

Tutorial de Python



Introducción

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

Historia

- Inventado en Holanda, principios de los 90 por Guido van Rossum
- El nombre viene del programa de televisión [Monty Python Flying Circus](#)
- Abierto desde el principio
- Considerado un lenguaje de scripting, pero es mucho más
- Escalable, orientado a objetos y funcional desde el principio
- Utilizado por Google desde el principio

Creador de Python

"Python es un experimento en cuánta libertad los programadores necesitan. Demasiada libertad y nadie puede leer el código de otro; Demasiada poca y la expresividad está en peligro."

- Guido van Rossum



El zen de Python

```
In [1]: 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!

- Hermoso es mejor que feo.
- Explicito es mejor que implícito.
- Simple es mejor que complejo.
- Complejo es mejor que complicado.
- El plano es mejor que el anidado.
- Dispersa es mejor que denso.
- La legibilidad es importante.
- Casos especiales no son lo suficientemente especiales para romper las reglas.
- Aunque la practicidad supera la pureza.
- Los errores nunca deben pasar en silencio.
- A menos que se silencie explícitamente.
- Ante la ambigüedad, rechaza la tentación de adivinar.
- Debería haber una - y preferiblemente sólo una - forma obvia de hacerlo.
- Aunque esa manera no puede ser obvia al principio a menos que usted es holandés.
- Ahora es mejor que nunca.
- Aunque nunca es a menudo mejor que *derecho* ahora.
- Si la implementación es difícil de explicar, es una mala idea.
- Si la implementación es fácil de explicar, puede ser una buena idea.
- Espacios de nombres son una buena idea - vamos a hacer más de esos!

Versiones de Python

Actualmente hay dos versiones de Python, 2.7 y 3.9. Python 3.0 introdujo muchos cambios incompatibles con la version 2.7 del lenguaje, por lo que el código escrito para 2.7 puede no funcionar bajo 3.9 y viceversa. Para esta clase todo el código utilizará Python 3.9. Puede comprobar su versión de Python en la línea de comandos ejecutando `python --version`.

Instalar Python

<https://www.anaconda.com/download/>



[Products](#) ▼

[Pricing](#)

[Solutions](#) ▼

[Resources](#) ▼

[Partners](#) ▼

[Blog](#)

[Company](#) ▼

[Contact Sales](#)

Individual Edition is now

ANACONDA DISTRIBUTION

The world's most popular open-source Python distribution platform

Anaconda Distribution

Download 

For Windows

Python 3.9 • 64-Bit Graphical Installer • 594 MB

Get Additional Installers



Introducción a Jupyter Notebooks y Python

En este tutorial, cubriremos:

- Jupyter: Creación de notebooks
- Python básico: Tipos de datos básicos (Contenedores, Listas, Diccionarios, Conjuntos, Tuplas), Funciones, Clases

Jupyter Notebooks

El *Jupyter Notebook* es una herramienta increíblemente poderosa para desarrollar y presentar proyectos de ciencia de datos de manera interactiva. Un *Notebook* integra el código y su salida en un único documento que combina visualizaciones, texto narrativo, ecuaciones matemáticas y otros medios. El flujo de trabajo intuitivo promueve el desarrollo iterativo y rápido, convirtiendo a los *Notebooks* en una opción cada vez más popular en la ciencia de datos contemporánea, el análisis y, cada vez más, la ciencia en general. Lo mejor de todo es que, como parte del proyecto de código abierto [Jupyter](#), son completamente gratuitos.

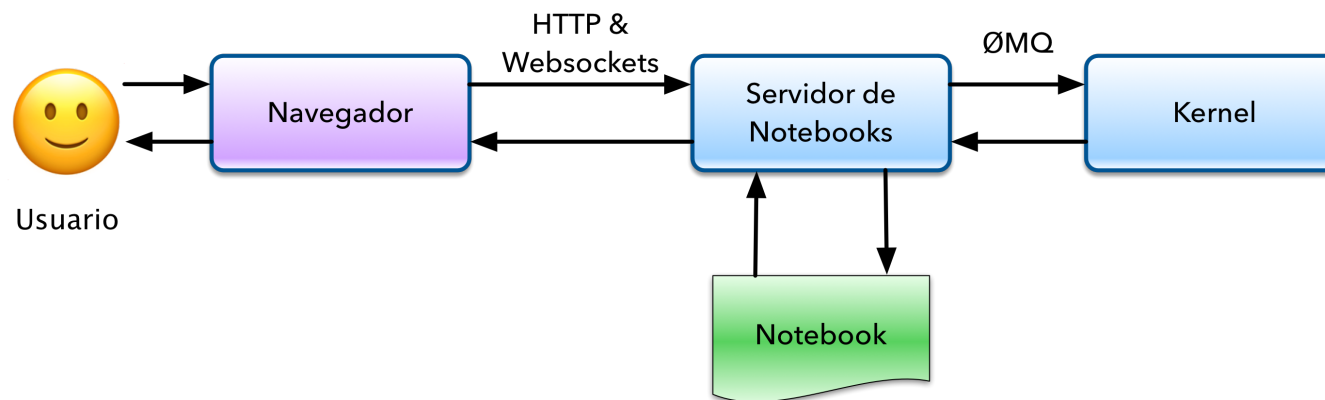
Jupyter Lab

JupyterLab es la próxima generación de los Notebook de Jupyter. Su objetivo es solucionar muchos problemas de usabilidad de los Notebooks, y amplía enormemente su alcance. JupyterLab ofrece un marco general para la computación interactiva y la ciencia de datos en el navegador, usando Python, Julia, R o uno de muchos otros lenguajes.


Además de proporcionar una interfaz mejorada para los Notebooks existentes, JupyterLab también incorpora dentro de la misma interfaz un explorador de archivos, consolas, terminales, editores de texto, editores Markdown, editores CSV, editores JSON, mapas interactivos, widgets, etc. La arquitectura es completamente extensible y está abierta a desarrolladores. En una palabra, JupyterLab es un IDE personalizable basado en la web para ciencia de datos y computación interactiva.

JupyterLab utiliza exactamente el mismo servidor y formato de archivo que el Jupyter Notebook clásico, por lo que es totalmente compatible con los Notebooks y kernels existentes. Los Notebook clásicos y JupyterLab pueden correr de un lado a otro en la misma computadora. Uno puede cambiar fácilmente entre las dos interfaces.

Arquitectura de JupyterLab



Teclas de acceso rápido

 jupyter 02-TutorialPython (autosaved)

File

Edit

View


Insert


Cell


Kernel


Widgets


Help





















 Run








Markdown

Teclas de acceso rápido


Keyboard shortcuts


The Jupyter Notebook has a cell and is indicated by a commands and is indicated


User Interface Tour


Keyboard Shortcuts 


Edit Keyboard Shortcuts


Notebook Help 


Markdown 


Python Reference 


IPython Reference 

NumPy Reference 

SciPy Reference 

Matplotlib Reference 

SymPy Reference 

pandas Reference 

About

The Jupyter Notebook has two different keyboard input modes. **Edit mode** allows you to type code or text into a cell and is indicated by a green cell border. **Command mode** binds the keyboard to notebook level commands and is indicated by a grey cell border with a blue left margin.

Command Mode (press `Esc` to enable)

[Edit Shortcuts](#)

`Alt-R` : Enter/Exit RISE Slideshow

`F` : find and replace

`Shift-B` : (un)set current cell as a Sub-slide cell

`Shift-C` : open the nbconfigurator page for RISE

`Shift-G` : (un)set current cell as a Fragment cell

`Shift-I` : (un)set current cell as a Slide cell

`Ctrl-Shift-F` : open the command palette

`Ctrl-Shift-P` : open the command palette

`Enter` : enter edit mode

`P` : open the command palette

`Shift-Enter` : run cell, select below

`Ctrl-Enter` : run selected cells

`Shift-Up` : extend selected cells above

`Shift-Down` : extend selected cells below

`Shift-J` : extend selected cells below

`Ctrl-A` : select all cells

`A` : insert cell above

`B` : insert cell below

`X` : cut selected cells

`C` : copy selected cells

`Shift-V` : paste cells above

`V` : paste cells below

`Z` : undo cell deletion

`D` , `D` : delete selected cells

`Shift-M` : merge selected cells, or current cell with cell below if only one cell is selected

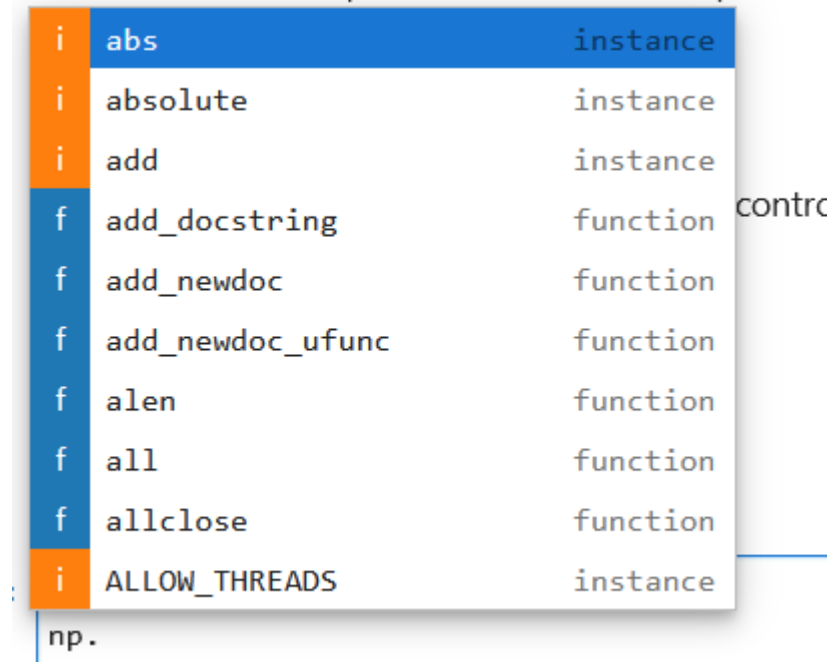
[Close](#)

Tecla TAB

