

# Introducción a Python

## Bloque III

José Manuel García Nieto – Universidad de Málaga ([jnieto@uma.es](mailto:jnieto@uma.es))

## Índice de Contenidos

- ¿Qué es Python?
- Indentación
- Tipos de datos
- Operadores
- Control de flujo
- Programación orientada a objetos
- Funciones predefinidas
- Estilo de código: PEP8
- Dependencias
- Pruebas unitarias



# Introducción a Python

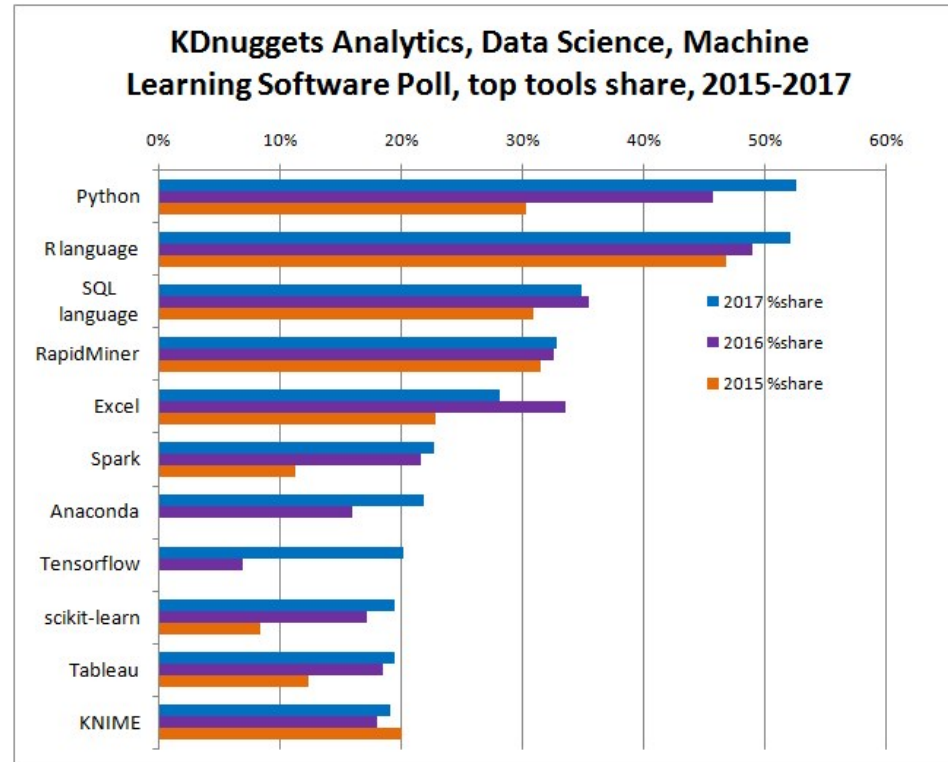
- Motivación: Programemos en Python

- Es un lenguaje de programación de alta abstracción
- Interpretado
- Dinámico y de propósito general
- Ideal para el “Prototipado” y el desarrollo rápido de aplicaciones (Curva de aprendizaje rápida)
- Gran comunidad de desarrolladores software libre (más de 91.000 repositorios en github)



# Introducción a Python

- Motivación: Programemos en Python
  - Ofrece un gran número de bibliotecas de funciones (y creciendo)
    - Sobre todo bibliotecas orientadas a cálculo, ciencias de datos y visualización

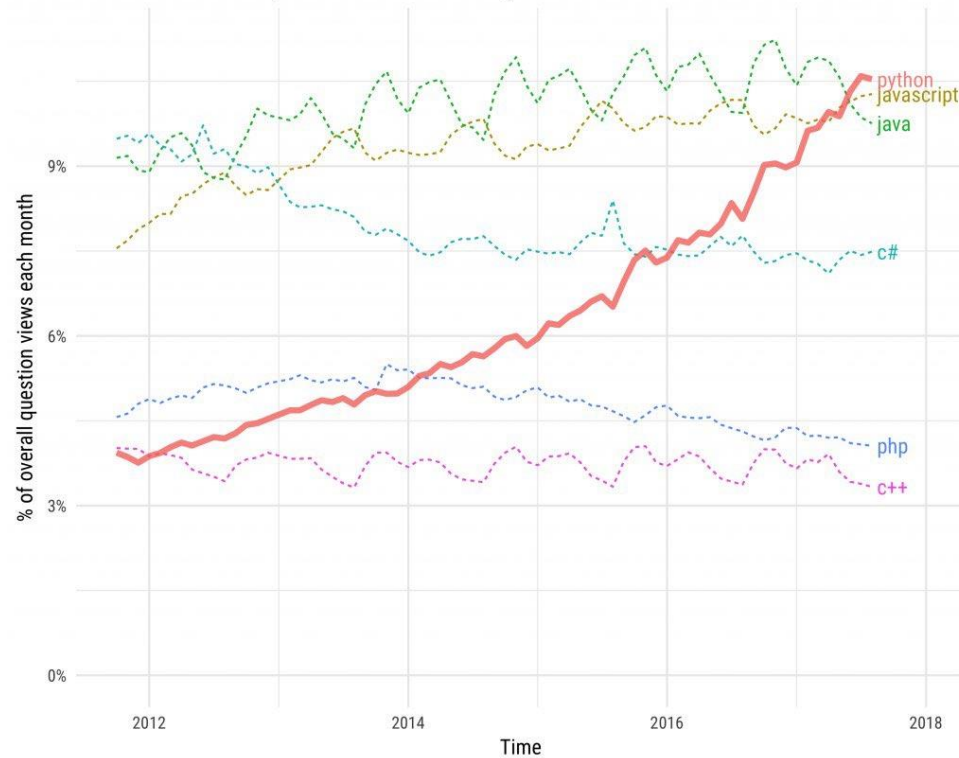


# Introducción a Python

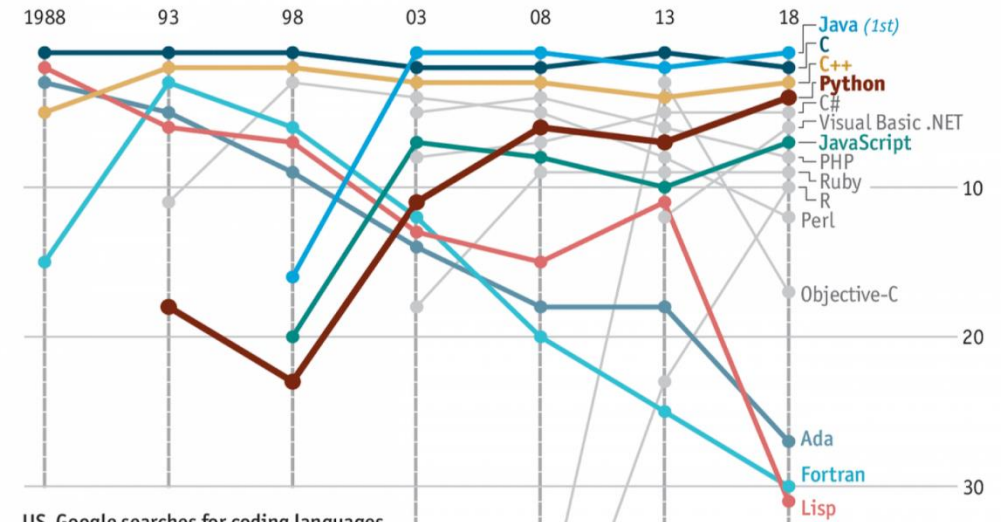
## • ¿Por qué Python?

