [ ]: goles[-1]
```

```
In [ ]: goles.ix[1:-1]
```

Obviamente, dado que se trata de claves y valores, se puede instanciar una serie directamente desde un diccionario.

```
In [ ]: pd.Series({'gaitan': 80, 'messi': 100})
```

Pero a diferencia de un diccionario, las etiquetas se pueden repetir. Es decir que pueden haber múltiples valores para una clave

```
In [ ]: s = pd.Series(np.random.rand(4) * 100, ['gaitan', 'messi', 'maradona', 'messi'])
        s['messi']
```

```
In [ ]: s.plot('bar')
```

Las series se utilizan mucho para representar datos en **función del tiempo**. Hay una función especial análoga a `range()` que permite generar índices temporales

```
In [ ]: pd.date_range('1/1/2010', periods=1000)
```

```
In [ ]: ts = pd.Series(np.random.randn(1000), index=pd.date_range('1/1/2010', periods=1000))
        ts = ts.cumsum()
        ts.plot()
```

1.2.2 Dataframe

Un DataFrame es una estructura tabular de **filas y columnas** (como una hoja de cálculo!). También se puede pensar un DataFrame como un conjunto de Series que comparten el índice (es decir, la primera columna)

```
In [ ]: df = pd.DataFrame(np.random.randn(1000,4), index=ts.index, columns=list('ABCD'))
        df.head()
```

```
In [ ]: df.tail()
```

Las columnas son Series!

```
In [ ]: type(df.A)
```

```
In [ ]: df["A"].cumsum().plot()
```

Se puede instanciar un DataFrame a partir de un diccionario.

```
In [ ]: df2 = pd.DataFrame({ 'A': 1.,
                             'B': pd.Timestamp('20130102'),
                             'C': pd.Series(1, index=range(4), dtype='float32'),
                             'D': np.array([3, 2, 1, .9]),
                             'E': 'foo' })

df2
```

```
In [ ]: df2[['B', 'C']]
```

```
In [ ]: import pickle
```

```
        pickle.loads(pickle.dumps(df2))
```

Pandas también puede importar archivos CSV o Excel, locales o remotos. Por ejemplo este [dataset](#) de remates al arco en la Champions League

```
In [ ]: url = 'https://raw.githubusercontent.com/rjtaavares/football-crunching/master/datasets/cl
        shots = pd.read_csv(url, index_col=0, na_values='N/A')
```

```
In [ ]: shots.columns
```

```
In [ ]: shots.head()
```

```
In [ ]:
```

Podemos filtrar el dataset por múltiples criterios. Por ejemplo, para calcular la distancia promedio de los goles que hizo el genio:

```
In [ ]: shots[(shots.player == 'Lionel Messi') & shots.goal].dist.mean()
```

Podemos separar subconjuntos basado en uno o más criterios

```
In [ ]: shots.groupby('team').describe()
```

```
In [ ]: shots.groupby(['team', 'goal']).dist.mean()
```

```
In [ ]:
```

Una alternativa aún más poderosa es hacer un pivot de datos, donde se definen una o múltiples filas y columnas para agrupar datos, y la dimensiones ('values') que se quieren ver resumidas. La función de agregación por defecto es `np.mean`

```
In [ ]: pt1 = pd.pivot_table(shots, index=['team'], columns=['goal'], values=['dist'])
pt1
```

como siempre, podemos graficar

```
In [ ]: pt1.plot(kind='bar', figsize=(13,6))
```

En los dataframes resultantes de un pivot, índice y claves no son lo mismo

```
In [ ]: pt1.index is pt1.keys()
```

Entonces esto no funciona

```
In [ ]: pt1['Barcelona']
```

```
In [ ]: pt2 = pd.pivot_table(shots, index=['team', 'player'], columns=['goal'], values=['dist'])
pt2
```

```
In [ ]: pt1.loc['Barcelona']
```

Las tablas pivoteantes son mucho más poderosas

```
In [ ]: pt2 = pd.pivot_table(shots, index=['team', 'player'], columns=['goal'], values=['dist'],
pt2
```

Notar que en este caso usamos un índice compuesto (team-player). Esto hace que el dataframe utilice un tipo de índice especial llamado `MultiIndex`

```
In [ ]: pt2.loc['Barcelona']
```

Podemos conseguir la fila particular para un índice compuesto

```
In [ ]: pt2.loc[('Barcelona', 'Lionel Messi')]
```

Y ahora podemos volver al dataset `tips` importado de `Seaborn`

```
In [ ]: tips.head()
```

```
In [ ]: tips.pivot_table(index=['day', 'sex'], columns=['time', 'smoker'], values=['total_bill'],
```

Y ya que estamos, veamos los gráficos facetados

```
In [ ]: g = sns.FacetGrid(tips, col="sex", hue="smoker")
g.map(plt.scatter, "total_bill", "tip")
g.add_legend();
```

Pandas es una herramienta poderosísima y sólo vimos un poquito. ¡A estudiarlo!

2 Python es infinito: Instalando paquetes

Python es un lenguaje utilizado en muchísimas áreas. El repositorio donde se suelen compartir paquetes se llama PyPi, el Python Package Index.

Desafío: elijan un tópico cualquiera y busquemos un paquete para instalar

