Python I: variables

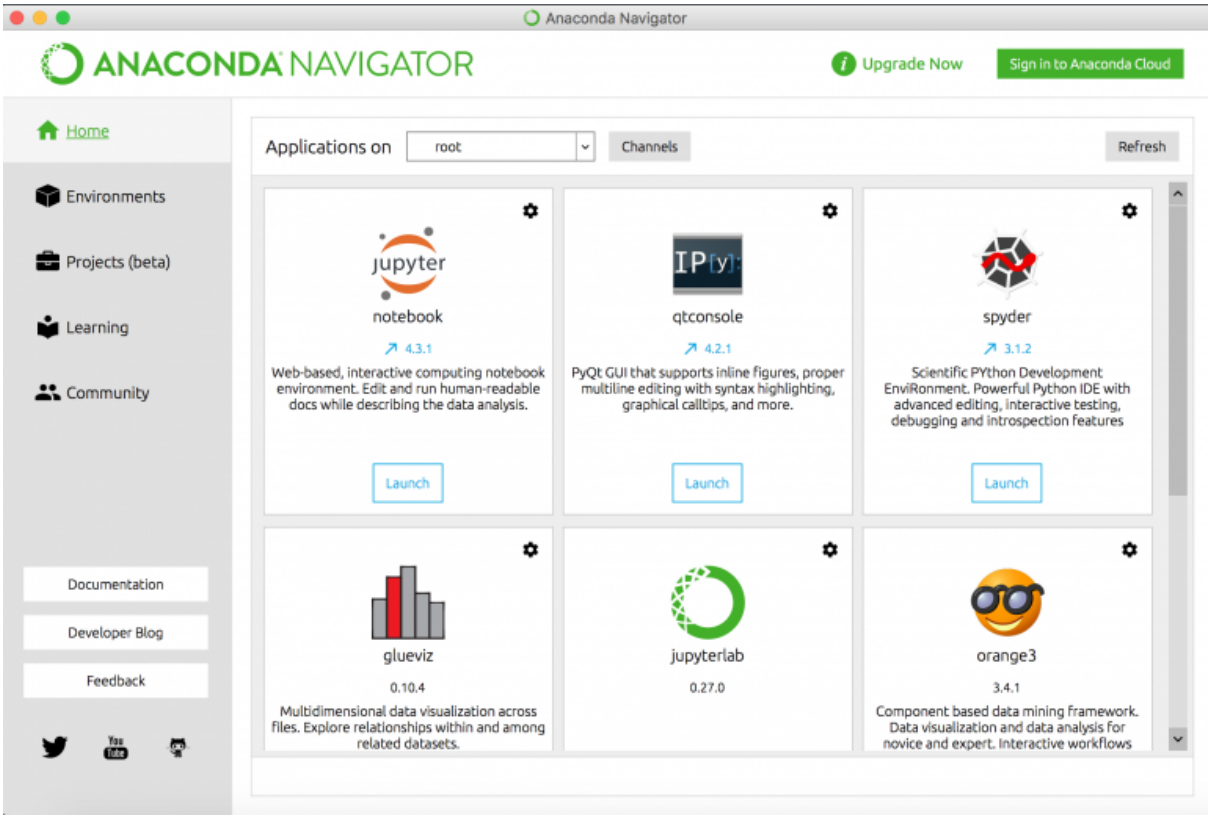
Llegó la hora de empezar a trabajar con nuestro lenguaje de programación elegido: Python.

Python es un lenguaje interpretado de cuarta generación. Su filosofía se resume en el ***Zen de Python***:

> Bello es mejor que feo.
> **Explícito es mejor que implícito.**
> **Simple es mejor que complejo.**
> Complejo es mejor que complicado.
> Plano es mejor que anidado.
> Espaciado es mejor que denso.
> **La legibilidad es importante.**
> Los casos especiales no son tan especiales como para romper las reglas.
> Sin embargo **la practicidad le gana a la pureza**.
> **Los errores nunca deberían pasar silenciosamente.**
> A menos que se silencien explícitamente.
> Frente a la ambigüedad, evitar la tentación de adivinar.
> Debería haber una, y preferiblemente solo una, manera obvia de hacerlo.
> A pesar de que esa manera no sea obvia a menos que seas Holandés.
> Ahora es mejor que nunca.
> A pesar de que nunca es muchas veces mejor que *ahora mismo*.
> **Si la implementación es difícil de explicar, es una mala idea.**
> Si la implementación es fácil de explicar, puede que sea una buena idea.
> Los espacios de nombres son una gran idea, ¡tengamos más de esos!