```
In [3]: import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

```
In [ ]: # autocompletar con TAB
```

```
np.
```



i	abs	instance
i	absolute	instance
i	add	instance
f	add_docstring	function
f	add_newdoc	function
f	add_newdoc_ufunc	function
f	alen	function
f	all	function
f	allclose	function
i	ALLOW_THREADS	instance

np.

Shift TAB

```
In [ ]: # despliega información sobre los parámetros de la función
np.linspace()
```

```
[ ]: # despliega información sobre los parámetros de la función
np.linspace()
```

Signature:

```
np.linspace(
    start,
    stop,
    num=50,
    endpoint=True,
    retstep=False,
    dtype=None,
    axis=0,
)
```

Docstring:

Return evenly spaced numbers over a specified interval.

Returns `num` evenly spaced samples, calculated over the interval [`start`, `stop`].

The endpoint of the interval can optionally be excluded.

.. versionchanged:: 1.16.0

Non-scalar `start` and `stop` are now supported.

Teclas de ayuda ? y ??

```
In [ ]: # despliega la ayuda sobre la función
np.linspace?
```

```
[2]: # despliega la ayuda sobre la función
np.linspace?
```

Signature:

```
np.linspace(
    start,
    stop,
    num=50,
    endpoint=True,
    retstep=False,
    dtype=None,
    axis=0,
)
```

Docstring:

Return evenly spaced numbers over a specified interval.

Returns `num` evenly spaced samples, calculated over the interval [`start`, `stop`].

The endpoint of the interval can optionally be excluded.

.. versionchanged:: 1.16.0

Non-scalar `start` and `stop` are now supported.

.. versionchanged:: 1.20.0

Values are rounded towards ``-inf`` instead of ``0`` when an integer ``dtype`` is specified. The old behavior can still be obtained with ``np.linspace(start, stop, num).astype(int)``

Parameters

```
In [ ]: ## despliega la ayuda y el código de la función
np.linspace??
```

```
[3]: ## despliega la ayuda y el código de la función  
np.linspace??
```

Signature:

```
np.linspace(  
    start,  
    stop,  
    num=50,  
    endpoint=True,  
    retstep=False,  
    dtype=None,  
    axis=0,  
)
```

Source:

```
@array_function_dispatch(_linspace_dispatcher)  
def linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None,  
             axis=0):
```

```
    """
```

Return evenly spaced numbers over a specified interval.

Returns `num` evenly spaced samples, calculated over the interval [`start`, `stop`].

The endpoint of the interval can optionally be excluded.

```
.. versionchanged:: 1.16.0
```

Non-scalar `start` and `stop` are now supported.

```
.. versionchanged:: 1.20.0
```

Values are rounded towards ```-inf``` instead of ```0``` when an integer `dtype` is specified. The old behavior can still be obtained with ```np.linspace(start, stop, num).astype(int)```

Parameters

Magics

- % → inline magic
- %% → cell magic

In [7]: `%lsmagic`

Out[7]: Available line magics:
%alias %alias_magic %autoawait %autocall %automagic %autosave %bookmark %cd %clear %cls %colors %conda %config %connect_info %copy %ddir %debug %dhist %dirs %doctest_mode %echo %ed %edit %env %gui %hist %history %killbgscripts %ldir %less %load %load_ext %loadpy %logoff %logon %logstart %logstate %logstop %ls %lsmagic %macro %magic %matplotlib %mkdir %more %notebook %page %pastebin %pdb %pdef %pdoc %pfile %pinfo %pinfo2 %pip %popd %pprint %precision %prun %ps %psource %pushd %pwd %pycat %pylab %qtconsole %quickref %recall %rehashx %reload_ext %ren %rep %rerun %reset %reset_selective %rmdir %run %save %sc %set_env %store %sx %system %tb %time %timeit %unalias %unload_ext %who %who_ls %whos %xdel %xmode

Available cell magics:
%! %HTML %%SVG %%bash %%capture %%cmd %%debug %%file %%html %%javascript %js %%latex %%markdown %%perl %%prun %ppypy %%python %%python2 %%python3 %%ruby %%script %%sh %%svg %%sx %%system %%time %%timeit %%writefile

Automagic is ON, % prefix IS NOT needed for line magics.