### Growth of major programming languages

Based on Stack Overflow question views in World Bank high-income countries

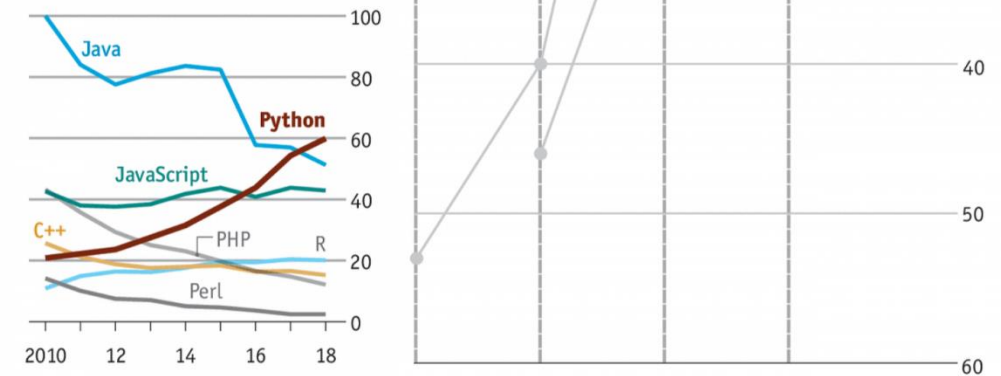


### Ranking of programming languages\*



### US, Google searches for coding languages

100 = highest annual traffic for any language



Source: TIOBE, Google Trends

\*Ranked by global search-engine popularity

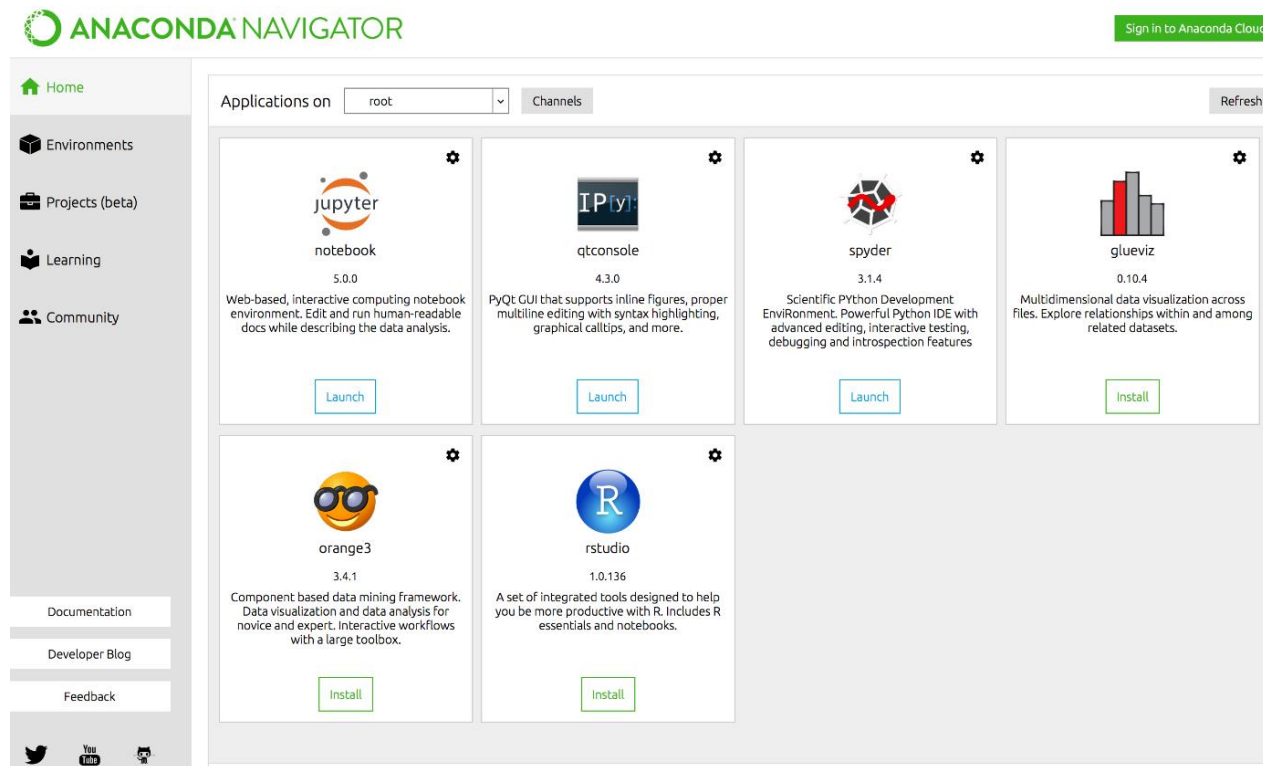
# Introducción a Python

- Entorno de trabajo
  - Python versión 3.7 : sintaxis Python 3
  - Nos centraremos en el entorno Anaconda
    - **Anaconda Navigator** y **Anaconda Prompt** para la gestión de aplicaciones
    - IDE de desarrollo y prueba: **Spyder**
    - Notebooks de **Jupyter** para la presentación de código
    - Herramientas **pip** y **conda** para la gestión de paquetes



# Introducción a Python

- Entorno de trabajo
  - Nos centraremos en el entorno Anaconda
    - **Anaconda Navigator** y **Anaconda Prompt** para la gestión de aplicaciones



# Introducción a Python

- Entorno de trabajo
  - Nos centraremos en el entorno Anaconda
    - IDE de desarrollo y prueba: **Spyder**



