

# Prácticas de Aprendizaje Automático

## Clase 1: Introducción a Python

Pablo Mesejo y Francisco Baldán

Universidad de Granada

Departamento de Ciencias de la Computación e Inteligencia Artificial



UNIVERSIDAD  
DE GRANADA



# Profesor

- Francisco Javier Baldán Lozano
- Email: [fjbaldan@go.ugr.es](mailto:fjbaldan@go.ugr.es)
- Tutorías:
  - Martes: 13:00-15:00
  - Se puede concertar cita en otro horario (avisadme por correo, al menos, 1 día antes).

# Sobre este curso (1)

- Esta clase no es un curso sobre Python
  - No tenéis que convertirlos en expertos en Python ni dominar todas las funcionalidades
  - No se evaluará la elegancia del código
    - Al margen de que el código entregado tiene que poder ejecutarse y resolver el problema indicado
- Python se considera una mera herramienta para resolver problemas de Aprendizaje Automático

# Sobre este curso (y 2)

- Asistencia a prácticas no obligatoria
- Podéis ir al grupo de prácticas que queráis, pero...
  - ... al estar apuntados a este grupo, seré yo quien evalúe vuestros trabajos

# Temporización Prácticas

## Calendario Aproximado

- Sesión 1: Semana 22 de Febrero
    - Seminario1. Introducción a Python
  - Sesión 2: Semana 01 de Marzo
    - Seminario1. Introducción a Python
  - Sesión 3: Semana 08 de Marzo
    - Presentación Práctica 1 (**Búsqueda Iterativa de Óptimos y Regresión Lineal**)
  - Sesión 4: Semana 15 de Marzo
    - Práctica 1 - Seguimiento/Dudas
  - Sesión 5: Semana 22 de Marzo
    - Práctica 1 - Seguimiento/Dudas
  - SEMANA SANTA
  - Sesión 6: Semana 05 de Abril
    - Presentación Práctica 2 (**Complejidad de H y Modelos Lineales**)
  - Sesión 7: Semana 12 de Abril
    - Práctica 2 - Seguimiento/Dudas
- Entrega P0 (Python / 5 ptos):**  
14 de Marzo
- Entrega P1 (12.5 ptos + 2):**  
4 de Abril

# Temporización Prácticas

## Calendario Aproximado

- Sesión 8: Semana 19 de Abril
    - Práctica 2 - Seguimiento/Dudas
  - Sesión 9: Semana 26 de Abril
    - Presentación Práctica 3 (**Ajuste de Modelos Lineales**)
  - Sesión 10: Semana 03 de Mayo
    - Práctica 3 - Seguimiento/Dudas
  - Sesión 11: Semana 10 de Mayo
    - Práctica 3 - Seguimiento/Dudas
  - Sesión 12: Semana 17 de Mayo
    - Práctica 3 - Seguimiento/Dudas
  - Sesión 13: Semana 24 de Mayo
    - Proyecto Final - Seguimiento/Dudas
  - Sesión 14: Semana 31 de Mayo (**sin clase Viernes**)
    - Proyecto Final - Seguimiento/Dudas
  - Sesión 15: Semana 07 de Junio (**sin clase Viernes**)
    - Proyecto Final - Seguimiento/Dudas
- Entrega P2 (12 ptos + 1.5):**  
25 de Abril
- Entrega P3 (20 ptos):**  
23 de Mayo
- Entrega Proyecto Final (25 ptos + 3):**  
14 de Junio

# Índice

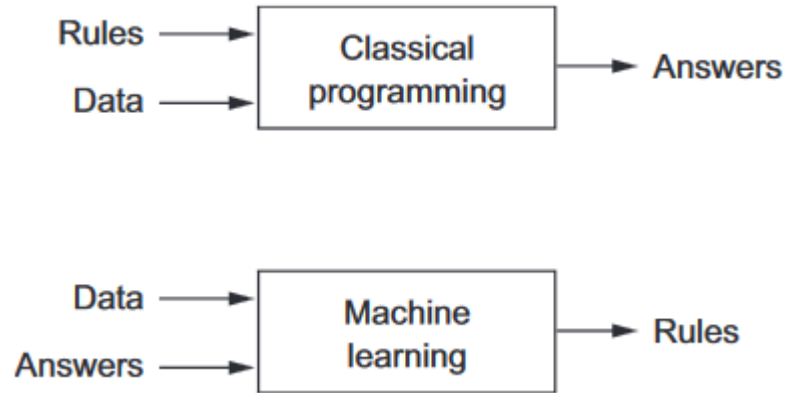
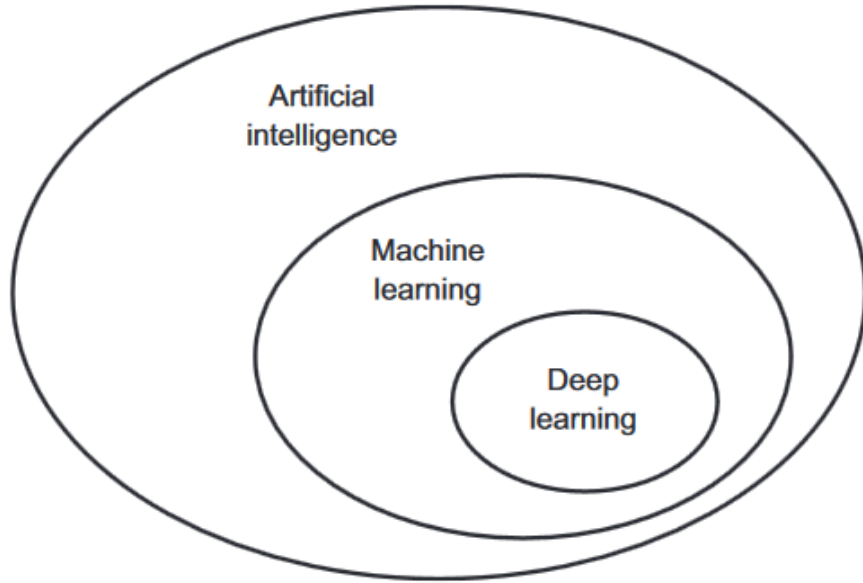
1. ¿Qué es el Aprendizaje Automático?
2. ¿Por qué usamos Anaconda, Python y Scikit-learn para las prácticas?
3. Instalación y uso básico de Anaconda
4. Editor Spyder
5. Primeros pasos con Python
6. Listas, tuplas y diccionarios
7. Indexado
8. Estructuras condicionales y bucles
9. Funciones
10. Clases

# ¿Qué es el Aprendizaje Automático?

- Que las máquinas aprendan a partir de los propios datos:  
datos + algoritmos de aprendizaje = aprendizaje automático
  - “Usar un conjunto de observaciones para descubrir un proceso subyacente”  
(Learning From Data, Abu-Mostafa et al., 2012)
    - Existe un patrón
    - No podemos describirlo matemáticamente
    - Tenemos datos
- Ejemplo:** recomendador de películas
- Nuestros gustos no son arbitrarios, sino que siguen ciertos patrones.
  - No podemos definir matemáticamente porqué nos gusta lo que nos gusta.
  - Tenemos datos de otras películas que nos han gustado.