Latex

In [12]: `%%latex`

Ejemplo cell magic...

```
\begin{equation}
\oint_S \{E_n dA = \frac{1}{\epsilon_0} Q_{\text{inside}}\}
\end{equation}
```

Ejemplo cell magic...

$$\oint_S E_n dA = \frac{1}{\epsilon_0} Q_{\text{inside}} \quad (1)$$

Ejecutar Comandos en el *shell*, usando la tecla !

In [11]: `!dir`

Volume in drive C has no label.
Volume Serial Number is 34B3-8AA6

Directory of C:\Users\wladi\Documents\Cursos\Aprendizaje-Automatico-2022\notebooks

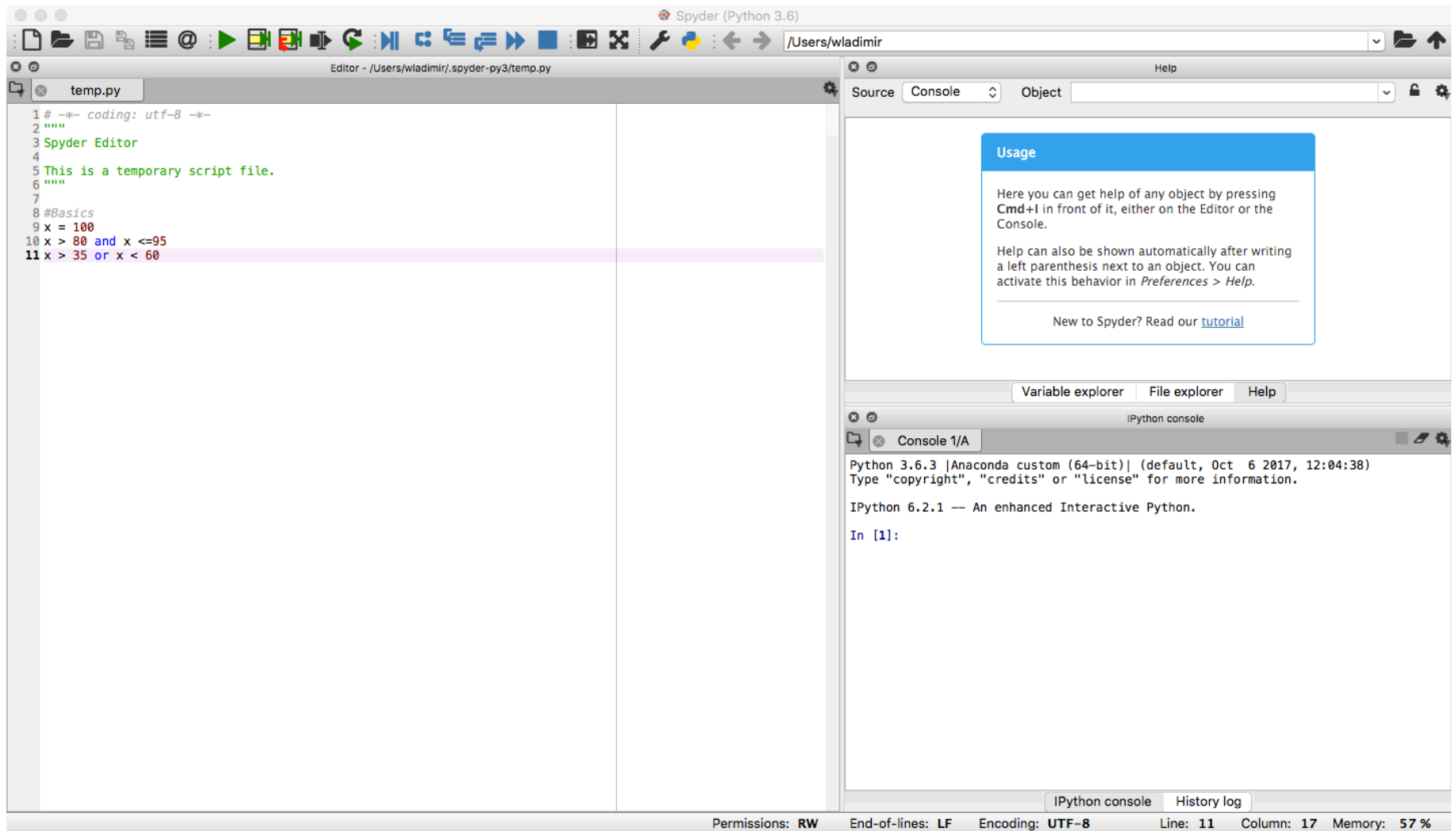
```
08/06/2022  09:07 a.ÿm.    <DIR>          .
07/06/2022  08:34 a.ÿm.    <DIR>          ..
07/06/2022  09:21 a.ÿm.    <DIR>          .ipynb_checkpoints
07/06/2022  08:57 a.ÿm.      154.879 01-Introduccion_Aprendizaje_Autom tico.ipynb
08/06/2022  09:07 a.ÿm.      87.034 02-TutorialPython.ipynb
31/05/2022  10:29 a.ÿm.      845 Modelo_Diabetes.pkl
           3 File(s)        242.758 bytes
           3 Dir(s)  285.904.863.232 bytes free
```

IDE para Python

¿Qué editor usar?

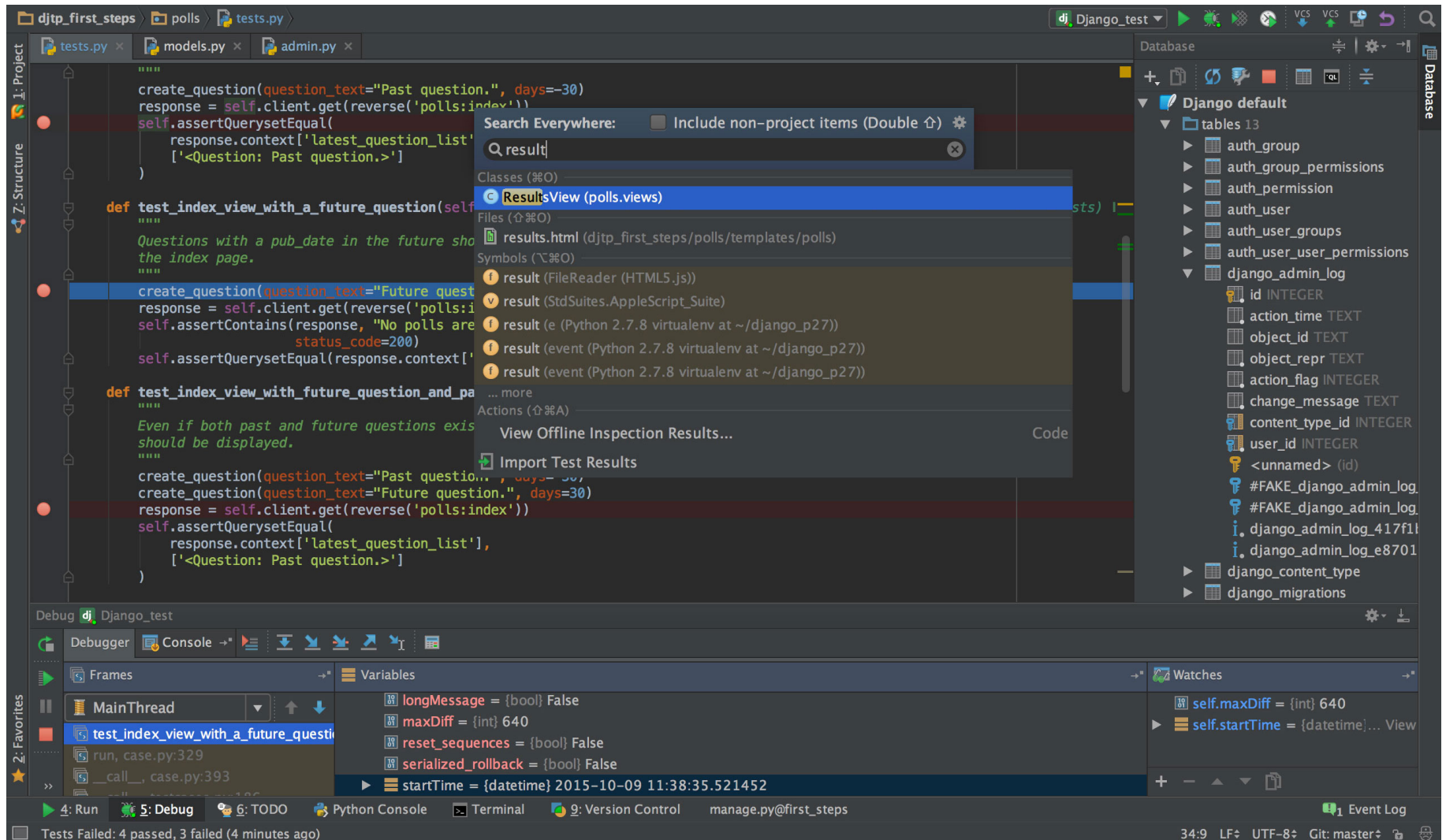
Python no exige un editor específico y hay muchos modos y maneras de programar.