The screenshot shows the Spyder Python IDE interface. The main editor displays a Python script for data manipulation using pandas. The script includes comments in Swedish and English, and code for reading a CSV file, creating a DataFrame, and performing operations like grouping and reversing rows. The variable explorer on the right shows the contents of the 'frame' DataFrame, including columns like 'Date', 'Informed', 'Sex', 'Age', and 'EduYears'. The console at the bottom shows the execution of the script, including a warning about setting a slice from a DataFrame.

```
1 # -*- coding: utf-8 -*-
2 """
3 Created on Wed Apr 29 09:29:09 2015
4
5 @author: erik
6 """
7
8 import pandas as pd
9
10 from pandas import DataFrame
11
12 def reverseScoring(df, high, cols):
13     df[cols] = high - df[cols]
14     return df
15
16
17 #The questionnaire is coded in Swedish. Changing stuff to english...
18 lv = ['Y1' + str(i) for i in range(1,21)]
19 firstnames, lastnames = ['Date', 'Informed', 'Sex', 'Age'], ['EduYears', 'Sub_id']
20 names = firstnames + lv + lastnames
21 frame = pd.read_csv('acs_okt.csv', skiprows=1, names=names, encoding='latin1')
22
23 print frame.head()
24
25 #Moving strings and leaving score (1-4)
26 #Doing col by col
27 for idx in range(len(lv)):
28     for i,row in enumerate(frame[lv[idx]]):
29         #need to turn row into string (was unicode type)
30         row = str(row)
31         frame[lv[idx]][i] = int(row[0])
32
33
34 #print frame[['Y1', 'Y2']]
35 frame['Sub_id'] = range(1, len(frame['Date'])+1)
36 #grouped = frame.groupby('Sex')
37
38 #Save the frame to csv
39 #frame.to_csv(path_or_buf='ACS_DATA.csv', sep=';')
40 print "Done"
41
42 toReverse = ['Y1', 'Y2', 'Y3', 'Y6', 'Y7', 'Y8', 'Y11', 'Y12', 'Y15', 'Y10',
43             'Y20']
44
45 revFrame = reverseScoring(frame, 5, toReverse)
46
47 revFrame.to_csv(path_or_buf='ACS_DATA_rev.csv', sep=';')
```

Name	Type	Size	Value
firstnames	list	4	['Date', 'Informed', 'Sex', 'Age']
frame	DataFrame	(234, 20)	Column names: Date, Informed, Sex, Age, Y1, Y2, Y3, Y4, Y5, Y6, Y7, Y8, Y9, Y10, Y11, Y12, Y15, Y16, Y17, Y18, Y19, Y20
i	int	1	9
idx	int	1	8
lv	list	20	['Y1', 'Y2', 'Y3', 'Y4', 'Y5', 'Y6', 'Y7', 'Y8', 'Y9', 'Y10', ...]
lastnames	list	2	['EduYears', 'Sub_id']
names	list	26	['Date', 'Informed', 'Sex', 'Age', 'Y1', 'Y2', 'Y3', 'Y4', 'Y5', 'Y6', ...]
row	unicode	1	1. Mycket sköllen

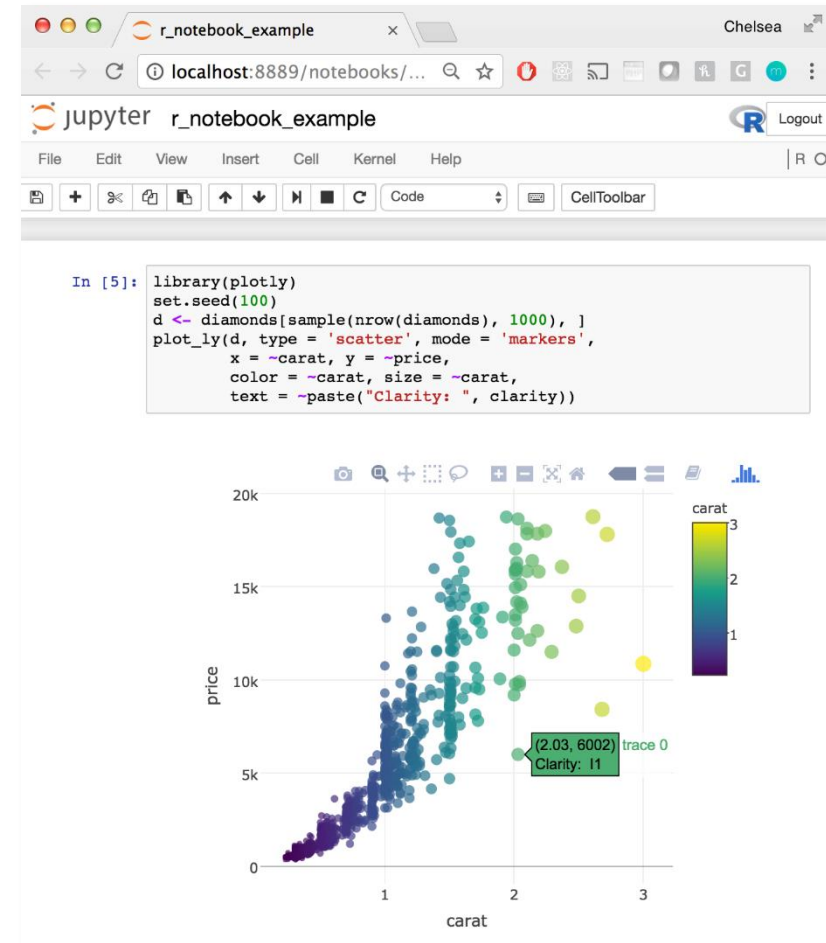
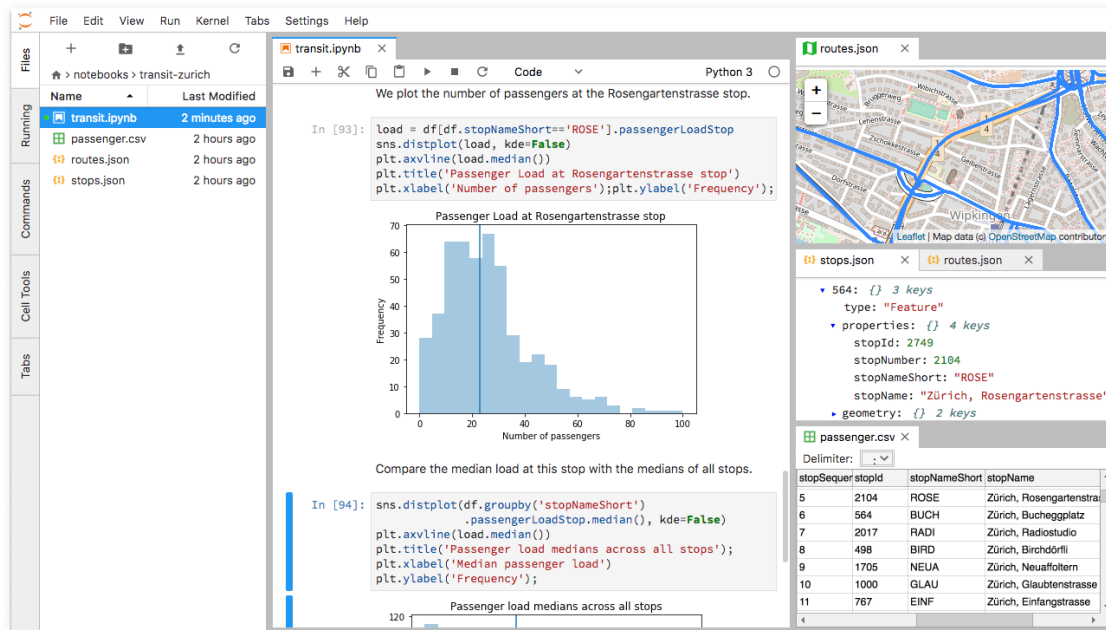
Python console output:

```
File ~/usr/local/lib/python2.7/dist-packages/spyderlib/widgets/externalshell/sitecustomize.py, line 699, in
runfile
execfile(filename, namespace)
File ~/usr/local/lib/python2.7/dist-packages/spyderlib/widgets/externalshell/sitecustomize.py, line 81, in
execfile
builtin.execfile(filename, *where)
File ~/home/erik/Dokument/Programming/Python/datawrangling/questionnaire.py, line 30, in <module>
row = str(row)
UnicodeEncodeError: 'ascii' codec can't encode characters in position 11-12: ordinal not in range(128)
/home/erik/Dokument/Programming/Python/datawrangling/questionnaire.py:31: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame
See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/indexing.html#indexing-view-versus-copy
frame[lv[idx]][i] = int(row[0])
In [2]:
```