# ¿Qué es el Aprendizaje Automático? (y 2)



Figuras extraídas de “Deep Learning with Python” (F. Chollet, 2018)

# ¿Por qué estudiar Aprendizaje Automático?

- Rama de la Inteligencia Artificial más pujante actualmente
- Muchísimas aplicaciones
  - Visión por computador y procesamiento de imágenes
    - Clasificación de imágenes
  - Procesado de señales
    - Reconocimiento del habla
  - Traducción automática
  - Conducción autónoma
  - Mejorar a humanos en actividades complejas (Go)
  - ...



# ¿Por qué estudiar Aprendizaje Automático? (y 2)

Por todo ello,

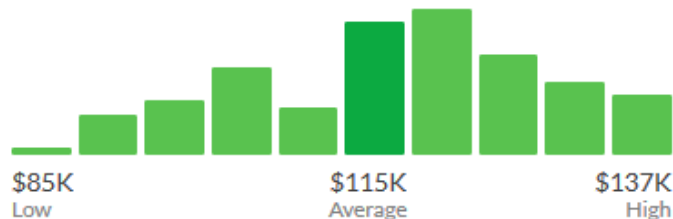
mucho trabajo (y bien pagado):

## Machine Learning Engineer Salaries in United States

103 Salaries Updated Feb 4, 2019

Average Base Pay

**\$114,826** /yr



**glassdoor**

# ¿Por qué Python? (1)

- + **Lenguaje de propósito general de alto nivel** creado a finales de los '80 por Guido Van Rossum

- + Permite el **desarrollo rápido de aplicaciones**

- + Cálculo científico de modo **similar a Matlab/Octave**

<https://numpy.org/doc/stable/user/numpy-for-matlab-users.html>

- + **Software libre** (Python Software Foundation License) **y gratuito**

- + Lenguaje interpretado que soporta programación orientada a objetos y que cuenta con una sintaxis simple e intuitiva

→ **Fácil de comenzar a trabajar con Python**



# ¿Por qué Python? (y 2)

- + **Ejecución rápida** (llamada a código compilado C)
- + Utilizado por los principales frameworks de **deep learning** (PyTorch, Theano, Keras, TensorFlow)
- + Debido a todo lo anterior
  - **ampliamente usado**
  - **mucha ayuda disponible** (foros, documentación)

Top Paying and Most Popular Programming Languages in 2020

Rank by Average Salary		Rank by Volume of Job Openings	
1. Python	\$119,000	1. Python	50,000
2. JavaScript	\$117,000	2. SQL	50,000
3. Java	\$104,000	3. Java	45,000
4. C	\$103,000	4. JavaScript	38,000
5. C++	\$102,000	5. C++	29,000
6. C#	\$97,000	6. C#	21,000
7. PHP	\$94,000	7. PHP	13,000
8. SQL	\$92,000	8. C	9,000

<https://www.codeplatoon.org/best-paying-most-in-demand-programming-languages-2020/>

Choose a Ranking

IEEE Spectrum

Trending Jobs

Open Custom

Create custom ranking

Language Types

Web

Enterprise

Mobile

Embedded

(Click to hide)

Language Ranking: IEEE Spectrum

Rank	Language	Type	Score
1	Python	Web	100.0
2	Java	Web	95.3
3	C	Mobile	94.6
4	C++	Mobile	87.0
5	JavaScript	Web	79.5
6	R	Web	78.6
7	Arduino	Embedded	73.2
8	Go	Web	73.1
9	Swift	Mobile	70.5
10	Matlab	Web	68.4

# ¿Por qué Scikit-learn?

- + **Paquete/librería de Python para aprendizaje automático**

Gran cantidad de algoritmos y funciones para el tratamiento y análisis de datos

Originalmente desarrollada por David Cournapeau en 2007

- + Construido sobre las librerías NumPy, SciPy y matplotlib (que veremos en la próxima clase)

- + Software libre (licencia BSD)

- + Muchas librerías compatibles (entre ellas para Deep Learning).



# Anaconda

- Distribución libre y abierta de Python
- Sistema de administración de paquetes **Conda**
- **Spyder** como entorno integrado de desarrollo (IDE)

**Anaconda Distribution**

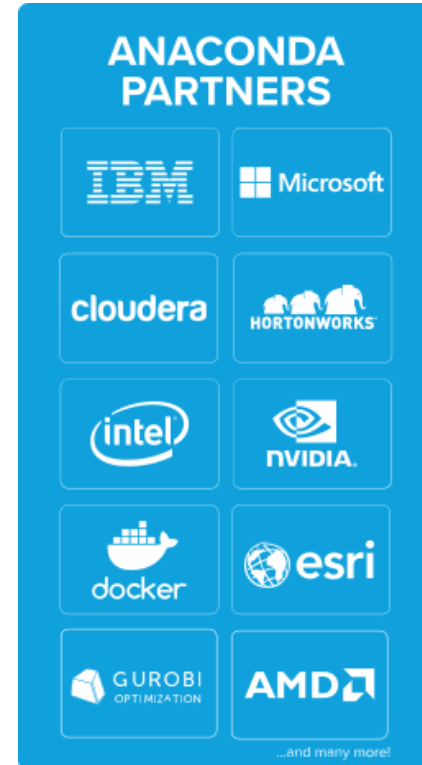
The World's Most Popular Python/R Data Science Platform



**ANACONDA®**

# ¿Por qué Anaconda?

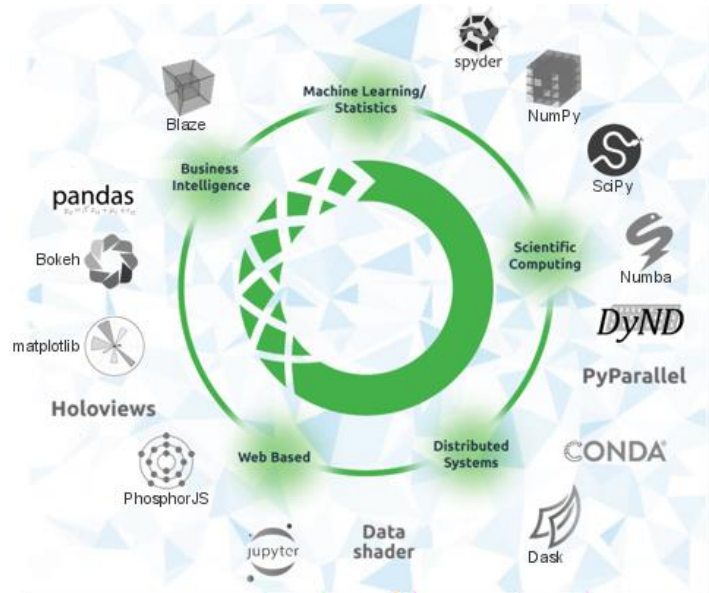
- Incluye más de 1000 librerías software para uso general y cálculo científico (*150 ya incluidas en la instalación inicial*)
- Más de 4,5 millones de usuarios y más de 150 empresas cliente
- Permite crear entornos:
  - Podríamos tener Anaconda 3.x y crear un entorno con Anaconda 2.7
  - También entornos con combinaciones de librerías concretas