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

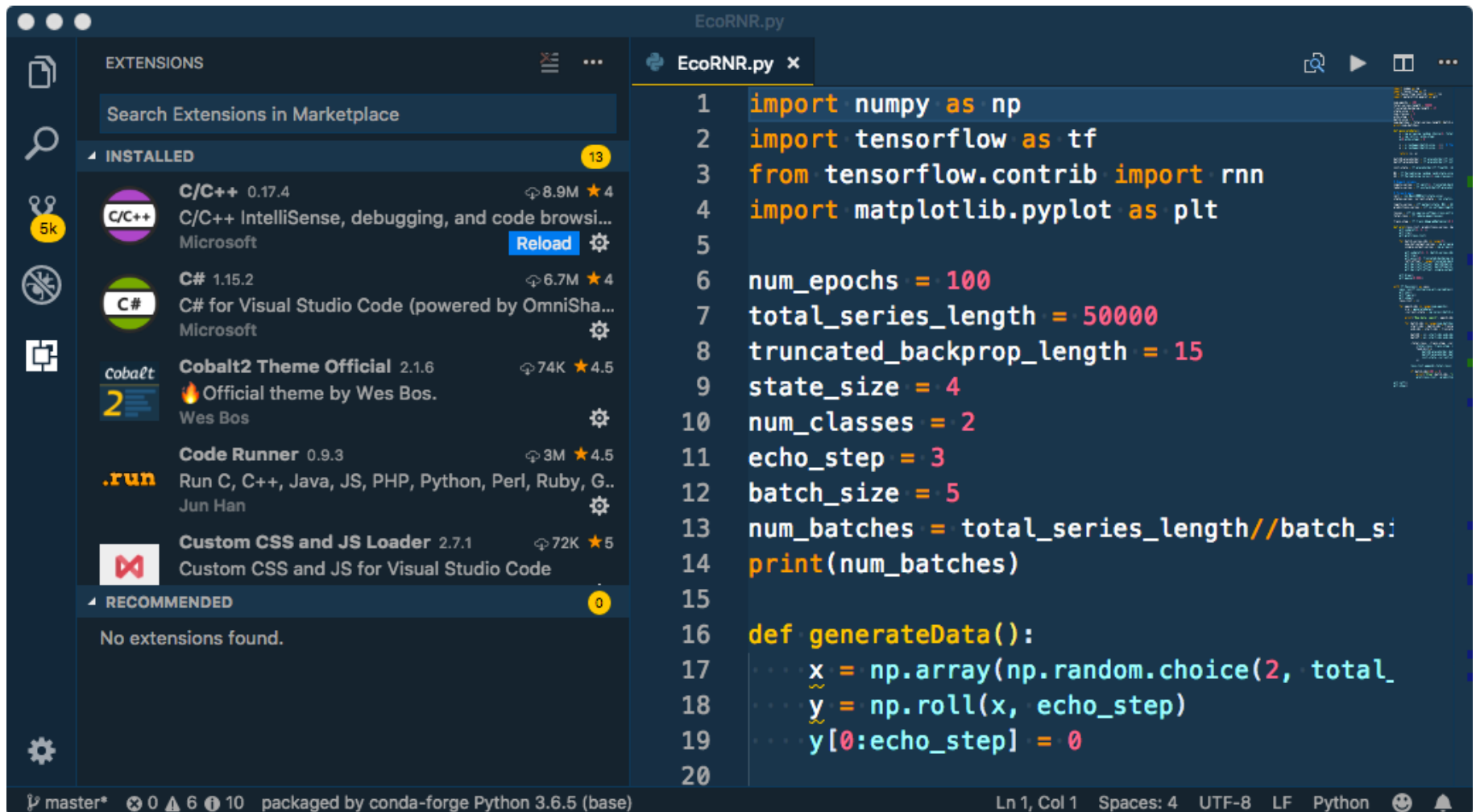


In []:

También existe un IDE para python llamado **PyCharm**



También se puede utilizar [Visual Studio Code](#). En el proceso de instalación de Anaconda preguntará si se quiere instalar también el Visual Studio Code



Python es un lenguaje de programación multiparadigm de alto nivel, de tipo dinámico. El código de Python a menudo se dice que es casi como pseudocódigo, ya que le permite expresar ideas muy poderosas en muy pocas líneas de código mientras que sea muy legible. Como ejemplo, aquí está una implementación del algoritmo clásico de quicksort en Python:

```
In [2]: def quicksort(arreglo):
        if len(arreglo) <= 1:
            return arreglo
        pivote = arreglo[int(len(arreglo) / 2)]
        izquierda = [x for x in arreglo if x < pivote]
        medio = [x for x in arreglo if x == pivote]
        derecha = [x for x in arreglo if x > pivote]
        return quicksort(izquierda) + medio + quicksort(derecha)

print (quicksort([3,6,8,10,1,2,1]))
```

```
[1, 1, 2, 3, 6, 8, 10]
```

Tipos de datos básicos

Números

Los números enteros y los punto flotante funcionan como se esperaría de otros lenguajes:

```
In [8]: x = 3
        print(x)
        type(x)

3
<class 'int'>
```

```
In [9]: print (x + 1)  # Suma;
        print (x - 1)  # Resta;
        print (x * 2)  # Multiplicación;
        print (x // 2) # División entera;
        print (x / 2)  # División punto flotante;
        print (x % 2)  # Modulo;
        print (x ** 2) # Exponenciación;

4
2
6
1
1.5
1
9
```

```
In [10]: x += 1
         print (x)  # Imprime "4"
         x *= 2
         print (x)  # Imprime "8"
```

4
8

```
In [14]: y = 2.5
print (type(y)) # Imprime "<class 'float'"
print (y, y + 1, y * 2, y ** 2) # Imprime "2.5 3.5 5.0 6.25"

<class 'float'>
2.5 3.5 5.0 6.25
1.4
```

```
In [16]: (3 + 2j) + (5 - 2j)
```

```
Out[16]: (8+0j)
```

Tenga en cuenta que a diferencia de muchos idiomas, Python no tiene operadores unarios de incremento (x++) o decremento (x--).