# Introducción a Python

- Entorno de trabajo
  - Nos centraremos en el entorno Anaconda
    - Notebooks de Jupyter para la presentación de código
    - Ejecución y visualización de resultados online



# Introducción a Python

- Entorno de trabajo
  - Nos centraremos en el entorno Anaconda
    - Herramientas **pip** y **conda** para la gestión de paquetes

```
C:\Users\user>python -m pip install --upgrade pip
Requirement already up-to-date: pip in c:\python27\lib\site-packages

C:\Users\user>pip list
You are using pip version 7.1.0, however version 7.1.2 is available.
You should consider upgrading via the 'python -m pip install --upgrade pip' command.
pbr (1.8.1)
pip (7.1.0)
scikit-learn (0.17)
setuptools (18.1)
six (1.10.0)
sklearn (0.0)
stevedore (1.9.0)
virtualenv (13.1.2)
virtualenv-clone (0.2.6)
virtualenvwrapper (4.7.1)

C:\Users\user>
```

# Introducción a Python

- Primer programa: “Hola Mundo”

```
holaMundo.py *  
1  if __name__ == "__main__":  
2      print('¡Hola mundo!')  
3
```

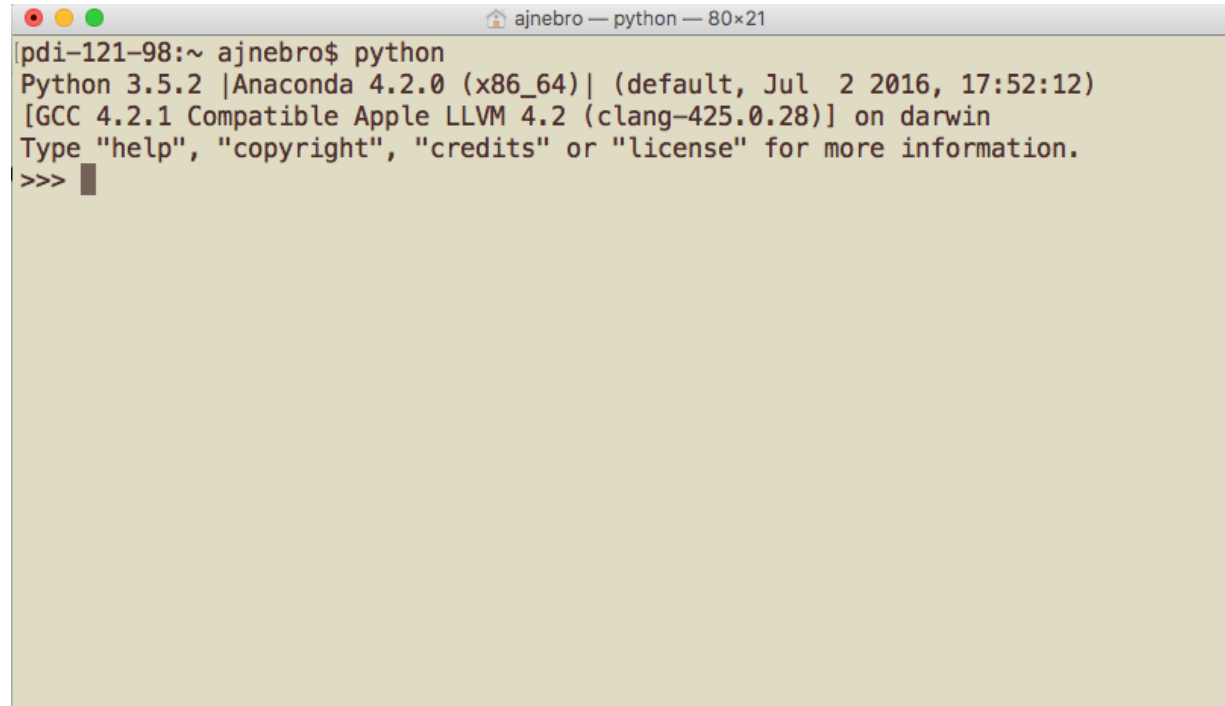
- En el interprete

```
>> python holaMundo.py
```

```
¡Hola mundo!
```

# Introducción a Python

- Primer programa: “Hola Mundo”
  - Modo interactivo
    - Se puede acceder al intérprete de Python mediante la consola:



```
ajnebro — python — 80x21
[pdi-121-98:~ ajnebro$ python
Python 3.5.2 |Anaconda 4.2.0 (x86_64)| (default, Jul 2 2016, 17:52:12)
[GCC 4.2.1 Compatible Apple LLVM 4.2 (clang-425.0.28)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> ]
```

# Introducción a Python

- Primer programa: Números
  - El intérprete actúa como calculadora

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
>>> 8 / 5 # la división siempre retorna un número de punto flotante
1.6
```

- División

```
>>> 17 / 3 # la división clásica retorna un punto flotante
5.666666666666667

>>> 17 // 3 # la división entera descarta la parte fraccional
5
```

# Introducción a Python

- Primer programa: Variables

- Utilizamos el signo (=) para asignar un valor a una variable
- Python asigna de forma dinámica el “tipo” de la variable: int, float, natural, bool, char, etc.