# En resumen, principalmente, haremos uso de...

- **Spyder**: entorno de desarrollo integrado (Integrated Development Environment, IDE) que incluye intérprete (IPython), editor, depurador (debugger), asistente de ayuda, etc.
- **Conda**: gestor de librerías que resuelve problemas de dependencias y de control de sus versiones.
- **Scikit-learn**: librería para desarrollo de algoritmos de aprendizaje automático.
- **NumPy**: librería que permite trabajar de manera eficiente con vectores y matrices.
- **Matplotlib**: librería para la creación de gráficos 2D y 3D.



# Instalación Anaconda

## Requisitos:

- License: Free use and redistribution under the terms of the End User License Agreement.
- OS: Windows 7+, 64-bit macOS 10.13+, or Linux, including Ubuntu, RedHat, CentOS 6+, & others
- Minimum 5 GB disk space to download and install.

## Instalación Anaconda & IDE Spyder:

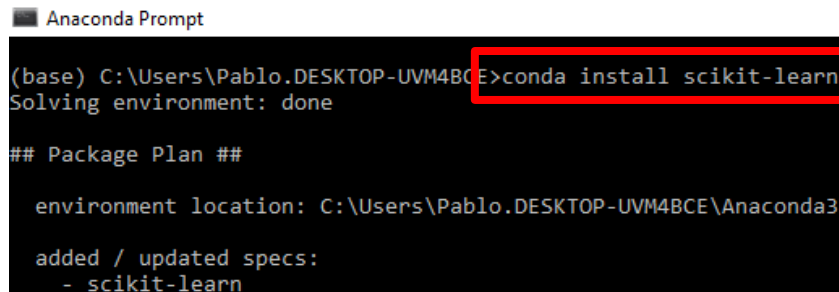
- Descargar: <https://www.anaconda.com/download/> (~279-529MB)  
→ **Python 3.8**
  - Windows → Doble click
  - Linux → `$ bash Anaconda3-2020.11-Linux-x86_64.sh`

# Instalación paquetes necesarios

`conda install scikit-learn`

## Windows

- Desde **Anaconda Prompt**
- Si queremos usar la consola (CMD), tenemos que añadirlo al path



```
Anaconda Prompt
(base) C:\Users\Pablo.DESKTOP-UVM4BCE>conda install scikit-learn
Solving environment: done

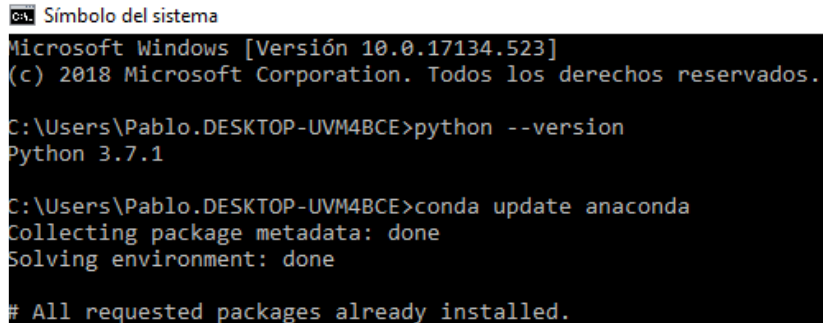
## Package Plan ##

  environment location: C:\Users\Pablo.DESKTOP-UVM4BCE\Anaconda3

added / updated specs:
- scikit-learn
```

## Linux

- `source ~/.bashrc`
- Si el terminal no reconoce **conda**  
→ modificar `.bashrc`  
`export PATH=~/.anaconda3/bin:$PATH`



```
Símbolo del sistema
Microsoft Windows [Versión 10.0.17134.523]
(c) 2018 Microsoft Corporation. Todos los derechos reservados.

C:\Users\Pablo.DESKTOP-UVM4BCE>python --version
Python 3.7.1

C:\Users\Pablo.DESKTOP-UVM4BCE>conda update anaconda
Collecting package metadata: done
Solving environment: done

# All requested packages already installed.
```

# Uso básico anaconda (1)

**Anaconda** usa **conda** para instalar paquetes.

- Ayuda: **conda -h**
- Instalar paquete: **conda install <name>**
- Actualizar: **conda update <name>**, ej. **conda update spyder**
- Desinstalar paquete: **conda remove <name>**

# Uso básico anaconda (y 2)

- Crear entorno: `conda create -n <name>`
- Eliminar entorno: `conda env remove -n <name>`
- Ayuda sobre comandos en entorno: `conda env <command> -h`
- Listar entornos: `conda-env list` o `conda env list` o `conda info -envs`
- Listar paquetes instalados en entorno: `conda list -n <env_name> <pkg_name>`
- Activar/Desactivar entorno:
  - Windows: `activate <name>` / `deactivate`
  - Linux: `source activate <name>` / `source deactivate <name>`

<https://conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html>

```
C:\Users\Pablo.DESKTOP-UVM4BCE>conda create -n first_tests
Collecting package metadata: done
Solving environment: done

## Package Plan ##

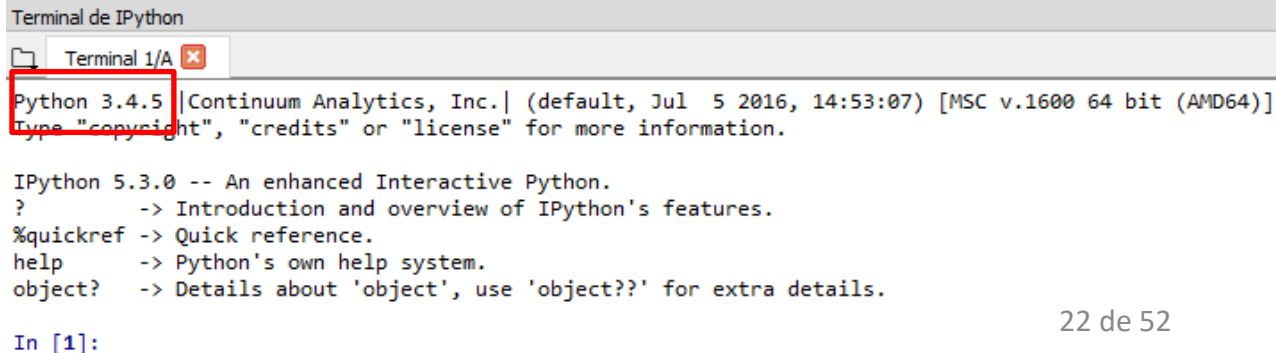
  environment location: C:\Users\Pablo.DESKTOP-UVM4BCE\Anaconda3\envs\first_tests

Proceed ([y]/n)? y
```