En resumen, Python busca ser un lenguaje flexible, fácil de escribir y, sobre todo, fácil de leer. El costo a pagar es que tiende a ser más lento que otros lenguajes, en términos de la cantidad de operaciones por unidad de tiempo. Por otra parte, una de las grandes fortalezas de Python es la gran comunidad de desarrolladores, lo que significa extensa documentación y un sinnúmero de recursos en línea. La documentación oficial de Python (con más detalle del que jamás podríamos leer por completo), están en `https://docs.python.org/3/`.

Para trabajar en Python, recomiendo descargar Anaconda, que incluye la terminal interactiva de Python, *cuadernos* y Spyder, un ejemplo de *Entorno de Desarrollo Integrado* (IDE) - un editor de texto inteligente. Para esto, vayan a `https://www.anaconda.com/products/individual-d` y hagan click en Descargar. Durante la instalación, no modifiquen las opciones. Al terminar abran el programa *Anaconda Navigator*:



Alternativamente pueden simplemente usar un IDE como VisualStudio Code, Spyder o Atom. Python tiene su propia terminal interactiva, a la que podemos acceder ejecutando

```
ipython
```

Aquí, podemos escribir nuestro primer programa:

```
print('Hello world!')
```

Para obtener ayuda:

```
help(print)
```

Podemos avanzar más rápidamente con algunos conceptos básicos análogos a los vistos en bash. (En Python, como en Bash, se pueden introducir comentarios – texto libre que no es ejecutado por el programa – anteponiendo un `#`, ya sea al comienzo de una línea o a la mitad.)

Variables numéricas

Tal como en bash, en python no es necesario declarar las variables; se declaran implícitamente la primera vez que se utilizan:

```
x = 4
```

Una variable puede alojar cualquier tipo de objeto. En un lenguaje de menor nivel (ej., `C`, `FORTRAN`), las variables se declaran más o menos así:

```
x = int 4
```

y el tipo de variable al que corresponde x queda fijo; para cambiarlo es necesario primero eliminar la variable manualmente y luego redefinirla de manera similar. Python, en cambio, debe adivinar el tipo del objeto a partir de su *asignación*, cada vez que se asigna un valor a una variable. Iremos viendo las posibilidades durante el curso. Si queremos saber qué tipo de objeto es la variable x, podemos hacer

```
type(x)
```

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

Noten que no escribí `print(type(x))`. Cuando realizamos una operación en la terminal de Python que no asignamos a una variable (también podríamos haber hecho `tp = type(x)`), el resultado de esa operación se imprime directamente en la terminal como el estado de salida del comando. Si sólo queremos saber el resultado de una operación, pero no almacenarlo (por ejemplo un cálculo simple, o una comprobación como la anterior), podemos ahorrarnos el `print`.

Entonces, `x` es un entero, **`int`**. También podemos definir números reales (**`float`**):

```
y = 3.6
```

```
type(y)
```

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

Puede obtenerse un int a partir de un float, y vice versa:

```
int(y)
```

```
> 3
```

```
float(x)
```

```
> 4.0
```

Podemos hacer aritmética con `int` y `float`:

```
y + 4*x**0.9
```

```
> 17.528809012737987
```

El operador `**` define una potencia. Ocurre algo especial con la división:

```
y / x
```

```
> 0.9
```

La división de dos `int`s dio como resultado un `float`. En muchos lenguajes esto no ocurriría, y habríamos tenido que definir originalmente

```
x = 4.

type(x)

> <class 'float'>
```

El punto hace que x sea `float` (`x` es implícitamente `4.0`), no `int`. En Python la transformación es transparente al usuario, y casi nunca es necesario definir un float explícitamente cuando el número es entero. (De hecho, sí era necesario en versiones anteriores de python.) Este es otro aspecto muy distinto de Python respecto de otros lenguajes como C o FORTRAN, en los que las variables se definen de un tipo específico que no puede ser modificado. Si queremos evaluar la razón *entera* entre estos dos números podemos usar el operador `//`:

```
y // x

> 0.0
```

El resultado sigue siendo un `float`, pero corresponde al número entero de veces que y cabe en x. En cambio,

```
9 // 2

> 4

2 // 9

> 0
```

La operación solo involucra `int`s, por lo que el resultado es un `int` también. Por último, veremos el operador “módulo”:

```
9 % 2

> 1

3.4 % 2

> 1.4
```

que corresponde al resto de la división.