```
>>> ancho = 20
>>> largo = 5 * 9
>>> ancho * largo
900
```

- Si una variable no está "definida" (con un valor asignado), intentar usarla producirá un error:

```
>>> n
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

```
>>> 'huevos y pan' # comillas simples
'huevos y pan'

>>> 'doesn\'t' # usa \' para escapar comillas simples...
"doesn't"

>>> "doesn't" # ...o de lo contrario usa comillas doblas
"doesn't"

>>> "'Si," le dijo.'
"'Si," le dijo.'

>>> "\"Si,\" le dijo."
"'Si,\" le dijo.'"

>>> "'Isn't," she said.'
"'Isn't," she said.'"

```

# Introducción a Python

- Primer programa: Cadenas de caracteres

- La función `print()`

```
>>> "Isn\t," she said.  
"Isn\t," she said.  
  
>>> print("Isn\t," she said.)  
"Isn\t," she said.
```

- Carácter salto de línea `\n`

```
>>> s = 'Primera línea.\nSegunda línea.'    # \n significa nueva línea  
>>> s                                         # sin print(), \n es incluido en la salida  
'Primera línea.\nSegunda línea.'  
  
>>> print(s)                                # con print(), \n produce una nueva línea  
Primera línea.  
Segunda línea.
```



# Introducción a Python

- Primer programa: Cadenas de caracteres
  - Cadenas de texto con múltiples líneas ("""...""")

```
>>> print("""\
Uso: algo [OPTIONS]
    -h                Muestra el mensaje de uso
    -H nombrehost     Nombre del host al cual conectarse
""")
```

- Obtiene como salida directamente:

```
Uso: algo [OPTIONS]
    -h                Muestra el mensaje de uso
    -H nombrehost     Nombre del host al cual conectarse
```

- Las cadenas de texto pueden ser concatenadas (pegadas juntas) con el operador + y repetidas con \*:

```
>>> 3 * 'un' + 'ium'           # 3 veces 'un', seguido de 'ium'
'unununium'
```

# Introducción a Python

- Primer programa: Cadenas de caracteres

- Las cadenas de texto se pueden indexar (subíndices), el primer carácter de la cadena tiene el índice 0

```
>>> palabra = 'Python'
>>> palabra[0]           # caracter en la posición 0
'P'
>>> palabra[5]           # caracter en la posición 5
'n'
>>> palabra[-1]          # último caracter
'n'
>>> palabra[-2]          # ante último caracter
'o'
>>> palabra[-6]
'P'
```

- Si intentamos acceder fuera del rango, dará error

```
>>> palabra[42]           # la palabra solo tiene 6 caracteres
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

# Introducción a Python

- Primer programa: Cadenas de caracteres

- También podemos indexar subcadenas

```
>>> palabra[0:2]      # caracteres desde la posición 0 (incluida) hasta la 2 (excluida)
'Py'
>>> palabra[2:5]      # caracteres desde la posición 2 (incluida) hasta la 5 (excluida)
'tho'
```

- Como el primero es siempre incluido, y que el último es siempre excluido. Esto asegura que  $s[:i] + s[i:]$  siempre sea igual a  $s$ :

```
>>> palabra[:2]        # caracteres desde el principio hasta la posición 2 (excluida)
'Py'
>>> palabra[4:]        # caracteres desde la posición 4 (incluida) hasta el final
'on'
>>> palabra[-2:] # caracteres desde la ante-última (incluida) hasta el final
'on'
```

```
+---+---+---+---+---+---+
| P | y | t | h | o | n |
+---+---+---+---+---+---+
 0  1  2  3  4  5  6
-6 -5 -4 -3 -2 -1
```

# Introducción a Python

- Primer programa: Cadenas de caracteres
  - Obtener la longitud de una cadena mediante la función len()

```
>>> s = 'supercalifrastilisticoespialidoso'  
>>> len(s)  
33
```

Resumen: El tipo de una variable lo define el propio valor de la variable:

```
variable = 'cadena'           # Tipo cadena  
  
variable = 5                   # Tipo entero  
  
variable = 0.05                # Tipo real  
  
variable = None                # Ausencia de valor  
  
type(variable)                # Para conocer el tipo de la variable
```

# Introducción a Python

- Indentación

```
public boolean esPrimo(int numero){  
    if (numero < 2){  
        return false;  
    }  
    int contador = 2;  
    boolean primo=true;  
    while ((primo) && (contador < numero)) {  
        if (numero % contador == 0) {  
            primo = false;  
        }  
        contador++;  
    }  
    return primo;  
}
```

JAVA

```
def es_primo(numero):  
    if numero < 2:  
        return False  
    contador = 2  
    primo = True  
    while primo and contador < numero:  
        if numero % contador == 0:  
            primo = False  
        contador += 1  
    return primo
```

PYTHON

En Python, el ámbito de las funciones, clases, métodos, etc lo define la indentación o sangrado. Por tanto, no se utilizan llaves para definir el ámbito

# Introducción a Python

- Tipos de datos
  - Tipos simples y compuestos

```
variable = 'cadena'           # Tipo cadena
variable = 5                   # Tipo entero
variable = 0.05                # Tipo real
variable = [1, 'cadena', 1.6]  # Lista
variable = (1, 'cadena', 1.6)  # Tupla
variable = {'x': 2, 'y': 1, 'z': 4}  # Diccionario
variable = set(1, 2, "hola")      # Conjunto
variable = None                 # Ausencia de valor
type(variable)                 # Para conocer el tipo de la variable
```

- Tipos de datos

- Tuplas

- Pueden contener elementos de igual o distinto tipo
    - Son inmutables
    - Se declaran usando paréntesis ()

```
>>> tupla = (1, "Hola", 4, False, 55)      # Creacion
>>> tupla
(1, 'Hola', 4, False, 55)
>>> print(tupla[4])
55
>>> tupla[2] = 5                          # No se puede modificar
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> tupla = (1, 2, 3) + (4, 5, 6)          # Operacion de concatenacion
>>> tupla
(1, 2, 3, 4, 5, 6)
>>>
```

- Tipos de datos

- Listas

- Pueden contener elementos de igual o distinto tipo
    - Son mutables
    - Se declaran usando corchetes []

```
>>> lista = [3, "hola", False, 55] # Creacion
>>> lista
[3, 'hola', False, 55]
>>> print(lista[3]) # Acceso a un campo
55
>>> lista[0] = "cadena" # Modificacion
>>> print(lista[0])
cadena
>>> lista.append(502) # Añadir al final
>>> lista
['cadena', 'hola', False, 55, 502]
```



- Tipos de datos
  - Diccionarios
    - Son listas de pares (clave, valor)
    - Son mutables
    - Se declaran usando llaves {}

```
>>> dict = {'clave':'dato', 'otra_clave':155}      # Creacion
>>> print(dict['otra_clave'])                      # Acceso
155
>>> dict['clave'] = "texto"                        # Modificacion
>>> print(dict['clave'])
texto
>>> dict['nueva_clave'] = 'nuevo_valor'            # Extension
>>> print(dict['nueva_clave'])
nuevo_valor
>>> dict
{'otra_clave': 155, 'nueva_clave': 'nuevo_valor', 'clave': 'texto'}
>>> for key, value in dict.items():                # Acceso claves y valores
...     print(key, value)
...
otra_clave 155
nueva_clave nuevo_valor
clave texto
```

- Tipos de datos
  - Conjuntos
    - Contienen valores no repetidos
    - Son mutables
    - Admiten operaciones de conjuntos (unión, intersección, etc.)