# Entornos Virtuales

- Entorno virtual o '*virtual environment*': carpeta en la que se encuentran los ejecutables de python y las distintas versiones de las librerías que vayamos a usar.
  - Gracias a los entornos virtuales, podremos estar desarrollando varias aplicaciones con distintas versiones de las librerías, incluso del propio intérprete de python.
- Ejemplo de utilidad de los entornos:
  - Necesidad de desarrollar una aplicación con una versión distinta de Python o unos requisitos específicos en las versiones de los paquetes empleados

```
conda create -n myenv python=3.4
activate myenv
# hay que instalar spyder
# en el nuevo entorno
conda install spyder
spyder
```



```
Terminal de IPython
Python 3.4.5 |Continuum Analytics, Inc.| (default, Jul 5 2016, 14:53:07) [MSC v.1600 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 5.3.0 -- 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]:
```

# Proyectos

- Directorio que contiene un fichero de configuración `anaconda-project.yml` junto con scripts, notebooks y otros archivos.
  - Se usan para encapsular proyectos de ciencia de datos y hacerlos fácilmente portables.
  - Un Anaconda Project automatiza la configuración, de modo que se puedan compartir proyectos que se pueden ejecutar con un único comando  
`anaconda-project run`

<https://anaconda-project.readthedocs.io/en/latest/user-guide/concepts.html>

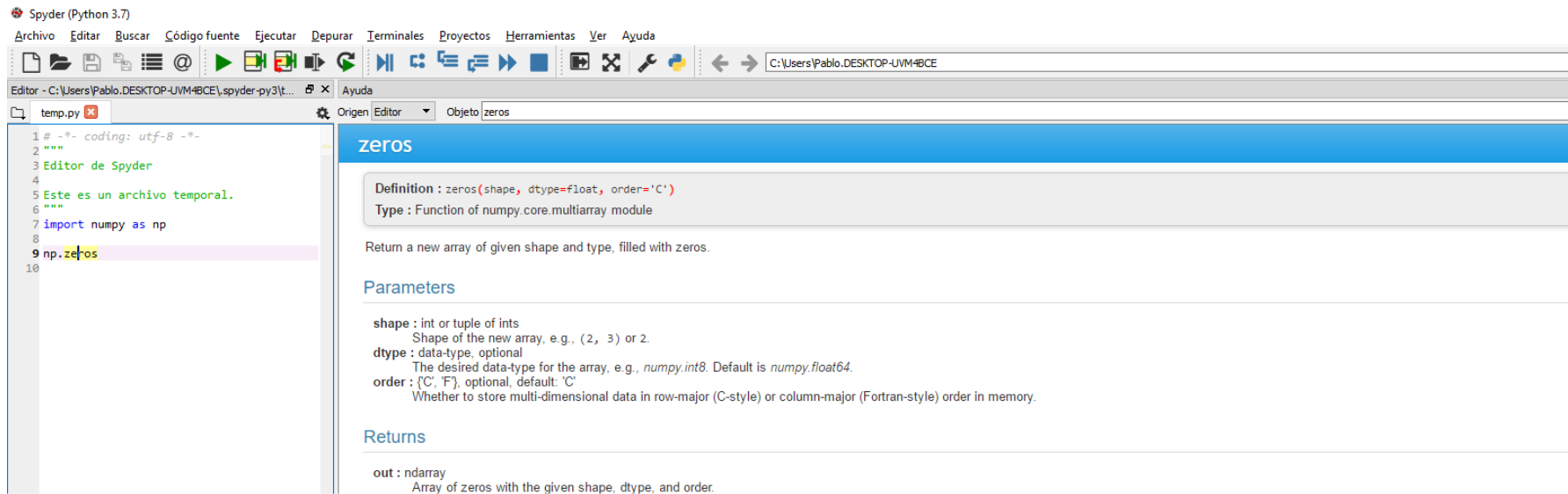
<https://anaconda-project.readthedocs.io/en/latest/user-guide/getting-started.html>

- Ejemplo:

```
$ cd /home/pablo/mystuff
$ anaconda-project init --directory iris
Create directory '/home/pablo/mystuff/iris'? y
Project configuration is in /home/pablo/mystuff/iris/anaconda-project.yml
```

# Ayuda Spyder

- **Ctrl + i** con el cursor sobre la función o método a consultar.



- Consultar código fuente: Ctrl + click izquierdo (**Ctrl + g** o click derecho – Ir a la definición)



# Primeros pasos (1)

- Iniciar Spyder.
- En la línea de comandos (terminal) escribir código que se evalúa sobre la marcha

```
In [1]: 2*5+10**2  
Out[1]: 110
```

- Se puede realizar asignaciones, importar paquetes, etc

- **Indentación** obligatoria

Podéis usarlo para hacer pruebas rápidas.

Importar un módulo usando un alias

```
In [2]: import numpy as np
```

```
In [3]: z = np.zeros((5,2), np.float32)
```

```
In [4]: z.shape  
Out[4]: (5, 2)
```

```
In [5]: z.sum()  
Out[5]: 0.0
```

```
In [6]: z  
Out[6]:  
array([[0., 0.],  
       [0., 0.],  
       [0., 0.],  
       [0., 0.],  
       [0., 0.]], dtype=float32)
```

# Primeros pasos (2)

```
# -*- coding: utf-8 -*-
```

```
#Variables
```

```
entero1 = 5 #Int
entero2 = 505 #Int
flotante = 50.5 #Float
boolean_t = True #Boolean
boolean_f = False #Boolean
string1 = 'String1' #String
string2 = 'String2' #String
```

```
#Operaciones aritméticas
```

```
suma = entero1 + flotante
resta = entero1 - flotante
producto = entero1 * flotante
print(producto)
division = entero1 / flotante
print(division)
division_entera = entero2 // entero1
print(division_entera)
resto = entero2 % entero1
print(resto)
```

Necesario, generalmente, para introducir strings y comentarios con acentos y ñ.