```
In [ ]: from IPython.display import IFrame
        IFrame('http://pypi.python.org/pypi', '100%', 400)
```

La herramienta oficial para instalar paquetes es [pip](#). La forma canónica es

```
pip install --user <nombre_paquete>
```

(el flag `--user` (opcional) instala el paquete a nivel usuario, sin requerir permisos de administración)

```
In [ ]: !pip install --help
```

Pip busca, baja, descomprime, e instala (y repite los pasos recursivamente para las dependencias). Por ejemplo, podemos instalar [pint](#), del amigo [Hernán Greco](#)

(Nota: esto no va a funcionar en `try.jupyter.org` debido a las limitaciones de salida de red por cuestiones de seguridad)

```
In [ ]: !pip install pint
```

```
In [ ]: import pint
        ur = pint.UnitRegistry()
```

```
In [ ]: vel = 60 * ur.km / ur.hr
        vel
```

```
In [ ]: vel.to('m/s')
```

Pero **¿dónde se instala un paquete?**. Si no le decimos lo contrario con algún parámetro del instalador, el paquete se descomprime y se copia a un directorio especial para Python llamado `site-packages`, que a su vez se encuentra en una **lista de directorios** (definida como una variable de entorno del sistema) en la que la maquinaria de importación de Python busca paquetes

```
In [ ]: pint.__file__
```

```
In [ ]: import sys
        sys.path
```

Para desinstalar paquetes, el subcomando de pip es `uninstall`

```
In [ ]: !pip freeze
```

Notaron ese directorio `".virtualenvs"` que se lee en mi `sys.path`? Es que yo estoy usando un **entorno virtual**

2.0.1 Entornos virtuales

Un entorno virtual (virtualenv) en python es una compartimentalización para tener multiples conjuntos de **dependencias por proyecto**, instalándolas en un directorio particular. Es decir, podemos tener un virtualenv en el que usamos numpy 1.8 y no vernos obligado a actualizarlo si comenzamos otro para el que necesitamos una feature que aparece en numpy 2.0.

Python 3 incorpora built-in una herramienta llamada [pyenv](#) para manejar entornos virtuales, pero la forma canónica sigue siendo utilizar la herramienta [virtuelenv](#)

Primero instalamos virtualenv

```
pip install virtualenv # o pip install --user virtualenv
```

Para crear un virtualenv basta usar

```
virtualenv <DIR>
```

Donde <DIR> es el directorio donde queremos instalar las dependencias exclusivas para nuestro proyecto. En mi caso es `~/virtualenvs/curso`

Una vez creado un virtualenv, cada vez que se quiere usarlo hay que activarlo. En sistemas linux/mac:

```
source <DIR>/bin/activate
```

o en sistemas windows:

```
<DIR>\Scripts\activate
```

Cuando el virtualenv está activado, se denotará por un prefijo en el prompt. Luego, podemos instalar paquetes directamente via pip, sin permisos de administrador, ya que estamos instalando los paquetes en el `site-packages` definido dentro del virtualenv.

Para facilitar la creacion y activacion de entornos con virtualenv se puede usar [virtualenvwrapper](#)

Para salir de un virtualenv (desactivarlo) se ejecuta

```
deactivate
```

2.1 Conda

Un problema del tandem virtualenv-pip es que, en general, los paquetes instalables son archivos comprimidos de **código fuente**. Eso no es ningun problema cuando se trata de paquetes python puro, porque Python se interpreta y no se compila, pero cuando es un paquete que tiene dependencias externas (fortran, cython, c, etc.) o requiere dependencias complejas en el sistema operativo, la instalación via pip requerirá compilar, eso implica compiladores, headers y un montón de cosas que complican la vida (además de ser un proceso lento).

Por eso la herramienta [conda](#), que viene preinstalada si usamos la distribución de python Anaconda, es muy interesante. En vez de bajar código fuente, baja paquetes precompilados (binarios) en un formato y desde un [repositorio especial](#) (distinto a PyPi).

Además, conda sabe crear por sí mismo entornos virtuales, es decir que es una mezcla entre virtualenv y pip.

Para crear un entorno virtual via conda


```
conda create -n <nombre_venv> <programas o paquetes basicos a instalar, opcionalmente con version>
```

Por ejemplo:

```
conda create -n curso python=3.4 numpy scipy matplotlib pandas sympy seaborn ipython ipython-notebook
```

Para activar el entorno, es análogo a la manera de `virtualenv`

```
source activate curso
```

O en Windows, directamente

```
activate curso
```

Para instalar paquetes dentro del entorno,

```
conda install <paquetes o programas>
```

2.2 Distribuyendo tus paquetes (o módulos)

Ya vimos algo de esto. Python usa un archivo especial llamado `setup.py` que invoca a una función llamada `setup()` con diversos parámetros que especifican de qué se trata el paquete, la versión, el autor, la lista explícita de módulos y paquetes que debe incluir, etc.

Este *script* `setup.py` es llamado por python (directamente o a través de pip) y nos presenta una lista de subcomandos.

Se nuevo sucede que Python trae incorporada lo necesario para escribir este archivo de `setup`, en el paquete `distutils` de la biblioteca estandar, pero la opción canónica superadora es usar el paquete [setuptools](#)