```
>>> conjunto = set([1, 2, "hola"])      # Creación
>>> conjunto
{1, 2, 'hola'}
>>> conjunto = set([1, 2, "hola", 1])   # Incluir duplicado (se ignora)
>>> conjunto
{1, 2, 'hola'}
>>> conjunto.add("adios")               # Ampliar
>>> conjunto
{1, 2, 'hola', 'adios'}
>>> conjunto2 = set([1, "hola"])
>>> conjunto & conjunto2                # Intersección
{1, 'hola'}
>>> conjunto | conjunto2                # Union
{1, 2, 'hola', 'adios'}
>>> conjunto - conjunto2                # Diferencia
{2, 'adios'}
```

- Operadores lógicos

```
>>> edad = 15
>>> if edad >= 12 and edad <= 18:                                     # And
...     print("Edad comprendida entre 12 y 18")
...
Edad comprendida entre 12 y 18
>>>
>>> x1 = 2
>>> x2 = 3.5
>>> if x1 < 5 or x2 < 5:                                             # Or
...     print("x1 o x2 son menores que 5")
...
x1 o x2 son menores que 5
>>> if not x1 > x2:                                                  # Not
...     print("x1 no es mayor que x2")
...
x1 no es mayor que x2
```

- Expresiones

```
==      Igual a

!=      Distinto de (<> está obsoleto)

>       Mayor que

<       Menor que

>=     Mayor o igual que

<=     Menor o igual que

is      Igual a (solo para hacer comparaciones
        entre referencias de objetos o para saber
        si es None, True o False)
```

# Introducción a Python

- Condiciones

```
>>> var = 100
>>> if var == 200:
...     print("var es 200")
... elif var == 150:
...     print("var es 150")
... elif var == 100:
...     print("var es 100")
... elif var is None:
...     print("var es None")
... else:
...     print("var no es ningún valor anterior")
...
var es 100
```

## Control de flujo

- Bucle For:

```
>>> colores = ["rojo", "azul", "verde"]
>>> for color in colores:
...     print("Color: ", color)
...
Color:  rojo
Color:  azul
Color:  verde
>>> numeros = (1, 2, 3)
>>> for numero in numeros:
...     print(numero)
...
1
2
3
```

- Bucle while

```
>>> colores = ["rojo", "azul", "verde"]
>>> count = 0
>>> while count < 3:
...     print("Color: ", colores[count])
...     count += 1
...
Color:  rojo
Color:  azul
Color:  verde
```

- En Python no existen ni switch ni do while

## POO: Clases

Constructor

Atributos (públicos)

```
class Coche(object):  
    attr_clase = 'atributo estático o de clase'  
  
    def __init__(self, combustible, bateria):  
        self.combustible = combustible  
        self.bateria = bateria  
  
    def repostar(self, combustible):  
        self.combustible += combustible  
  
    def recargar(self, bateria):  
        self.bateria += bateria
```

Métodos

```
# Crear instancia  
coche = Coche(2, 5)
```

```
# Usar método  
coche.repostar(1)
```

```
# Mostrar combustible (atributo)  
print(coche.combustible)
```

```
# Acceder a atributo de clase  
print(Coche.attr_clase)
```

## POO: Atributos protegidos

```
class Coche(object):
```

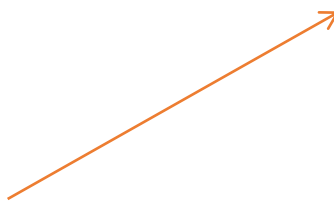
```
    def __init__(self, combustible, bateria):
        self._combustible = combustible
        self._bateria = bateria
```

Los atributos que comienzan por un guión bajo son considerados protegidos. Solo se debería acceder dentro de la clase o las subclases

# Se puede acceder al atributo protegido combustible desde el exterior, pero no se debe. Cualquier IDE muestra un aviso de que tal acción no es adecuada

```
coche = Coche(12, 8)
print(coche._combustible)
```

**Se puede acceder al atributo protegido exteriormente, pero no se debe**





## POO: Atributos privados

```
class Coche(object):
```

```
    def __init__(self, combustible, bateria):
        self.__combustible = combustible
        self.__bateria = bateria
```

Los atributos que comienzan por dos guiones bajos son considerados privados.

# NO se puede acceder al atributo privado  
combustible

```
coche = Coche(12, 8)
```

```
print(coche.__combustible)
```

**Error**



## POO: getters y setters

```
class Coche(object):  
    def __init__(self, combustible, bateria):  
        self.combustible = combustible  
        self.bateria = bateria
```

PYTHON

**En Python, no definimos métodos *getters/setters*, sino que se accede directamente a los atributos, que por lo general se definen públicos.**

```
public class Coche {  
    private int combustible;  
    private int bateria;  
  
    public int getCombustible() {  
        return combustible;  
    }  
    public void setCombustible(int combustible) {  
        this.combustible = combustible;  
    }  
    public int getBateria() {  
        return bateria;  
    }  
    public void setBateria(int bateria) {  
        this.bateria = bateria;  
    }  
}
```

JAVA

## POO: getters y setters

- Python sigue el **Uniform Access Principle**, el cual obliga a que el acceso a los atributos se deba hacer de una forma uniforme en todos los casos.
- Por tanto, si definimos métodos getter/setter incumplimos dicho principio, ya que tendríamos varias formas de acceder a los atributos dependiendo de si son públicos o privados:

```

coche = Coche(5, 7)
print(coche.combustible) # forma de acceso común en Python
print(coche.get_bateria()) # otra forma de acceso
```

 **MAL**

## POO: getters, setters y properties

```
class Circulo(object):  
    def __init__(self):  
        self.__radio = None
```

```
circulo = Circulo()  
circulo.radio = -1 # set  
print(circulo.radio) # get
```

@property

```
def radio(self):  
    print('Accediendo a radio')  
    return self.__radio
```

@radio.setter

```
def radio(self, radio):  
    if radio < 0:  
        raise ValueError("radio debe ser un numero no negativo")  
    self.__radio = radio
```

Con *property* accedemos a los métodos getter/setter como si fueran accesos al atributo (no incumplimos el *Uniform Access Principle*). El uso de *property* es comparable a los métodos *getter/setter* en JAVA, pero en Python es más apropiado usarlo sólo cuando se requiera lógica en los accesos del atributo.

## POO: getters, setters y properties