Sin el `print()` no ves el resultado de la operación

```
In [2]: suma = entero1 + flotante
...: resta = entero1 - flotante
...: producto = entero1 * flotante
...: print(producto)
252.5
```

```
In [3]: division = entero1 / flotante
...: print(division)
0.09900990099009901
```

```
In [4]: division_entera = entero2 // entero1
...: print(division_entera)
101
```

```
In [5]: resto = entero2 % entero1
...: print(resto)
0
```

# Primeros pasos (3)

```
#Operaciones lógicas
igual = entero2==(500 + 5)
no_igual = entero1 != suma
mayor = entero2 > entero1 #>=
menor = entero1 < entero2 # <=
and_logico = igual and mayor
or_logico = igual or no_igual

#Cambiar tipos
entero2flotante = float(entero1)
flotante2entero = int(flotante)
astring = str(entero2)
abool = bool(entero1)

#Strings
formatear = 'String con entero %d, flotante %f y string %s' % (entero1,
                                                             flotante,
                                                             string1)

concatenar = string1 + str(entero1)

#Mostrar por pantalla
print('Dos de los strings:', string1, string2)
print('String y entero:', concatenar, entero1)
```

## Ayuda sobre un comando concreto

Terminal de IPython

Terminal 5/A

```
In [6]: help('!=')
Operator precedence
*****
```

The following table summarizes the operator precedence in Python, from lowest precedence (least binding) to highest precedence (most binding). Operators in the same box have the same precedence. Unless the syntax is explicitly given, operators are binary. Operators in the same box group left to right (except for exponentiation, which groups from right to left).

Note that comparisons, membership tests, and identity tests, all have the same precedence and have a left-to-right chaining feature as described in the Comparisons section.

Operator	Description
"lambda"	Lambda expression
"if" - "else"	Conditional expression
"or"	Boolean OR
"and"	Boolean AND
"not" "x"	Boolean NOT

# Primeros pasos (y 4)

`None`

# Representa la ausencia de algo

`x = None`

# Las variables pueden ser None

`array = [1,2, None]`

# Las listas pueden contener None

`def func():`

`return None`

# Las funciones pueden devolver None

- No se pueden usar como nombres de variables las siguientes palabras reservadas de Matlab:

```
from keyword import iskeyword, kwlist
```

```
kwlist
```

Out: ['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']

```
iskeyword('try')
```

Out: True

# (Algunos) Tipos de Datos

Tipo	Clase	Notas	Ejemplo
int	Entero	Números enteros con un rango ilimitado	53
float	Real	Coma flotante de doble precisión	3.141592653589793
complex	Complejo	Parte real e imaginaria	(3 + 5j)
bool	Booleano	Valores verdadero o falso	True o False
str	Cadena	Inmutable, contiene texto	"Hola"
list	Secuencia	Mutable, contiene objetos de diverso tipo	[4, "Hola", 3.14]
tuple	Secuencia	Inmutable, contiene objetos de diverso tipo	(4, "Hola", 3.14)
set	Conjunto	Mutable, sin orden, sin duplicados	set([4, "Hola", 3.14])
frozenset	Conjunto	Inmutable, sin orden, sin duplicados	frozenset([4, "Hola", 3.14])
dict	Diccionario	Pares de clave:valor	{"clave1":4, "clave2":"Hola"}

# Más sobre tipos de datos (1)

- Todos los tipos de datos son objetos en python: id, type, valor
  - **type**(dato), devuelve el tipo
  - **id**(dato), devuelve su identificador (un entero único)
- Se pueden forzar los tipos para obtener el resultado deseado:
  - **bool**(dato), **int**(dato), **float**(dato), **str**(dato),...

Python es un lenguaje:

- **Fuertemente tipado:**  $1 + '1' \rightarrow \text{Error!}$
- **Dinámicamente tipado:**  $\text{foo} = [1,2,3] \dots \rightarrow \dots \text{foo} = \text{'hello!'}$

<https://wiki.python.org/moin/Why%20is%20Python%20a%20dynamic%20language%20and%20also%20a%20strongly%20typed%20language>

# Más sobre tipos de datos (y 2)

- Tipado fuerte:
  - Los objetos/valores obedecen reglas estrictas sobre cómo pueden interactuar.
  - Ejemplo de tipado débil (PHP):  
`$x = 1 + "1"; // x es 2`
- Tipado dinámico:
  - Son los datos quienes tienen tipo, y no las variables. Es decir, el tipo de dato que un objeto puede almacenar es mutable.
  - Ejemplo de tipado estático (C#):  
`int i = 5;  
i = "5"; // Error! i solo puede contener enteros!`

# Operaciones y Precedencia

Operación	Símbolo	Ejemplo	Resultado	Precedencia
Suma	+	$6 + 2$	8	4
Resta	-	$6 - 2$	4	4
Multiplicación	*	$6 * 2$	12	2
División Real	/	$7 / 2$	3.5	2
División Entera	//	$6.5 // 2$	3.0	2
Módulo	%	$5 \% 2$	1	3
Potenciación	**	$6 ** 2$	36	1



# Más sobre operaciones

Al realizar ciertas operaciones, los resultados pueden presentar errores de redondeo:

```
>>> 100 / 3  
33.333333333333336
```

Debido a estos errores, dos operaciones que debieran dar el mismo resultado pueden dar resultados diferentes:

```
>>> 4 * 3 / 5  
2.4  
>>> 4 / 5 * 3  
2.4000000000000004
```

Y viceversa:

```
>>> round(3.45, 1)  
3.5  
>>> round(3.55, 1)  
3.5
```

Este error se debe a que **Python almacena los números decimales en binario** (con 53 bits de precisión, norma IEEE-754), **y pasar de decimal a binario provoca errores de redondeo.**

Cuando se pide un cálculo con números decimales, Python convierte esos números decimales a binario, realiza la operación en binario y convierte el resultado de nuevo a decimal para mostrárselo al usuario.

En la mayor parte de los casos, esto no es especialmente problemático. No obstante, si se necesita total exactitud, se pueden utilizar bibliotecas específicas: **decimal, fractions o mpmath**

# Listas, tuplas y diccionarios (1)

```
In [1]: tupla = (5, 't1', True, 0.5)
...: print(tupla)
(5, 't1', True, 0.5)
```

```
In [2]: lista = [5, 't1', True, 0.5]
...: print(lista)
[5, 't1', True, 0.5]
```

```
In [3]: l_tupla = len(tupla)
...: l_lista = len(lista)
...: print(l_tupla)
...: print(l_lista)
```

4  
4

```
In [4]: print(tupla[2])
...: print(lista[2])
```

True  
True

```
In [5]: lista[2] = 1000
...: print(lista)
[5, 't1', 1000, 0.5]
```

**Tuplas:** Almacenan datos de cualquier tipo y se acceden mediante índices enteros.

- **No pueden modificarse, solo consultarse (son estáticas).**

Nota para usuarios de MATLAB:  
Python indexa de 0 a N-1

```
In [6]: print(tupla[4])
Traceback (most recent call last):
```

```
File "<ipython-input-6-09eb472720a7>", line 1, in <module>
    print(tupla[4])
```