Float: números de punto flotante

Veamos el siguiente ejemplo:

```
4.1 % 2

> 0.09999999999999964
```

El término *float* se origina en la representación de los números reales en un computador. Estos números se representan como un entero multiplicado por alguna base elevado a una potencia,

$$x = a \cdot b^e$$

de manera que el exponente, e, puede "flotar" para representar distintos números. Esto es más fácil verlo en base 10. Por ejemplo,

$$12.34 = 1234 \cdot 10^{-2}$$

$$1.234 = 1234 \cdot 10^{-3}$$

de manera que sólo es necesario almacenar los números `1234` y `-2` (en el primer caso). Los computadores siempre representan los números reales con base 2. Por ejemplo, esto significa que el número 0.1 no puede calcularse exactamente; su valor más cercano se obtiene (con "64 bits" de precisión) con:

```
3602879701896397 / 2**55
```

igual a 0.1000000000000000055511151231257827021181583404541015625. Esto es más precisión de la que necesitaremos probablemente en todas nuestras aplicaciones, pero estas diferencias pueden acumularse, de manera que por ejemplo

```
0.3 - 3*0.1

> -5.551115123125783e-17
```

De hecho,

```
0.3 - 0.1 - 0.1 - 0.1

> -2.7755575615628914e-17
```

Por suerte, Python se encarga de imprimir números amigables la mayoría de las veces.

Strings

También podemos definir variables literales, "strings" (`str`):

```
z = 'Hola mundo!'
```

```
type(z)

> <class 'str'>
```

Un número definido entre comillas (o apóstrofes) será un string, no una variable numérica:

```
p = '2'

type(p)

> <class 'str'>

x + p

> TypeError: unsupported operand type(s) for +: 'float' and 'str'
```

Hemos cometido nuestro primer error (Urra!): un *error de tipo*. No puede sumarse un float con un string. Pero podemos convertir nuestro string a float (o int) y no habrá problema:

```
x + float(p)

> 6.0
```

Los strings tienen varias características útiles. Se puede acceder cada uno de los caracteres indexando un string, *contando desde cero*:

```
z[0]

> 'H'
```

y en reversa con números negativos desde -1:

```
z[-1]

> '!'
```

Se pueden concatenar dos strings:

```
z + ' Soy Cristobal!'

> 'Hola mundo! Soy Cristobal!'
```

Podemos usar variables dentro de strings de dos maneras. La manera preferida es definiendo “f-strings”, las que especificamos con una f precediendo la declaración:

```
f'y vale {y}'

> 'y vale 3.6'
```

La segunda forma es usando la operación - el *método* - `format` de un string:

```
'y vale {0}'.format(y)

> 'y vale 3.6'
```

Ambos pueden hacer lo mismo, pero el primero es más compacto y más rápido. Otros ejemplos:

```
f'y vale {y}; por ejemplo, y+x = {y}+{x} = {y+x}'

'y vale {0}; por ejemplo, y+x = {0}+{1} = {2}'.format(y, x, y+x)
```

En ambos casos el resultado es:

```
> f'y vale 3.6; por ejemplo, y+x = 3.6+4 = 7.6'
```

Por supuesto, también podríamos haber impreso un string dentro de otro en lugar de concatenarlos (en general, esto debería preferirse):

```
f'{z} Soy Cristobal!'

> 'Hola mundo! Soy Cristobal!'
```

Incluso podemos hacer todo tipo de operaciones dentro de las llaves:

```
f'y * 2**x / (y-1) = {y*2**x/(y-1}'

> 'y * 2**x / (y-1) = 22.153846153846153'
```

Formatos de impresión

Podemos darle un formato específico a cada variable impresa. Por ejemplo, para imprimir exactamente dos decimales de un número real:

```
f'y vale {y:.2f}'

> 'y vale 3.60'
```

e incluso especificar el número de espacios totales utilizados:

```
f'y vale [{y:6:.2f}]'

> 'y vale [  3.60]'
```

o imprimirlo en notación científica:

```
f'{y:.2e}'
```

```
> '3.60e+00'
```

```
f'{y-3*1.2:.5e}'
```

```
> '4.44089e-16'
```

Podemos manipular el formato de un número entero:

```
f'x vale [{x:03d}]'
```

```
> 'x vale [004d]'
```

o un string:

```
f'|{z:20s}|'
```

```
> '|Hola mundo!         |'
```

```
f'|{z:>20s}|'
```

```
> '|         Hola mundo!|'
```

```
f'|{z:^20s}|'
```

```
> '|    Hola mundo!     |'
```