```
class Circulo(object):
    def __init__(self):
        self.__radio = None

    def __get_radio(self):
        print('Accediendo a radio')
        return self.__radio

    def __set_radio(self, radio):
        if radio < 0:
            raise ValueError("radio debe ser un número no negativo")
        self.__radio = radio

radio = property(fget=__get_radio, fset=__set_radio)
```

→ Otra forma de definir property

## POO: Herencia

```
class A(object):
    def __init__(self):
        print('Soy constructor A')
```

```
class B(A):
    def __init__(self):
        print('Soy constructor B')
        super(B, self).__init__()
```

```
b = B()
```

```
# Resultado:
Soy constructor B
Soy constructor A
```

La clase B hereda de A

Llama al constructor de la  
clase A

## POO: Herencia

### Method Resolution Order (MRO)

`super()` recorre el **MRO**, que es el orden de herencia, y delega en la primera clase que encuentra por encima de *Humano* que define el método `que_soy()`.

Dicho orden da prioridad a los padres inmediatos antes que a los abuelos. Por tanto, si *Mamífero* no tuviera definido el método `que_soy()`, se ejecutaría el método `que_soy()` de *Sapiens*, y si ésta no lo tuviera, se ejecutaría el de la clase *Animal*, que es el abuelo.

```
class Animal(object):
    def que_soy(self):
        print("Soy Animal")
```

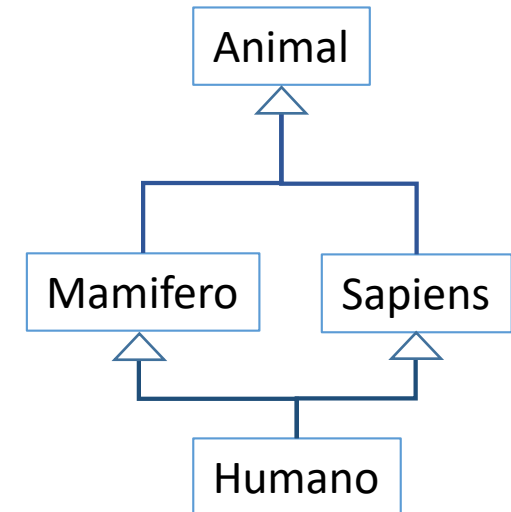
```
class Mamifero(Animal):
    def que_soy(self):
        print("Soy Mamifero")
```

```
class Sapiens(Animal):
    def que_soy(self):
        print("Soy Sapiens")
```

```
class Humano(Mamifero, Sapiens):
    def que_soy(self):
        super(Humano, self).que_soy()
```

```
humano = Humano()
humano.que_soy()
print("Orden de herencia:", Humano.__mro__)
```

Para ver el orden de herencia



### Herencia múltiple:

La clase *Humano* hereda de *Mamífero* y *Sapiens*

```
>>> Humano.__mro__
Orden de herencia:
(<class '__main__.Humano'>,
 <class '__main__.Mamifero'>,
 <class '__main__.Sapiens'>,
 <class '__main__.Animal'>, <class 'object'>)
```

## POO: Métodos estáticos

```
class Calculadora(object):  
    def __init__(self, modelo):  
        self.modelo = modelo  
  
    @staticmethod  
    def suma(x, y):  
        return x + y  
  
print(Calculadora.suma(2, 4))
```

**Definimos @staticmethod cuando el método no trabaja con los atributos, cuya lógica es propia de la clase, no de la instancia**



## POO: Métodos de clase

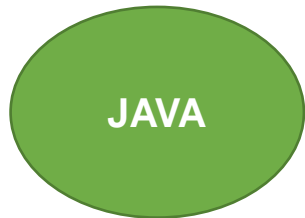
```
class Atomo(object):  
    def __init__(self, nombre):  
        self.nombre = nombre  
  
    @classmethod  
    def fision(cls):  
        h1 = cls("Isotopo primero")  
        h2 = cls("Isotopo segundo")  
        return h1, h2
```

```
atomo = Atomo("Uranio")  
hijo1, hijo2 = atomo.fision()
```

Usamos **@classmethod** cuando queremos devolver objeto/s de la misma clase. Es como definir otro constructor, pero solo tenemos acceso a la clase, no al objeto.

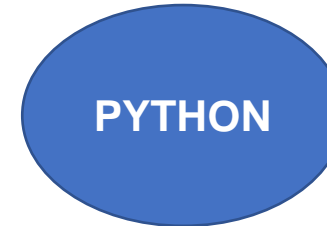
Se usa **cls** en lugar de **self** por convención, ya que hacemos referencia a la clase, no al objeto

## POO: Sobrecarga de métodos



```
public class Raton {
    String nombre;

    public Raton() {
        this.nombre = "";
    }
    public Raton(String nombre) {
        this.nombre = nombre;
    }
}
```



```
class Raton:
    def __init__(self, nombre=""):
        self.nombre = nombre
```

Valor por defecto

```
# Creamos instancias con y sin argumento
raton = Raton('Perez')
raton2 = Raton()
```

**En Python podemos simular la sobrecarga de métodos con el uso de valores por defecto en los argumentos**

## POO: Interfaces

"Abstract Base Class" (ABC),  
Combina la estructura de una  
interfaz con la posibilidad de  
incluir alguna mínima  
implementación

```
1 from abc import abstractmethod, ABCMeta
2
3 class ClassInterface(metaclass=ABCMeta):
4     @abstractmethod
5     def read(self, maxbytes = 1):
6         pass
7
8     @abstractmethod
9     def write(self, data: int):
10        pass
11
12
13 class ClassImplementation(ClassInterface):
14     def __init__(self):
15         pass
16
17     def read(self, maxbytes = 1):
18         print(maxbytes)
19
20     def write(self, data: int):
21         print(data)
22
23 if __name__ == '__main__':
24     a = ClassImplementation()
25     a.read()
26     a.write(23)
```

## POO: Excepciones

```

1 class Numero(object):
2     def __init__(self):
3         self.numero = -5
4
5     def comprobar_numero(self):
6         if self.numero < 0:
7             raise ValueError('El número es negativo')
8
9
10 if __name__ == '__main__':
11     try:
12         numero = Numero()
13         numero.comprobar_numero()
14     except ValueError:
15         print("Oops!  Es un número negativo")

```

Lanza excepción



```

try:
    ...
except ArithmeticError as err:
    ...
except ValueError as err:
    ...
except Exception as err:
    ...

```

## POO: Excepciones