```
In [ ]: %cd ..
```

```
In [ ]: %mkdir test_pack/
        %mkdir test_pack/marquesina
```

```
In [ ]: %cd test_pack
```

```
In [ ]: %ls
```

Supongamos que queremos distribuir nuestra [función marquesina](#) de la clase 2. Vamos a crear un paquete. Como es nuestro único código, lo podemos poner directamente en el `__init__.py` del proyecto

```
In [ ]: %%writefile marquesina/__init__.py
```

```
def marquesina(cadena, ancho=60, alto=1, character="*"):
    """
    Wraps ``cadena`` in a box of ``character`` symbols with padding ``ancho`` x ``alto``
    """

    cadena = cadena.center(ancho)
```

```

cadena = caracter + cadena[1:-1] + caracter
cadena += '\n'
relleno = " " * ancho
relleno = caracter + relleno[1:-1] + caracter
relleno += '\n'
tapa = caracter * ancho
return tapa + '\n' + relleno * alto + cadena + relleno * alto + tapa

```

Siguiendo la [doc](#), una versión mínima del `setup.py` se vería así

```

In [ ]: %%writefile setup.py
        from setuptools import setup, find_packages

        setup(
            name='marquesina',
            version = '0.1',
            description='Marquesina, decorate your text',
            long_description='A package with functions to decorate text',
            author='Martín Gaitán',
            author_email='curso@',
            packages = find_packages(),
        )

```

Así se ve el árbol de archivos

```

In [ ]: !tree

```

Nuestro `setup.py` es un programa de línea de comandos, con ayuda y todo!

```

In [ ]: !python setup.py install --help

```

Ahora podemos instalar nuestro paquete, por ejemplo via `pip install -e` que instala el paquete sin copiar los archivos a `site-packages`, permitiendo que si modificamos el código impacte automáticamente en la versión importable

```

In [ ]: !pip install -e .

```

```

In [ ]: from marquesina import marquesina

```

```

In [ ]: print(marquesina('hola mundo'))

```

Supongamos que lanzamos una nueva versión que incorpora otra función

```

In [ ]: %%writefile -a marquesina/__init__.py

        def boca():
            print(marquesina('¡Agua Boca!'))

```

No haría falta cambiar el `setup.py` para importar el paquete o sus funciones, pero podemos usar una funcionalidad de `setuptools` que permite convertir funciones en simples programas de línea de comando instalables.

```
In [ ]: %%writefile setup.py
        from setuptools import setup, find_packages

        setup(
            name='marquesina',
            version = '0.2',
            description='Marquesina, decorate your text',
            author='Martín Gaitán',
            author_email='curso@',
            packages = find_packages(),      # ['marquesina']
            entry_points = {
                'console_scripts': [          # esto crear un script 'boquita' que ejecuta mar
                    'boquita = marquesina:boca',
                ]
            }
        )
```

```
In [ ]: !pip install -e .
```

Entonces **boquita** ahora es un programa en nuestro sistema!

```
In [ ]: !boquita
```

Para escribir "scripts" de linea de comandos que reciban parámetros, se puede utilizar el paquete de las biblioteca estándar [argparse](#) o bien usar [Docopt](#)
Vamos más allá

```
In [ ]: %%writefile marquesina/plot.py

import matplotlib.pyplot as plt

def marquesina_plt(text, x=5, y=2, text_size=80):
    """
    return create a matplotlib's Text instance of `text`
    """
    coef = x/max(len(text), 6)
    plt.figure(figsize=(x/coef, y/coef))
    plt.tick_params(bottom=False, left=False, right=False, top=False, labelleft=False, 1
    return plt.text(0.5, 0.5, text, size=text_size, ha="center", va="center")
```

```
In [ ]: marquesina_plt('aguante boca!', x=2)
```

```
In [ ]:
```

```
In [ ]: %%writefile setup.py
        from setuptools import setup, find_packages

        setup(
            name='marquesina',
            version = '0.3',
```

```

description='Marquesina, decorate your text',
author='Martín Gaitán',
author_email='curso@',
packages = find_packages(),      # ['marquesina']
install_requires=['matplotlib'],
entry_points = {
    'console_scripts': [          # esto crear un script 'boquita' que ejecuta mar
        'boquita = marquesina:boca',
    ]}
)

```

```
In [ ]: !pip install -e .
```

```
In [ ]: from marquesina.plot import marquesina_plt
```

```
In [ ]: %cd test_pack/
```

Luego, podemos armar nuestro paquete distribuible. Es un simple archivo comprimido con todo lo necesario

```
In [ ]: !python setup.py sdist
```

```
In [ ]: !ls dist
```

Setuptools también sabe registrar y subir nuestro paquete a PyPi

```
# solo hace falta hacerlo la primera vez
python setup.py register
```

Luego cada vez que queramos publicar una nueva versión

```
python setup.py sdist upload
```

```
In [ ]:
```