Reemplazando, si queremos, los espacios por cualquier carácter:

```
f'|{z:->20s}|'
```

```
> '|---------Hola mundo!|'
```

Para mensajes alineados a la izquierda podemos usar el carácter **<**:

```
f'|{z:-<20s}|'
```

```
> '|Hola mundo!---------|'
```

Estructuras de datos

Tuplas

Existen varios tipos de estructuras de datos - objetos que contienen objetos. Ya vimos uno de ellos: el string, que contiene caracteres (en cambio, un `int` es solo un `int`), pero la estructura más sencilla es la *tupla*, que se define como un conjunto de objetos separados por coma, envueltos en paréntesis (aunque estos últimos son opcionales):

```
x = (y, z)
```

```
x

> (3.6, 'Hola mundo!')

type(x)

> <class 'tuple'>
```

Aquí hay dos elementos nuevos:

- Primero, hemos redefinido la variable `x`, cambiando su tipo, sin más.
- Además, acabamos de descubrir una característica de las tuplas (y de todas las estructuras en Python): sus elementos pueden ser de cualquier tipo, y no necesitan ser todos del mismo tipo.

Una vez definido, una tupla no puede ser modificada (es *inmutable*). Podemos acceder a los elementos de esta tupla tal como en strings (contando *desde cero*):

```
x[0]

> 3.6

x[1]

> 'Hola mundo!'
```

Algunas de las cosas que podemos hacer con una tupla es buscar (la primera ocurrencia de) un elemento:

```
x = (8, 4, 4, 7, 1, 0, 4, 3, 1)

x.index(7)

> 3

x.index(4)

> 1
```

o contar el número de ocurrencias:

```
x.count(1)

> 2
```

Ser inmutable hace de las tuplas estructuras de acceso más rápido que otros, y que ocupan menos espacio en memoria (en aplicaciones cotidianas esto no resulta muy importante).

Listas

La contraparte flexible (*mutable*) de la tupla es la **lista**:

```
t = [y, z]

type(t)

> <class 'list'>
```

Que sea mutable significa que se pueden modificar sus elementos:

```
t[0] = 9.9
```

no sólo de valor sino de tipo:

```
t[1] = -1

t

> [9.9, -1]
```

En una tupla, en cambio:

```
x[0] = -9

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

Hemos cometido otro *error de tipo*: a un *objeto de tipo tupla* no se le pueden asignar elementos. Encontraremos más errores a lo largo del curso; noten que el mensaje es explícito en el error que se ha cometido. Moraleja: *¡Leer los errores con atención!*

Más aún, podemos agregar elementos a la lista:

```
t.append(0)

t

> [9.9, -1, 0]
```

Noten que en la primera línea modificamos la variable **`t`** *en su lugar*; no fue necesario definir el resultado como una variable nueva. De hecho,

```
v = t.append('manzana')

v

>
```

La operación `append` no entrega ningún resultado (en el sentido que hemos estado hablando). Para ver esta variable es necesario imprimirla:

```
print(v)
```

```
> None
```

```
type(v)
```

```
> <class 'NoneType'>
```

La variable **None** es una variable vacía, pero es una variable como cualquier otra:

```
t.append(None)
```

```
t
```

```
> [9.9, -1, 0, 'manzana', None]
```

Aprovechemos que la lista es larga para demostrar cómo acceder a un rango de los datos, por índice:

```
t[1:3]
```

```
> [-1, 0]
```

```
t[:3]
```

```
> [9.9, -1, 0]
```

```
t[-2:] # igual a t[3:]
```

```
> ['manzana', None]
```

Con tuplas sería igual. Noten que el índice final del rango no se incluye en el resultado; en Python los rangos siempre están definidos como `[a,b)`.

También podemos eliminar elementos de una lista, tanto por índice:

```
t.pop(1)
```

```
t
```

```
> [9.9, 0, 'manzana', None]
```

como por elemento:

```
t.remove(None)
```

```
t
```

```
> [9.9, 0, 'manzana']
```

En este caso, al igual que `index`, se opera sobre la primera ocurrencia del elemento solicitado. La operación **`pop`** entrega el valor eliminado de la lista:

```
u = t.pop(1)

u

> 1

t

> [9.9, 'manzana']
```

Al igual que dos strings, tanto las tuplas como las listas pueden concatenarse fácilmente:

```
(1, 2) + (3, 4)

> (1, 2, 3, 4)