**IndexError:** tuple index out of range

# Listas, tuplas y diccionarios (2)

**Listas:** Almacenan datos de cualquier tipo y se acceden mediante índices enteros.

- Son dinámicas, es decir, pueden modificarse sus elementos, añadir, eliminar,...

```
#Añadir elemento
lista.append(False) #Al final
position = 1
lista.insert(position, 't21') #En una posición concreta
```

```
#Eliminar elemento
lista.remove('t1') #Buscando por el propio elemento
lista.pop() #Al final
lista.pop(1) #En la posición 1
```

```
#Concatenar
lista2 = ['a', 'b', 'c']
lista_combinada = lista + lista2 #Pone lista2 al final
                                # de lista
```

```
#Copiar
lista_copia = lista.copy()
```

```
In [3]: lista.append(False)
...: print(lista)
[5, 't1', 1000, 0.5, False]

In [4]: position = 1
...: lista.insert(position, 't21')
...: print(lista)
[5, 't21', 't1', 1000, 0.5, False]

In [5]: lista.remove('t1')
...: print(lista)
[5, 't21', 1000, 0.5, False]

In [6]: lista.pop()
...: print(lista)
[5, 't21', 1000, 0.5]

In [7]: lista.pop(1)
...: print(lista)
[5, 1000, 0.5]

In [8]: lista2 = ['a', 'b', 'c']
...: lista_combinada = lista + lista2
...: print(lista_combinada)
[5, 1000, 0.5, 'a', 'b', 'c']

In [9]: lista_copia = lista.copy()

In [10]: print(lista_copia)
[5, 1000, 0.5]
```

# Listas, tuplas y diccionarios (y 3)

**Diccionarios:** Almacenan datos de cualquier tipo y se acceden mediante palabras clave (keys).

- Pueden modificarse sus elementos, añadir, eliminar, ...

```
#Declarar  
diccionario = {'a': 1, 'b': 2.0}
```

```
#Añadir elemento  
diccionario['c'] = False
```

```
#Mostrar por pantalla  
print(diccionario)
```

```
#Eliminar elemento  
del diccionario['c']  
print(diccionario)
```

```
#Keys  
diccionario.keys()
```

```
#Values  
diccionario.values()
```

```
In [29]: diccionario = {'a': 1, 'b': 2.0}  
....:
```

```
.... #Añadir elemento  
.... diccionario['c'] = False  
....
```

```
.... #Mostrar por pantalla  
.... print(diccionario)
```

```
{'a': 1, 'b': 2.0, 'c': False}
```

```
In [30]: del diccionario['c']  
.... print(diccionario)
```

```
{'a': 1, 'b': 2.0}
```

```
In [31]: print(diccionario.keys())  
dict_keys(['a', 'b'])
```

```
In [32]: print(diccionario.values())  
dict_values([1, 2.0])
```

# Indexado (1)

- `lista[inicio:fin:pasos]`

- Toda la lista: `lista[:]`
- Desde inicio: `lista[inicio:]`
- Hasta fin: `lista[:fin]`
- Solo pares: `lista[::2]`

```
In [33]: lista = [5, 't1', True, 0.5]
```

```
In [34]: lista[:]
```

```
Out[34]: [5, 't1', True, 0.5]
```

```
In [35]: lista[0:2]
```

```
Out[35]: [5, 't1']
```

```
In [36]: lista[3:]
```

```
Out[36]: [0.5]
```

```
In [37]: lista[::2]
```

```
Out[37]: [5, True]
```

- Podemos usar índices negativos, estos comenzarán a contar desde el final de la lista.

```
In [38]: lista[-1]
```

```
Out[38]: 0.5
```

- La posición -1 es la del último elemento de la lista.



Posiciones

0

1

2

3

4

5

6

7

8



Posiciones

-9

-8

-7

-6

-5

-4

-3

-2

-1



# Indexado (y 2)

```
lista = [0, 1, 2, 3, 4]
print(lista[0:15:1])
[0, 1, 2, 3, 4]
print(lista[0:15:2])
[0, 2, 4]
print(lista[0:4:2])
[0, 2]
print(lista[0:-1:2])
[0, 2]
print(lista[::-1])
[4, 3, 2, 1, 0]
print(lista[3:0:-1])
[3, 2, 1]
print(lista[3::-1])
[3, 2, 1, 0]
print(lista[0:3:-1])
[]
```

# Condicionales y bucles

```
#Condicional
if condicion:
    #Hacer algo
elif otra_condicion:
    #Hacer algo
else:
    #Hacer algo

#Bucle for
for i in range(inicio, fin, paso):
    #Hacer algo
    print(i)

for elemento in lista:
    #Hacer algo

#Bucle while
while condicion:
    #Hacer algo
```

```
In [33]: for x, y in [(1,10), (2,20), (3,30)]:
...:     print (x, y)
1 10
2 20
3 30
```

La indentación es importante!!!

```
nota_obtenida = 6.8
if nota_obtenida < 5:
    print('Suspense')
elif 5 >= nota_obtenida < 7:
    print('Aprobado')
elif 7 >= nota_obtenida < 8.5:
    print('Notable')
else:
    print('Sobresaliente')
```

print('Suspense')

IndentationError: expected an indented block

# Funciones (1)

- Se pueden crear funciones nuevas o emplear aquellas disponibles en los módulos.
- Tres formas principales de importar una librería o módulo:

```
import module as md  
md.fun()
```

```
from module import *  
fun()
```

```
from module import fun as f1  
f1()
```

- Para conocer todas las funciones dentro del módulo importado disponemos de **dir**(module).



# Funciones (2)

```
def funcion(a, b=1):  
    c = a + b  
    return c
```

Valor por defecto para el parámetro

```
c = funcion(1,2) #O funcion(a=1, b=2)  
print(c)  
c_def = funcion(1)  
print(c_def)
```

Sin **return**  
el **print** posterior  
imprimiría **None**

**return** fuerza la salida de la función  
y devuelve un valor al **caller**

---

```
In [1]: a = [5,3,2,6,1]
```

```
In [2]: a  
Out[2]: [5, 3, 2, 6, 1]
```

```
In [3]: a.sort()
```

```
In [4]: a  
Out[4]: [1, 2, 3, 5, 6]
```

Al omitir el segundo parámetro (**b**),  
**b** tiene el valor por defecto (1)

Hay métodos que son **in-place**: no devuelven nada, modifican el objeto directamente, y no necesitan asignaciones

# Funciones (3)

Cuidado a la hora de realizar asignaciones!

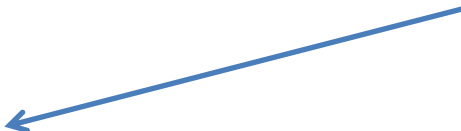
```
In [47]: some_guy = 'Fred'

In [48]: first_names = []
...: first_names.append(some_guy)

In [49]: print(first_names)
['Fred']

In [50]: another_list_of_names = first_names
...: another_list_of_names.append('George')
...: some_guy = 'Bill'

In [51]:
...: print (some_guy, first_names, another_list_of_names)
...:
Bill ['Fred', 'George'] ['Fred', 'George']
```



¿Por qué **first\_names**  
no contiene solamente ['Fred']?