Python también tiene tipos incorporados para enteros largos y números complejos; Puede encontrar todos los detalles en la [documentación](#).

Booleanos

Python implementa todos los operadores habituales para la lógica booleana, pero usa palabras en inglés en lugar de símbolos (`&&` , `||` , etc.):

```
In [17]: t, f = True, False
print (type(t)) # Imprime "<class 'bool'"

<class 'bool'>
```

Ahora veamos las operaciones:

```
In [18]: print (t and f) # Y Lógico;
print (t or f)  # O Lógico;
print (not t)   # NO Lógico;
print (t != f)  # XOR Lógico;

False
True
False
True
```

Cadena de caracteres (Strings)

```
In [19]: hola = 'hola' # Los literales de cadenas pueden usar comillas simples
mundo = "mundo" # o comillas dobles; no importa.
print (hola, len(hola))

hola 4
```

```
In [20]: hm = hola + ' ' + mundo # Concatenación de cadenas
print (hm) # imprime "hola mundo"
```

hola mundo

```
In [21]: hm12 = '%s %s %d' % (hola, mundo, 12) # formatos de cadena estilo sprintf
print (hm12) # imprime "hola mundo 12"
```

hola mundo 12

Los objetos de cadena tienen un montón de métodos útiles; por ejemplo:

```
In [22]: s = "hola"
print (s.capitalize()) # Capitalizar una cadena; Imprime "Hola"
print (s.upper())      # Convertir una cadena en mayúsculas; Imprime "HOLA"
print (s.rjust(7))     # Justificar a la derecha una cadena, relleno con espacios; Imprime "  hola"
print (s.center(7))    # Centrar una cadena, relleno con espacios; Imprime " hola "
print (s.replace('l', '(ell)')) # Reemplazar todas las instancias de una subcadena con otra;
                                # Imprime "ho(ell)a"
print (' mundo '.strip()) # Elimina los espacios en blanco al inicio y al final; Imprime "mundo"
```

Hola
HOLA
 hola
 hola
ho(ell)a
mundo

Puede encontrar una lista de todos los métodos de cadenas en la [documentación](#).

```
In [1]: l=[]
for i in range(2000, 3201):
    if (i%7==0) and (i%5!=0):
        l.append(str(i))

print(','.join(l))
```

2002,2009,2016,2023,2037,2044,2051,2058,2072,2079,2086,2093,2107,2114,2121,2128,2142,2149,2156,2163,2177,2184,2191,2198,2212,2219,2226,2233,2247,2254,2261,2268,2282,2289,2296,2303,2317,2324,2331,2338,2352,2359,2366,2373,2387,2394,2401,2408,2422,2429,2436,2443,2457,2464,2471,2478,2492,2499,2506,2513,2527,2534,2541,2548,2562,2569,2576,2583,2597,2604,2611,2618,2632,2639,2646,2653,2667,2674,2681,2688,2702,2709,2716,2723,2737,2744,2751,2758,2772,2779,2786,2793,2807,2814,2821,2828,2842,2849,2856,2863,2877,2884,2891,2898,2912,2919,2926,2933,2947,2954,2961,2968,2982,2989,2996,3003,3017,3024,3031,3038,3052,3059,3066,3073,3087,3094,3101,3108,3122,3129,3136,3143,3157,3164,3171,3178,3192,3199

Contenedores

Python incluye varios tipos de contenedores: listas, diccionarios, conjuntos y tuplas.

Listas

Una lista es el equivalente de Python de una matriz, pero es redimensionable y puede contener elementos de diferentes tipos:

```
In [78]: xs = [3, 1, 2]    # Crear una lista
print (xs, xs[2])
print (xs[-1])           # Los índices negativos cuentan desde el final de la lista; Imprime "2"
```

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

```
In [82]: xs[2] = 'mundo'   # Las listas pueden contener elementos de diferentes tipos
print (xs)
```

```
[3, 1, 'mundo']
```

```
In [86]: xs.append('hola') # Añadir un nuevo elemento al final de la lista
print (xs)
```

```
[3, 1, 'mundo', 'hola', 'hola', 'hola']
```

```
In [40]: x = xs.pop(1)     # Elimina y devuelve el último elemento de la lista
print (x, xs)
```

```
1 [3, 'mundo']
```

Como de costumbre, usted puede encontrar todos los detalles sobre listas en la [documentación](#).

Rebanado

Además de acceder a los elementos de la lista de uno en uno, Python proporciona sintaxis concisa para acceder a las sublistas; Esto se conoce como rebanar:

```
In [88]: nums = [0, 1, 2, 3, 4]
print (nums)           # Imprime "[0, 1, 2, 3, 4]"
print (nums[2:4])       # Obtener una porción del índice 2 al 4 (exclusivo); imprime "[2, 3]"
print (nums[2:])         # Obtener una porción del índice 2 hasta el final; imprime "[2, 3, 4]"
print (nums[:2])         # Obtener una porción desde el principio hasta el índice 2 (exclusivo); imprime "[0, 1]"
print (nums[:])          # Obtener una porción de toda la lista; imprime "[0, 1, 2, 3, 4]"
print (nums[:-1])        # Los índices de las porciones pueden ser negativos; imprime "[0, 1, 2, 3]"
nums[2:4] = [8, 9]       # Asignar una nueva sublista a una porción
print (nums)            # Imprime "[0, 1, 8, 9, 4]"
```

```
[0, 1, 2, 3, 4]
[2, 3]
[2, 3, 4]
[0, 1]
[0, 1, 2, 3, 4]
[0, 1, 2, 3]
[0, 1, 8, 9, 4]
```

```
In [91]: nums[0] = -1
nums
```

```
Out[91]: [-1, 1, 8, 9, 4]
```

Lazos

Puedes realizar un lazo sobre los elementos de una lista de esta forma:

```
In [42]: animales = ['gato', 'perro', 'mono']
for animal in animales:
    print (animal)
```

```
gato
perro
mono
```

Si desea tener acceso al índice de cada elemento dentro del cuerpo de un lazo, utilice la función `enumerate` :

```
In [47]: animales = ['gato', 'perro', 'mono']
for idx, animal in enumerate(animales):
    print ('#%d: %s' % (idx + 1, animal))
```

```
#1: gato
1
2
3
#2: perro
1
2
3
#3: mono
1
2
3
```

Listas por comprensión:

Al programar, con frecuencia queremos transformar un tipo de datos en otro. Como ejemplo simple, considere el siguiente código que calcula los números cuadrados:

```
In [44]: nums = [0, 1, 2, 3, 4]
cuadrados = []
for x in nums:
    cuadrados.append(x ** 2)
print (cuadrados)
```

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

Puede simplificar este código utilizando una listas por comprensión:

```
In [48]: nums = [0, 1, 2, 3, 4]
cuadrados = [x ** 2 for x in nums]
print (cuadrados)
```

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

Listas por comprensión puede tambien contener condiciones:

```
In [49]: nums = [0, 1, 2, 3, 4]
cuadrados_pares = [x ** 2 for x in nums if x % 2 == 0]
print (cuadrados_pares)
```

```
[0, 4, 16]
```

Diccionarios

Un diccionario almacena pares (clave, valor) parecidos a un Mapa en Java o un objeto en Javascript. Puedes usarlo así:

```
In [54]: d = {'gato': 'lindo', 'perro': 'peludo'} # Crear un nuevo diccionario con algunos datos
print (d['gato']) # Obtener una entrada de un diccionario; imprime "lindo"
print (d) # Compruebe si un diccionario tiene una clave dada; imprime "True"
```

```
lindo
{'gato': 'lindo', 'perro': 'peludo'}
```

```
In [58]: d['pez'] = 'mojado' # Agrega o modifica una entrada en un diccionario
print (d['pez'])# Imprime "mojado"
print (d)
d['pez'] = 'seco'
print (d)
```

```
mojado
{'gato': 'lindo', 'pez': 'mojado', 'perro': 'peludo'}
{'gato': 'lindo', 'pez': 'seco', 'perro': 'peludo'}
```

```
In [59]: print (d['mono']) # KeyError: 'mono' no es una clave de d
```

```
-----  
KeyError                                Traceback (most recent call last)  
<ipython-input-59-5f1b4e95498d> in <module>()  
----> 1 print (d['mono']) # KeyError: 'mono' no es una clave de d  
  
KeyError: 'mono'
```

```
In [61]: print (d.get('mono', 'Esa clave no existe')) # Obtener un elemento con un valor predeterminado; imprime "N / A"  
print (d.get('pez', 'N/A')) # Obtener un elemento con un valor predeterminado; imprime "mojado"
```

Esa clave no existe
seco

```
In [62]: del d['pez'] # Eliminar un elemento de un diccionario  
print (d.get('pez', 'N/A')) # "pez" ya no es una clave; imprime "N / A"
```

N/A

Puedes encontrar todo lo que necesitas saber sobre diccionarios en la [documentación](#).

Es fácil de iterar sobre las claves en un diccionario:

```
In [63]: d = {'pajaro': 2, 'gato': 4, 'araña': 8}  
for animal in d:  
    patas = d[animal]  
    print ('Un %s tiene %d patas' % (animal, patas))
```

Un araña tiene 8 patas
Un gato tiene 4 patas
Un pajaro tiene 2 patas

Si desea acceder a las claves y sus valores correspondientes, utilice el método items:

```
In [64]: d = {'pajaro': 2, 'gato': 4, 'araña': 8}  
for animal, patas in d.items():  
    print ('Un %s tiene %d patas' % (animal, patas))
```

Un araña tiene 8 patas
Un gato tiene 4 patas
Un pajaro tiene 2 patas

Diccionario por comprensión: Estos son similares a la lista por comprensión, pero le permiten construir fácilmente diccionarios. Por ejemplo:

```
In [65]: nums = [0, 1, 2, 3, 4]  
cuadrados_numeros_pares = {x: x ** 2 for x in nums if x % 2 == 0}  
print (cuadrados_numeros_pares)
```



```
{0: 0, 2: 4, 4: 16}
```

Conjuntos

Un conjunto es una colección desordenada de elementos distintos. Como ejemplo simple, considere lo siguiente:

```
In [66]: animales = {'gato', 'perro'}
print ('gato' in animales)  # Compruebe si un elemento está en un conjunto; imprime "True"
print ('pez' in animales)   # imprime "False"

True
False
```

```
In [67]: animales.add('pez')      # Agregar un elemento a un conjunto
print ('pez' in animales)
print (len(animales))           # Número de elementos en un conjunto;

True
3
```

```
In [68]: animales.add('gato')     # Agregar un elemento que ya está en el conjunto no hace nada
print (len(animales))
animales.remove('gato')         # Eliminar un elemento de un conjunto
print (len(animales))

3
2
```

Lazos: Iterar sobre un conjunto tiene la misma sintaxis que iterar sobre una lista; Sin embargo, puesto que los conjuntos están desordenados, no se pueden hacer suposiciones acerca del orden en el que se visitan los elementos del conjunto:

```
In [69]: animales = {'gato', 'perro', 'pez'}
for idx, animal in enumerate(animales):
    print ('#%d: %s' % (idx + 1, animal))
# Imprime "#1: pez", "#2: gato", "#3: perro"

#1: gato
#2: pez
#3: perro
```

Conjunto por comprensión: Al igual que las listas y los diccionarios, podemos construir fácilmente conjuntos utilizando conjuntos dpor comprensión:

```
In [70]: from math import sqrt
print ({int(sqrt(x)) for x in range(30)})

{0, 1, 2, 3, 4, 5}
```

Tuplas

Una tupla es una lista ordenada (inmutable) de valores. Una tupla es en muchos aspectos similar a una lista; Una de las diferencias más importantes es que las tuplas se pueden utilizar como claves en diccionarios y como elementos de conjuntos, mientras que las listas no pueden. Aquí hay un ejemplo trivial:

```
In [71]: d = {(x, x + 1): x for x in range(10)} # Crear un diccionario con claves de tupla
t = (5, 6) # Crear una tupla
print (type(t))
print (d)
print (d[(1, 2)])

<class 'tuple'>
{(0, 1): 0, (1, 2): 1, (5, 6): 5, (2, 3): 2, (4, 5): 4, (6, 7): 6, (8, 9): 8, (9, 10): 9, (3, 4): 3, (7, 8): 7}
1
```

```
In [72]: t[0] = 1
```

```
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-72-0a69537257d5> in <module>()
----> 1 t[0] = 1

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

Funciones

Las funciones de Python se definen mediante la palabra clave `def`. Esta es la sintaxis básica:

```
def NombreFuncion(arg1, arg2,... argN):

    ''' Documentación'''

    sentencias

    return <valor>
```

```
In [73]: def signo(x):  
        if x > 0:  
            return 'positivo'  
        elif x < 0:  
            return 'negativo'  
        else:  
            return 'cero'  
  
        for x in [-1, 0, 1]:  
            print (signo(x))
```

```
negativo  
cero  
positivo
```

Definiremos a menudo las funciones para que tomen argumentos opcionales, como esto:

```
In [74]: def hola(nombre, gritar=False):  
        if gritar:  
            print ('HOLA, %s' % nombre.upper())  
        else:  
            print ('Hola, %s!' % nombre)  
  
        hola('Juan')  
        hola('Manuel', gritar=True)
```

```
Hola, Juan!  
HOLA, MANUEL
```

Si no se conoce el número de argumentos que una función debe aceptar, entonces se usa un símbolo de asterisco antes del argumento

```
In [2]: def sumar_n(*args):  
        res = 0  
        reslist = []  
        for i in args:  
            reslist.append(i)  
        print(reslist)  
        return sum(reslist)
```

```
In [3]: sumar_n(1,2,3,4,5)
```

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

```
Out[3]: 15
```

Funciones lambda

Estas son funciones pequeñas que no están definidas con ningún nombre y llevan una sola expresión cuyo resultado se devuelve. Las funciones de Lambda son muy útiles cuando se opera con listas. Estas funciones están definidas por la palabra clave lambda seguida de las variables, dos puntos y la expresión respectiva.