[1, 2] + [3, 4]

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

Pero no uno con otro:

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

> TypeError: can only concatenate list (not "tuple") to list
```

Para este mismo propósito podemos hacer:

```
t.extend([3, 4])

t

> [9.9, 'manzana', 3, 4]
```

Como ya dijimos, ambos tipos de objetos pueden tener elementos de cualquier tipo - incluidas listas y tuplas:

```
t.append([10, 11, 12])

t

> [9.9, 'manzana', [10, 11, 12]]

w = (y, z, [10, 11, 12])

w

> (3.6, 'Hola mundo!', [10, 11, 12])
```

Nota importante: como dice el Zen de Python, *explícito es mejor que implícito*, y *la legibilidad es importante*. Para estos ejemplos, hemos definido variables con

nombres de una sola letra. Sin embargo, recordar qué es cada variable se vuelve un reto. Es mejor darle a cada variable un nombre descriptivo. Por ejemplo, podríamos haber ejecutado

```
saludo = 'Hola mundo!'
```

Mucho mejor. Podemos hacer toda las operaciones tal como antes:

```
f'|{saludo:-^20s}|'
```

```
> '|----Hola mundo!-----|'
```

Indexación

Volvamos un segundo a la indexación. En detalle, ésta se define de la siguiente manera:

```
[comienzo:final:salto]
```

de manera que, por ejemplo:

```
x = [8, 4, 4, 7, 1, 0, 4, 3, 1]
```

```
x[1:8:3]
```

```
> [4, 1, 3]
```

Entrega los elementos segundo, quinto y octavo (índices 1, 4 y 7).

Sets

Un *set* es una estructura de datos únicos, *no estructurada*. Se puede definir de dos maneras:

```
set1 = set([3, 4, 8, 2])
```

```
set2 = {9, 3, 0, 1, 6, 5, 3}
```

```
type(set1), type(set2)
```

```
> (<class 'set'>, <class 'set'>)
```

Para inicializar un set vacío sólo sirve la primera opción,

```
vacio = set([])
```

Que no sea estructurado significa que los elementos no necesariamente se acceden en el orden que se definieron:

```
set1
```

```
> {8, 2, 3, 4}
```

y por lo tanto no pueden ser indexados:

```
set1[3]
```

```
> Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'set' object is not subscriptable
```

También dijimos que contiene datos únicos:

```
set2
```

```
> {0, 1, 3, 5, 6, 9}
```

Esto significa que los sets permiten varias operaciones adicionales. Primero, para encontrar los elementos únicos de una lista o tupla, basta convertirla a `set`:

```
x
```

```
> [8, 4, 4, 7, 1, 0, 4, 3, 1]
```

```
set(x)
```

```
{0, 1, 3, 4, 7, 8}
```

Podemos encontrar la unión de dos o más sets:

```
set1.union(set2)
```

```
> {0, 1, 2, 3, 4, 5, 6, 8, 9}
```

Lo que también podemos lograr con el operador `|` (“`OR`”):

```
set1 | set2
```

```
> {0, 1, 2, 3, 4, 5, 6, 8, 9}
```

su intersección:

```
set1 & set2 # tambien set1.intersection(set2)
```

```
> {3}
```

su diferencia:

```
set1 - set2 # tambien set1.difference(set2)
```

```
> {8, 2, 4}
```

```
set2 - set1 # tambien set2.difference(set1)

> {0, 1, 5, 6, 9}
```

su *diferencia simétrica* (el complemento de la intersección):

```
set1 ^ set2 # también set1.symmetric_difference(set2)

> {0, 1, 2, 4, 5, 6, 8, 9}
```

Además, podemos agregar elementos de a uno:

```
set1.add(0)

set1

> {0, 2, 3, 4, 8}
```

o de a varios:

```
set1.update(['a', 'b', 'c'])

set1

> {0, 2, 3, 4, 'b', 8, 'c', 'a'}
```

Y podemos eliminar objetos (esta vez sólo por valor):

```
set1.remove(4)

set1

> {0, 2, 3, 'b', 8, 'c', 'a'}
```

(El método `discard` funciona igual que `remove` pero si el elemento no existe ignora la operación en lugar de generar un `KeyError`.)

Los sets pueden contener cualquier tipo de objeto *inmutable*:

```
set1.add((3, 2))

set1

> {0, 2, 3, 4, 'b', 8, 'c', 'a', (3, 2)}
```

pero

```
set1.add([2, 0])

> Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Diccionarios

La última estructura de datos nativa es el *diccionario*. Este es un elemento definidor de Python: permite mapear nombres y variables libremente. Hay dos maneras equivalentes de definirlos:

```
diccionario = {'nombre': 'Andrea', 'apellido': 'Zapata', 'nota': 7.0, 'ranking': 1}

diccionario = dict(nombre='Andrea', apellido='Zapata', nota=7.0, ranking=1)
```

Los valores en un diccionario se pueden acceder a través de nombres (o *claves*, *keys*):

```
diccionario['nombre']

> 'Andrea'
```

y no a través de índices:

```
diccionario[1]

> KeyError: 1
```

Esta vez cometimos un *error de clave*: no existe la clave `1` en **`diccionario`**. Cometeríamos el mismo error si intentamos acceder cualquier otra clave que no existe:

```
diccionario['curso']

> KeyError: 'curso'
```

Ya aprenderemos cómo lidiar con estos errores. Por ahora podemos ahorrarnos estos errores usando el método **`get`**:

```
print(diccionario.get('curso'))

> None
```

Con **`get`**, si la clave no existe no se genera un error (lo que interrumpiría todo nuestro programa), si no que obtenemos **`None`**. De hecho, podemos especificar qué valor queremos obtener en caso de que la clave no exista:

```
diccionario.get('curso', -99)

> -99
```

Esto nos permite controlar de manera más fácil qué pasa cuando intentamos obtener una clave que no existe.

Podemos agregar elementos a un diccionario simplemente asignando a un índice que no existe:

```
diccionario['curso'] = 'Programacion'

diccionario

> {'nombre': 'Andrea', 'apellido': 'Zapata', 'nota': 7.0,
'ranking': 1, 'curso': 'Programacion'}
```

al igual que eliminar elementos:

```
diccionario.pop('ranking')

> 1

diccionario

> {'nombre': 'Andrea', 'apellido': 'Zapata', 'nota': 7.0,
'curso': 'Programacion'}
```

Al igual que las demás estructuras, los diccionarios pueden contener cualquier tipo de objeto:

```
diccionario['lista'] = [1, 2, 3, 4, 5]
```

Herramientas de ayuda

Si ya instalaron Anaconda pueden abrir el Navegador de Anaconda, en el que tendrán muchas opciones para elegir cómo seguir trabajando:

- **QtPython** es una terminal interactiva de IPython que permite ejecutar código al vuelo, con facilidades como auto-compleción (con `TAB`) y ayudas extendidas (ej. `?print`).
- **Jupyter notebook** es un *cuaderno* para escribir código en Python y otros lenguajes de programación (Julia, Python, R → Jupyter!). Incluye todas las facilidades de la terminal interactiva pero con una interfaz más amigable y a través de documentos que pueden guardarse en el computador para uso (incluyendo edición) posterior. A pesar de abrirse en el navegador de internet, toda la información que manera un cuaderno de Jupyter es local, y puede funcionar sin conexión a internet. Un detalle importante es que estos cuadernos son útiles para un análisis rápido, visualización de datos y otros procedimientos sencillos, pero no para programas complejos (pues nuestro

código tenderá a ser desordenado y poco prolijo). Pero por el momento sirve nuestro propósito.

- **JupyterLab** es una extensión de los cuadernos que permite abrir más de un cuaderno (y otros tipos de archivos) en una sola pestaña del navegador, además de muchas extensiones adicionales. JupyterLab cumple esencialmente la función de un IDE.
- **Spyder** es un IDE - un editor de texto inteligente. La primera vez que lo abran puede seguir la demostración para descubrir muchas de las opciones, que incluye auto compleción basado no solamente en las opciones de Python sino en su propio historial de escritura. En Spyder los códigos deben ser guardados como archivos .py, pero pueden ser ejecutados al vuelo mientras escribimos también.

En algunas clases más adelante utilizaremos JupyterLab (con los cuidados ya mencionados). Una de las ventajas, que ya usé como argumento para el uso de **`nano`** como editor de texto cotidiano, es la posibilidad de trabajar casi por completo con el teclado. Algunos comandos rápidos son:

```
Enter -> ingresar a la celda para escribir en ella

Esc -> salir de la celda

Shift+Enter -> Ejecutar celda

b -> nueva celda abajo (“below”)

a -> nueva celda arriba (“above”)

c, x, v -> copiar, cortar, pegar celda(s)

m -> convertir celda a texto Markdown

y -> convertir celda a código

dd -> eliminar celda

z -> deshacer operación de celda
```