Ambas variables  
(**another\_list\_of\_names** y  
**first\_names**) están ligadas al  
mismo objeto.

El string object (**some\_guy**) y el  
list object (**first\_names**) siguen  
siendo los únicos objetos creados  
por el intérprete de Python.

```
id(first_names)
1758256788808

id(another_list_of_names)
1758256788808
```

# Funciones (4)

```
In [54]: first_names = ['Fred', 'George', 'Bill']  
...: last_names = ['Smith', 'Jones', 'Williams']  
...: name_tuple = (first_names, last_names)  
  
In [55]: print(name_tuple)  
(['Fred', 'George', 'Bill'], ['Smith', 'Jones', 'Williams'])  
  
In [56]: first_names.append('Igor')  
  
In [57]: print(first_names)  
['Fred', 'George', 'Bill', 'Igor']  
  
In [58]: print(name_tuple)  
(['Fred', 'George', 'Bill', 'Igor'], ['Smith', 'Jones', 'Williams'])
```

Un objeto “inmutable” (como una tupla) puede “mutar” si contiene objetos “mutables” (como listas)

# Funciones (5)

```
In [60]: def foo(bar):  
...:     bar.append(42)  
...:     print(bar)
```

```
In [61]: answer_list = []  
...:     foo(answer_list)  
[42]
```

```
In [62]: print(answer_list)  
[42]
```

Si **bar** se refiere a un objeto mutable (como una lista), y **foo** cambia su valor, entonces estos cambios serán visibles fuera de la función.

---

```
In [63]: def foo(bar):  
...:     bar = 'new value'  
...:     print (bar)  
...:     # >> 'new value'  
...:  
...:  
...: answer_list = 'old value'  
...: foo(answer_list)  
new value
```

```
In [64]: print(answer_list)  
old value
```

Si **bar** se refiere a un objeto inmutable (como un string), lo máximo que puede hacer **foo** es crear una variable interna **bar** y ligarla a algún otro objeto.

# Funciones (y 6)

Un último ejemplo:

```
In [1]: def spam(eggs):  
...:     eggs.append(1)  
...:     eggs = [2, 3]  
...:     print(eggs)
```

```
In [2]: ham = [0]  
...: spam(ham)  
[2, 3]
```

```
In [3]: print(ham)  
[0, 1]
```

Cuando se llama a **spam**,  
**eggs** y **ham** apuntan al mismo valor ([0])

Cuando hacemos **eggs.append(1)**  
→ [0] se convierte en [0, 1]

Cuando hacemos **eggs = [2, 3]**  
→ ahora **eggs apunta a una nueva lista en memoria** que contiene [2, 3]  
Pero **ham** apunta todavía a la lista [0, 1]

# Formateo de salida

```
print('Valor real de 1/3', 1/3, sep=' : ')\nprint('Valor real de 1/3 : ' + str(1/3))\nprint('Valor real de 1/3 : {} '.format(1/3))
```

Valor real de 1/3 : 0.3333333333333333

```
print('Resultado 1 - {} | Resultado 2 - {}'.format(1/3, 1/5))
```

Resultado 1 – 0.3333333333333333 | Resultado2 – 0.2

```
print('Resultado 1 - {0:5.3f} | Resultado 2 - {1:5d}'.format(1/3, 25))
```



Resultado 1 – 0.333 | Resultado 2 – 25

```
print('Resultado 1 - {1:8d} | Resultado 2 - {0:7.4f}'.format(1/3, 25))
```

Resultado 1 – 25 | Resultado 2 – 0.3333

# Clases (1)

```
class Clase():
```

```
    def __init__(self, a):  
        self.a = a
```

```
    def llamar(self, b):  
        return self.a*b
```

`__init__()` no es un constructor (`__new__()`): es llamado inmediatamente después de que el objeto haya sido creado y se usa para inicializarlo

```
class Clase2(Clase):
```

```
    def __init__(self, a, b=2.0):  
        super().__init__(a)  
        self.b=b
```

```
    def llamar(self, c):  
        return self.a*self.b*c
```

```
    def __call__(self, c):  
        return self.llamar(c)
```

Clase2 hereda los métodos de Clase

Sobreescribe el método `llamar()`

```
In [96]: c = 3
```

```
In [97]: clase2 = Clase2(a=1)
```

```
In [98]: d = clase2.llamar(c)
```

```
In [99]: print(d)  
6.0
```

Cuidado con las asignaciones en los métodos que modifiquen atributos (como `__init__`).

No olvidéis el **self**. **self** no es una palabra reservada: podríais usar **this** o **myself**, por ejemplo, aunque no se recomienda por ser una convención fuertemente aceptada.

# Clases (y 2)

```
In [1]: class Point(object):
...:     def __init__(self, x = 0, y = 0):
...:         self.x = x
...:         self.y = y
...:
...:     def distance(self):
...:         """Find distance from origin"""
...:         return (self.x**2 + self.y**2) ** 0.5
```

```
In [2]: p1 = Point(6,8)
```

```
In [3]: p1.distance()
Out[3]: 10.0
```

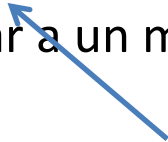
```
In [4]: Point.distance(p1)
Out[4]: 10.0
```

```
In [5]: type(Point.distance)
Out[5]: function
```

```
In [6]: type(p1.distance)
Out[6]: method
```

Si p1 es una instancia de P (i.e. un objeto):


p1.meth(arg) y P.meth(p1, arg) son formas equivalentes de llamar a un método

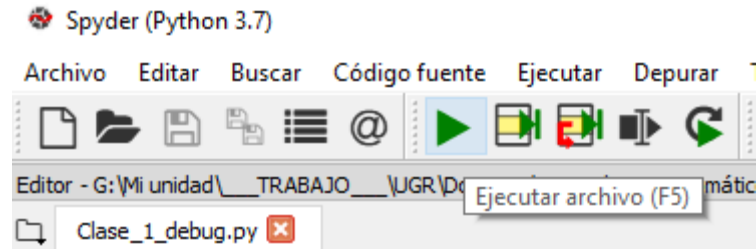


El **self** se refiere a p1



# Ejecutando scripts completos

- Si queréis ejecutar un fichero (p.ej. en Windows)
  - Id a Anaconda Prompt o CMD (si habéis metido Python en el path)
  - Ejecutad `python myfile.py`
- Seleccionad el bloque de código que queráis ejecutar en el editor de Spyder
  - F9 → se ejecutará en el terminal de Ipython
- Ejecutad archivo en Spyder con F5 o pulsando 