```
In [4]: lambda x: x * x
```

```
Out[4]: <function __main__.<lambda>(x)>
```

Las funciones lambda se pueden invocar en el momento de definir las:

```
In [6]: (lambda x: x * x) (5)
```

```
Out[6]: 25
```

También se pueden asignar a una variable:

```
In [8]: z = lambda x: x * x  
z(8)
```

```
Out[8]: 64
```

Función map

La función `map()` básicamente ejecuta la función que se define para cada elemento de la lista por separado.

```
In [12]: lista1 = [1,2,3,4,5,6,7,8,9]  
eg = map(lambda x:x+2, lista1)  
list(eg)
```

```
Out[12]: [3, 4, 5, 6, 7, 8, 9, 10, 11]
```

```
In [13]: lista2 = [9,8,7,6,5,4,3,2,1]  
eg2 = map(lambda x,y:x+y, lista1,lista2)  
list(eg2)
```

```
Out[13]: [10, 10, 10, 10, 10, 10, 10, 10, 10]
```

Función `filter`

La función `filter()` se utiliza para filtrar los valores en una lista. Tenga en cuenta que la función `filter()` devuelve el resultado como un objeto `filter`.

Obtener los elementos menores que 5:

```
In [19]: filter(lambda x:x<5,lista1)
```

```
Out[19]: <filter at 0x11495cf28>
```

Para obtener la lista se debe pasar el objeto `filter` a la función `list()`

```
In [20]: list(filter(lambda x:x<5,lista1))
```

```
Out[20]: [1, 2, 3, 4]
```

Iteradores y Generadores

En Python, todo sobre lo que se puede iterar se llama iterable:

```
In [21]: tazon = {  
    "manzana": 5,  
    "cambur": 3,  
    "naranja": 7  
}  
  
for fruta in tazon:  
    print(fruta.upper())
```

```
MANZANA  
CAMBUR  
NARANJA
```

Muy a menudo, queremos iterar sobre algo que requiere una cantidad moderadamente grande de memoria para almacenar

Iterador

Considere la función básica `range` de Python:

```
In [22]: range(10)
range(0, 10)
total = 0
for x in range(int(1e6)):
    total += x

total
```

Out[22]: 499999500000

Para evitar la asignación de un millón de enteros, `range` realmente utiliza un iterador.

En realidad, no necesitamos un millón de números enteros a la vez, solo cada número entero a su vez hasta un millón.

Debido a que podemos obtener un iterador, decimos que un rango es iterable.

Entonces podemos hacer un bucle sobre él:

```
In [23]: for i in range(3):
        print(i)
```

```
0
1
2
```

Una vez que tenemos un objeto iterador, podemos pasarlo a la función `next`. Esto mueve el iterador hacia adelante y nos da su siguiente elemento:

```
In [25]: a = iter(range(3))
```

```
In [26]: type(a)
```

Out[26]: range_iterator

```
In [27]: next(a)
```

Out[27]: 0

```
In [28]: next(a)
```

Out[28]: 1

```
In [29]: next(a)
```

Out[29]: 2

```
In [30]: next(a)
```

```
-----  
StopIteration                                Traceback (most recent call last)  
<ipython-input-30-15841f3f11d4> in <module>  
----> 1 next(a)  
  
StopIteration:
```

Esto le dice a Python que la iteración ha terminado. Por ejemplo, si estamos en un bucle `for i in range(3)`, esto nos permite saber cuándo debemos salir del bucle.

Podemos convertir un iterable o iterador en una lista con el constructor de listas `list()`:

```
In [31]: list(range(5))
```

```
Out[31]: [0, 1, 2, 3, 4]
```

Definir nuestro propio iterable

Cuando escribimos el `next(a)`, internamente Python intenta llamar al método `__next__()` de `a`. Del mismo modo, `iter(a)` llama a `a.__iter__()`.

Podemos hacer nuestros propios iteradores definiendo clases que se puedan usar con las funciones `next()` e `iter()`: este es el protocolo del iterador.

Para cada uno de los conceptos en Python, como secuencia, contenedor, iterable, el lenguaje define un protocolo, un conjunto de métodos que una clase debe implementar para ser tratado como un miembro de ese concepto.

Para definir un iterador, los métodos que deben admitirse son `__next__()` y `__iter__()`.

`__next__()` debe actualizar el iterador.

Veremos por qué necesitamos definir `__iter__` en un momento.

Aquí hay un ejemplo de cómo definir una clase de iterador personalizado:

```
In [39]: class fib_iterator:
        """Un iterador sobre una parte de la secuencia de Fibonacci."""

        def __init__(self, limite, semilla1=1, semilla2=1):
            self.limite = limite
            self.previo = semilla1
            self.actual = semilla2

        def __iter__(self):
            return self

        def __next__(self):
            (self.previo, self.actual) = (self.actual, self.previo + self.actual)
            self.limite -= 1
            if self.limite < 0:
                raise StopIteration()
            return self.actual
```

```
In [40]: x = fib_iterator(5)
        next(x)
```

Out[40]: 2

```
In [41]: next(x)
```

Out[41]: 3

```
In [42]: next(x)
```

Out[42]: 5

```
In [43]: for x in fib_iterator(5):
        print(x)
```

2
3
5
8
13

```
In [44]: sum(fib_iterator(1000))
```

Out[44]: 2979242185081433603368828199816319009156731305438197590327781734405367221904889045200345081638463455390550965338859432428149784690
42830417586260359446115245634668393210192357419233828310479227982326069668668250

Generadores

Hay una buena cantidad de código repetitivo en la definición basada en la clase anterior de un iterable.

Python proporciona otra forma de especificar algo que cumple con el protocolo `iterator` : generadores.

```
In [45]: def mi_generador():  
        yield 5  
        yield 10
```

```
In [46]: x = mi_generador()  
        next(x)
```

```
Out[46]: 5
```

```
In [47]: next(x)
```

```
Out[47]: 10
```

```
In [48]: next(x)
```

```
-----  
StopIteration                                Traceback (most recent call last)  
<ipython-input-48-92de4e9f6b1e> in <module>  
----> 1 next(x)  
  
StopIteration:
```

```
In [49]: for a in mi_generador():  
        print(a)
```

```
5  
10
```

```
In [50]: sum(mi_generador())
```

```
Out[50]: 15
```

Una función que tiene sentencias `yield` en lugar de una sentencia de `return` regresa temporalmente: se convierte automáticamente en algo que implementa `__next__` .

Cada llamada de `next()` devuelve el control a la función donde la dejó.

El control pasa de un lado a otro entre el generador y la persona que llama. Nuestro ejemplo de Fibonacci, por lo tanto, se convierte en una función más que en una clase.

