

# 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

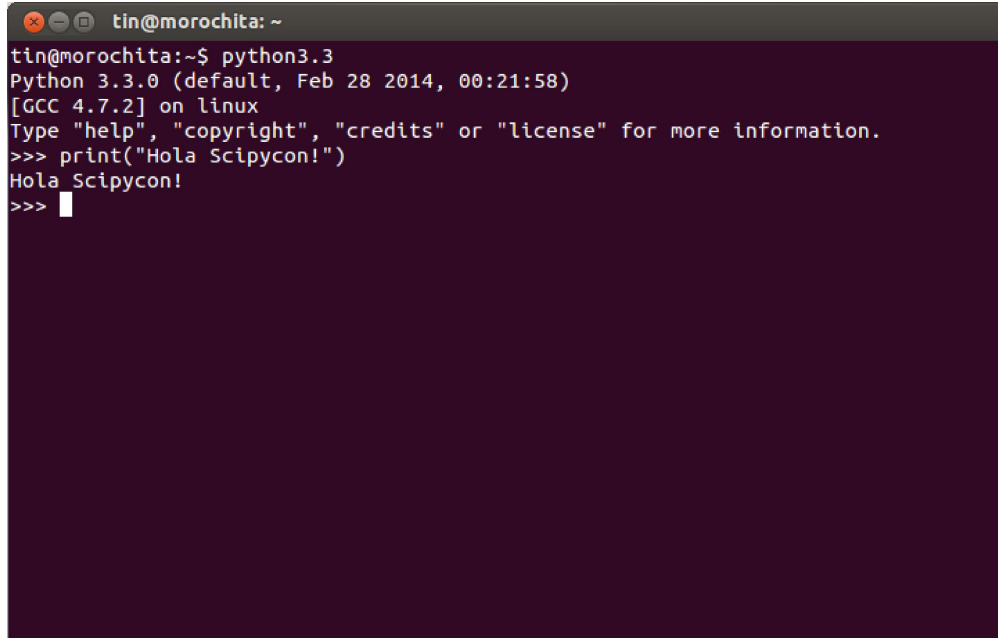
### 1.1.1 Instalación

- 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



```
tin@morochita: ~  
tin@morochita:~$ python3.3  
Python 3.3.0 (default, Feb 28 2014, 00:21:58)  
[GCC 4.7.2] on linux  
Type "help", "copyright", "credits" or "license" for more information.  
>>> print("Hola Scipycon!")  
Hola Scipycon!  
>>> 
```

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 inscruados 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 intérprete Python y obtendremos el resultado

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

```
Hola curso
```

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

```
¡Hola curso!
```

```
In [ ]:
```

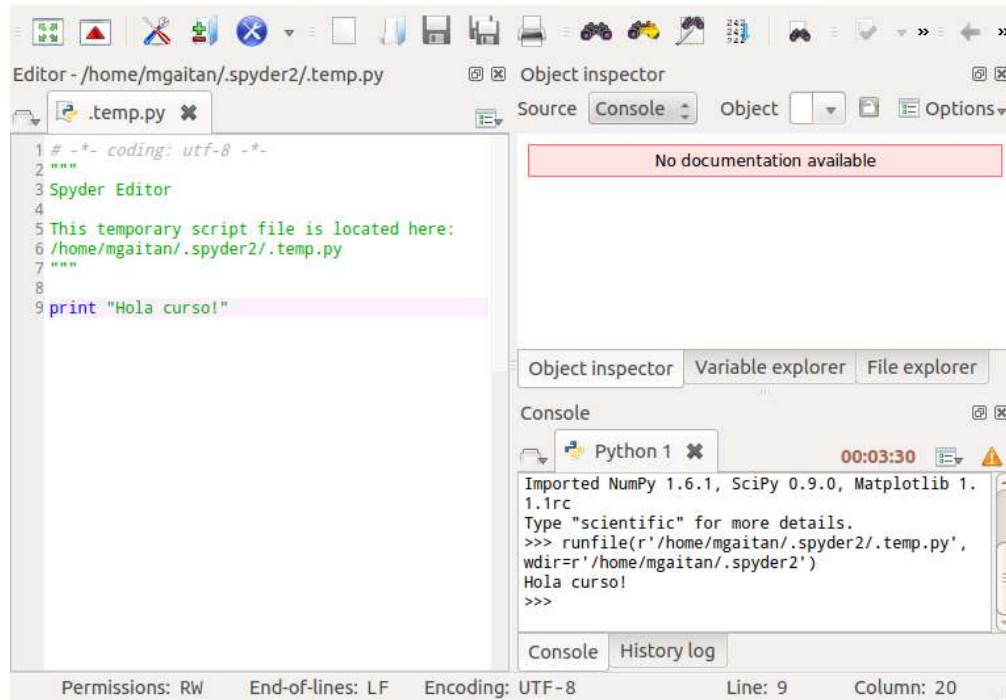
IPython agrega muchas funcionalidades complementarias que no son parte del lenguaje Python. Por ejemplo el signo `!` que precede la línea anterior indica que se ejecutará un programa/comando del sistema en vez de código python

### 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 **instrospección**. 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'Higgings"
         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

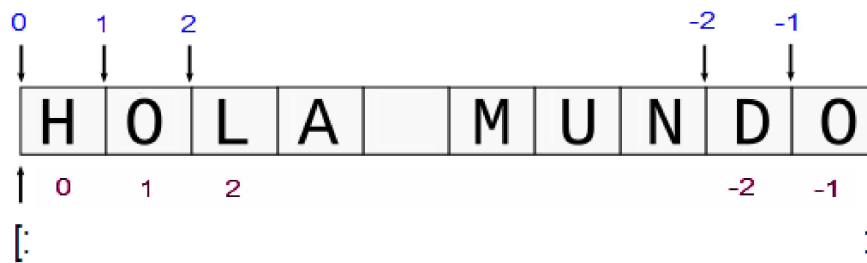
```
In [73]: a = "hola,amigos,como"
          a.split(',')
```

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

Y el método inverso es join, para unir muchas cadenas intercalándolas 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 argumento
```

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

```
In [40]: "La parte real es {numero.real:.5f} y la imaginaria es {numero.imag} {numero}".format(numero=1+2j)
```

```
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, "O'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 r
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 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 explícitos.

```
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 formula  $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$  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', 'cismor
```

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)
```