# Depurando código (1)

- Si os confunde la sintaxis, probad en el shell interactivo de IPython
- Comprobad valores (**print()**), tipos (**type()**), tamaños/formas (**shape()**), y atributos/métodos (**dir()**)

```
2 ** 5 / 2
16.0
```

```
2 ** (5 / 2)
5.656854249492381
```

```
In [1]: import numpy as np
....:
....: def funcion1(x):
....:     x.append(1)
....:     print(np.shape(x))
....:     print(len(x))
....:     print(type(x))
....:     print(dir(x))
....:     x = (2,3)
....:     print(np.shape(x))
....:     print(len(x))
....:     print(type(x))
....:     print(dir(x))
....:
....: y = [0, 1, 2, 3, 4, 5]
....: funcion1(y)
```

```
(7,)
```

```
7
```

```
<class 'list'>
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__',
 '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__',
 '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__',
 '__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__',
 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
(2,)
```

```
2
```

```
<class 'tuple'>
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__',
 '__getattribute__', '__getitem__', '__getnewargs__', '__gt__', '__hash__',
 '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__',
 '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'count', 'index']
```

# Depurando código (2)

Operaciones que pueden ser de utilidad:

- Verificar si dos arrays son *aproximadamente* iguales

`np.allclose(x, y)` #Se puede especificar la tolerancia

- Verificar si un array es próximo a cero (p.ej. el gradiente)

`np.allclose(x, 0)`

- Seleccionar todos los elementos menores o iguales a 0 en un array

`x[x <= 0]`

```
In [1]: import numpy as np
...: x = np.zeros((1,25))
...: z = np.repeat([1.e-05], 25)
...: np.allclose(x,z,atol=1.e-05)
```

Out[1]: True

```
In [2]: np.allclose(x,z,atol=1.e-06)
```

Out[2]: False

```
In [3]: z = np.repeat([1.e-05, 0, 3], 10)
...: z
```

Out[3]:

```
array([1.e-05, 1.e-05, 1.e-05, 1.e-05, 1.e-05, 1.e-05, 1.e-05, 1.e-05,
       1.e-05, 1.e-05, 0.e+00, 0.e+00, 0.e+00, 0.e+00, 0.e+00, 0.e+00,
       0.e+00, 0.e+00, 0.e+00, 0.e+00, 3.e+00, 3.e+00, 3.e+00, 3.e+00,
       3.e+00, 3.e+00, 3.e+00, 3.e+00, 3.e+00, 3.e+00])
```

```
In [4]: z[z<=0] #Get values
```

Out[4]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])

```
In [5]: z<=0 #Get boolean values about the condition
```



Out[5]:

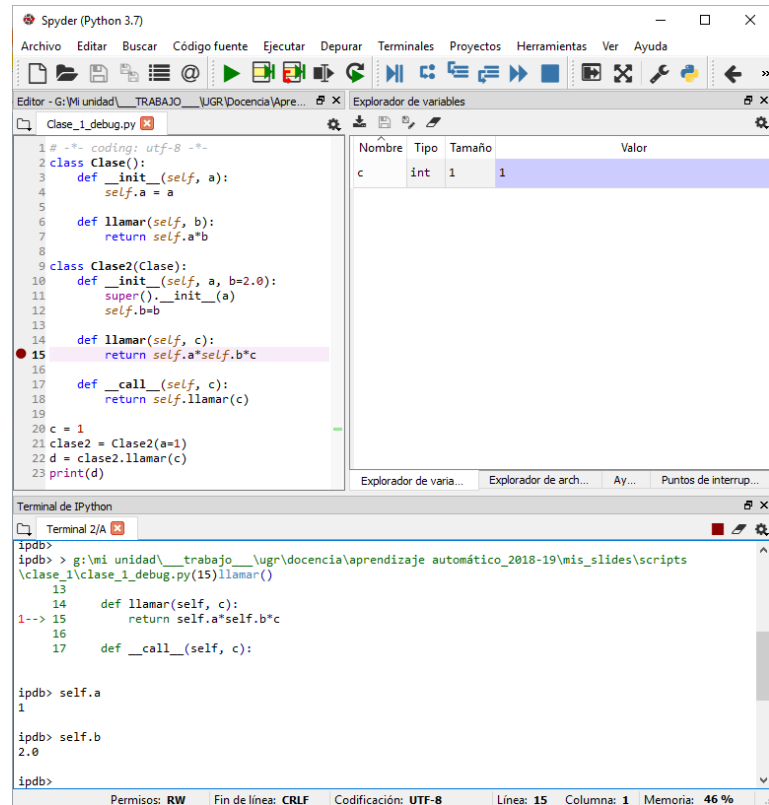
```
array([False, False, False, False, False, False, False, False, False,
       False, True, True, True, True, True, True, True, True, True,
       True, True, False, False, False, False, False, False, False,
       False, False, False])
```

```
In [6]: [i for i in range(len(z)) if z[i] <= 0] #Get indexes
```

Out[6]: [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

# Depurando código (y 3)

- Sobre la línea donde queráis poner un *breakpoint*
  - F12 (tanto para poner el *breakpoint* como para quitarlo)
  - Acordaos de pulsar el botón de Debug , y no el de ejecutar el código 



The screenshot shows the Spyder Python IDE interface. The main editor displays a Python script named 'Clase\_1\_debug.py'. A breakpoint is set at line 15, which is highlighted in red. The variable explorer on the right shows a single variable 'c' of type 'int' with a value of 1. The terminal at the bottom shows the execution of the script, with the current line of execution being line 15.

```
1 # -*- coding: utf-8 -*-
2 class Clase():
3     def __init__(self, a):
4         self.a = a
5
6     def llamar(self, b):
7         return self.a*b
8
9 class Clase2(Clase):
10     def __init__(self, a, b=2.0):
11         super().__init__(a)
12         self.b=b
13
14     def llamar(self, c):
15         return self.a*self.b*c
16
17     def __call__(self, c):
18         return self.llamar(c)
19
20 c = 1
21 clase2 = Clase2(a=1)
22 d = clase2.llamar(c)
23 print(d)
```

Variable Explorer:

Nombre	Tipo	Tamaño	Valor
c	int	1	1

Terminal de IPython:

```
ipdb> > g:\mi_unidad\trabajo\ugr\docencia\aprendizaje automático_2018-19\mis_slides\scripts\
\clase_1\clase_1_debug.py(15)llamar()
13
14     def llamar(self, c):
1--> 15         return self.a*self.b*c
16
17     def __call__(self, c):

ipdb> self.a
1

ipdb> self.b
2.0

ipdb>
```

# Prácticas de Aprendizaje Automático

## Clase 1: Introducción a Python

Pablo Mesejo y Francisco Baldán

Universidad de Granada

Departamento de Ciencias de la Computación e Inteligencia Artificial



UNIVERSIDAD  
DE GRANADA