```
In [51]: def generador_fibs(limite, semilla1=1, semilla2=1):  
        actual = semilla1  
        previo = semilla2  
  
        while limite > 0:  
            limite -= 1  
            actual, previo = actual + previo, actual  
            yield actual
```

```
In [52]: sum(generador_fibs(5))
```

```
Out[52]: 31
```

```
In [53]: for a in generador_fibs(10):  
        if a % 2 == 0:  
            print(a)
```

```
2  
8  
34  
144
```

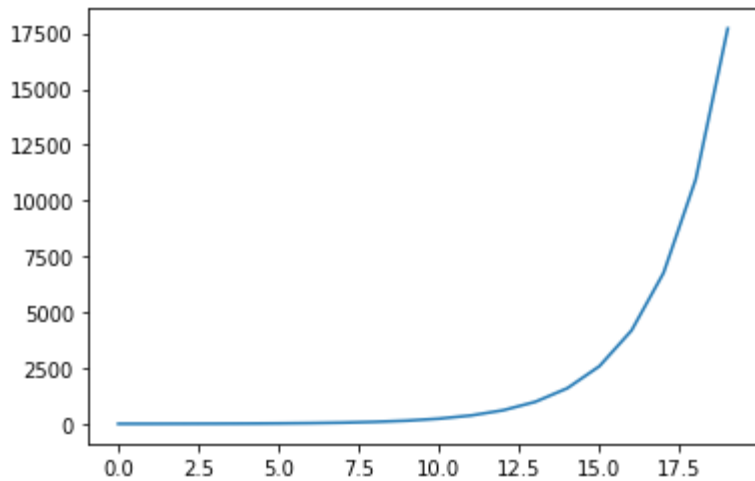
A veces es posible que necesitemos reunir todos los valores de un generador en una lista, como antes de pasarlos a una función que espera una lista:

```
In [54]: list(generador_fibs(10))
```

```
Out[54]: [2, 3, 5, 8, 13, 21, 34, 55, 89, 144]
```

```
In [58]: %matplotlib inline  
import matplotlib.pyplot as plt  
  
plt.plot(list(generador_fibs(20)))
```

```
Out[58]: [<matplotlib.lines.Line2D at 0x11c141a90>]
```



Clases

Para definir una clase en la programación de Python, utilizamos la palabra clave `class`. Es como si usáramos `def` para definir una función en Python. Y como una función, una clase Python3 también puede tener una cadena de documentación.

```
In [59]: class fruta:
        '''Esta clase crea instancias de frutas'''
        pass
```

```
In [60]: fruta
```

```
Out[60]: __main__.fruta
```

Una clase Python3 puede tener atributos y métodos para ejecutarse en esos datos.

```
In [61]: class fruta:
        '''Esta clase crea instancias de frutas'''
        color = ''
        def decir_hola(self):
            print('Hola')
```

```
In [63]: naranja = fruta()
        naranja
```

```
Out[63]: <__main__.fruta at 0x11c092ef0>
```

También puede crear un atributo sobre la marcha.

```
In [64]: naranja.forma = 'Redonda'
naranja.forma
```

```
Out[64]: 'Redonda'
```

Acceso a miembros de una clase en Python

```
In [65]: naranja.color
```

```
Out[65]: ''
```

Esto devuelve una cadena vacía porque eso es lo que especificamos en la definición de clase.

```
In [66]: naranja.decir_hola()
```

Hola

Aquí, llamamos al método `decir_hola()` en `naranja`. Un método puede tomar argumentos, si se define de esa manera.

```
In [67]: class fruta:
    '''Esta clase crea instancias de frutas'''
    color = ''
    def decir_hola(self):
        print('Hola')

    def tamaño(self, x):
        print(f'Mi tamaño es {x}')
```

```
In [68]: naranja = fruta()
naranja.tamaño(7)
```

Mi tamaño es 7

Una clase de Python también puede tener algunos atributos especiales, como `__doc__` para la cadena de documentación.

```
In [69]: fruta.__doc__
```

```
Out[69]: 'Esta clase crea instancias de frutas'
```

Constructor se define con el metodo `__init__()`

```
In [71]: class fruta:
        '''Esta clase crea instancias de frutas'''
        def __init__(self, color, tamaño):
            self.color = color
            self.tamaño = tamaño

        def saludo(self):
            print(f'Mi color es {self.color} y mi tamaño es {self.tamaño}')
```

```
In [72]: naranja = fruta('anaranjado', 7)
naranja.saludo()
```

Mi color es anaranjado y mi tamaño es 7

Atributos de la clase

```
In [73]: class fruta:
        '''Esta clase crea instancias de frutas'''
        forma = 'Redonda'
        def __init__(self, color, tamaño):
            self.color = color
            self.tamaño = tamaño

        def saludo(self):
            print(f'Mi color es {self.color} y mi tamaño es {self.tamaño}')
```

```
In [74]: fruta.forma
```

```
Out[74]: 'Redonda'
```

Otro ejemplo

```
In [75]: class Saludo:

    # Constructor
    def __init__(self, nombre):
        self.nombre = nombre # Crear una variable de instancia

    # Método de instancia
    def saludar(self, gritar=False):
        if gritar:
            print ('HOLA, %s!' % self.nombre.upper())
        else:
            print ('Hola, %s' % self.nombre)

s = Saludo('Fred') # Construct an instance of the Greeter class
s.saludar()        # Call an instance method; prints "Hello, Fred"
s.saludar(gritar=True) # Call an instance method; prints "HELLO, FRED!"
```

```
Hola, Fred
HOLA, FRED!
```

Ejercicio

Desarrolle un programa en Python 3 que implemente el juego de adivinar un número entre 1 y 100:

- Pida al jugador que ingrese su nombre. Utilice su nombre para imprimir un saludo.
- Genere un número aleatorio de 1 a 100 y guárdelo como un número objetivo para que el jugador lo adivine.
- Lleve un registro de cuántas suposiciones ha hecho el jugador. Antes de cada suposición, hágales saber cuántas suposiciones (de 10) que han dejado.
- Pida al jugador que adivine cuál es el número objetivo.
- Si la suposición del jugador es menor que el número objetivo, diga "Oops. Su conjetura fue baja". Si la suposición del jugador es mayor que el número objetivo, diga "Oops. Su conjetura fue alta".
- Si la suposición del jugador es igual al número objetivo, dígales "Buen trabajo, [nombre]! ¿Adivinaste mi número en [número de conjeturas] conjeturas!"
- Si el jugador se queda sin turnos sin adivinar correctamente, diga "Lo siento, no obtuvo mi número, mi número fue [objetivo]".
- Sigue permitiendo que el jugador adivine hasta que lo logren, o se quedan sin turnos.

Referencias

- Tutorial de Python oficial actualizado y traducido al español <http://docs.python.org.ar/tutorial/>
- Curso de introducción a Python para científicos e ingenieros de la Universidad de Alicante en Youtube https://www.youtube.com/playlist?list=PLoGFizEtm_6iheDXw2-8onKClyxgstBO1
- Introducción a la programación con Python, Universitat Jaume I <http://repositori.uji.es/xmlui/bitstream/10234/102653/1/s93.pdf>