```

1 class TextoDemasiadoCortoError(ValueError):
2     pass
3
4
5 # función
6 def validate(texto):
7     if len(texto) < 10:
8         raise TextoDemasiadoCortoError(texto)
9
10
11 if __name__ == '__main__':
12     validate(texto='hola')

```

← Excepción personalizada

## Funciones predefinidas

- Python cuenta con funciones predefinidas:
  - `min()`: devuelve el máximo de un iterable
  - `max()`: devuelve el máximo de un iterable
  - `round()`: redondea un número decimal
  - `isinstance()`: comprueba si un objeto es de una clase
  - `len()`: devuelve la longitud de una secuencia o colección
  - `int()`: convierte a entero
  - `str()`: convierte a cadena
  - `sorted()`: ordena un iterable, devolviendo una lista ordenada
  - ...

<https://docs.python.org/3.5/library/functions.html>

- Módulos

- Un módulo es un archivo conteniendo definiciones y declaraciones de Python
- El nombre del archivo es el nombre del módulo con el sufijo .py agregado. Por ejemplo:  
**fibonacci.py**

```
# módulo de números Fibonacci
def fib(n): # escribe la serie Fibonacci hasta n
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b
    print()

def fib2(n): # devuelve la serie Fibonacci hasta n
    resultado = []
    a, b = 0, 1
    while b < n:
        resultado.append(b)
        a, b = b, a+b
    return resultado
```

# Introducción a Python

- Módulos

- Una vez guardado, es posible importar el módulo desde el intérprete de Python (o desde otro módulo) con la siguiente orden

```
>>> import fibo
```

- Ahora podemos acceder a las funciones

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987

>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]

>>> fibo.__name__
'fibo'
```



# Introducción a Python

- Módulos

- Se puede también ejecutar los módulos de Python como scripts (desde fuera del interprete!!)

```
python fibo.py <argumentos>
```

- El código en el módulo será ejecutado, tal como si se hubiese importado, pero con `__name__` con el valor de `"__main__"`. Eso significa que agregando este código al final del módulo:

```
if __name__ == "__main__":  
    import sys  
    fib(int(sys.argv[1]))
```

- Se puede hacer que el archivo sea utilizable tanto como script, como módulo importable

```
$ python fibo.py 50  
1 1 2 3 5 8 13 21 34
```

# Introducción a Python

- Módulos

- Ejercicio: desarrollar un módulo llamado **círculos.py**, que dado un radio, calcule funciones básicas como el **perímetro** y el **área**
- Podemos utilizar la librería estándar de matemáticas

```
import math
```

```
acos(x), asin(x), atan(x), atan2(x, y), ceil(x), cos(x), cosh(x), exp(x), fabs(x),  
floor(x), fmod(x, y), frexp(x), hypot(x, y), ldexp(x, y), log(x), log10(x), modf(x),  
pow(x, y), sin(x), sinh(x), sqrt(x), tan(x), tanh(x).
```

Y las constantes pi y e.

## Estilo de codificación

### JAVA

```
// Nombre de clase  
public class MotorTurbo {  
  
// Nombre de método  
public void escribirHola(){
```

### PYTHON (PEP8)

```
# Nombre de clase  
class MotorTurbo(object):  
  
// Nombre de método  
def escribir_hola(self):
```

**Python usa PEP8 como estilo de código. Hay muchas más normas de estilo en PEP8: espacios entre los métodos, entre los argumentos, etc**

## Gestion de dependencias

- Las dependencias en Python se gestionan con el comando pip

Buscar dependencia

```
pip search dependencia
```

Instalar dependencia

```
pip install dependencia
```

Desinstalar dependencia

```
pip uninstall dependencia
```

Generar fichero *requirements.txt* con todas las dependencias instaladas

```
pip freeze > requirements.txt
```

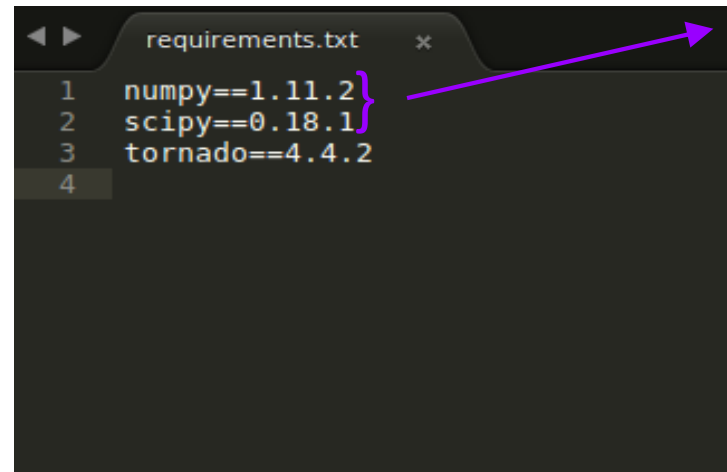
Instalar todas las dependencias definidas en *requirements.txt*

```
pip install -r requirements.txt
```

El uso de un IDE como *Pycharm* permite realizar estas operaciones a través de una interfaz gráfica

## Gestion de dependencias

### Ejemplo de *requirements.txt*



```
requirements.txt
1  numpy==1.11.2
2  scipy==0.18.1
3  tornado==4.4.2
4
```

### Dependencias:

- *numpy*: módulo para cómputo científico
- *scipy*: módulo para matemáticas, ciencia e ingeniería
- *tornado*: módulo para conexiones asíncronas

## Gestion de dependencias

- **Si se usa el intérprete python del sistema, todas las dependencias que se instalen o eliminen se hacen globalmente, en el propio sistema local**
- Para evitar esto, se suele usar virtualenv, que permite crear un entorno virtual para cada proyecto, incorporando un intérprete y la posibilidad de instalar dependencias particulares para para el proyecto

## Gestion de dependencias

Crear entorno virtual con el nombre de directorio *venv*

```
virtualenv venv --distribute
```

Acceder al entorno virtual

```
source venv/bin/activate
```

Una vez dentro del entorno virtual, todas las dependencias que instalemos con ***pip*** se instalarán dentro del entorno virtual. De igual forma, si generamos el fichero *requirements.txt*, éste tendrá declarado todas las dependencias instaladas en el entorno virtual

El uso de un IDE como *Pycharm* permite realizar estas operaciones a través de una interfaz gráfica

## Pruebas unitarias

```
class Triangulo(object):

    def __init__(self, lado1=None, lado2=None,
                  lado3=None):
        self.lado1 = lado1
        self.lado2 = lado2
        self.lado3 = lado3

    def es_equiletero(self):
        if self.lado1 is None or self.lado1 is None or \
            self.lado1 is None:
            return False
        elif self.lado1 == self.lado2 and \
            self.lado2 == self.lado3:
            return True
        else:
            return False
```

```
import unittest
```

```
class TrianguloTestCase(unittest.TestCase):
    # Se ejecuta justo antes de cada test
    def setUp(self):
        print("setUp: INICIANDO TEST")
        # Instanciamos de la clase Triangulo
        self.triangulo = Triangulo()

    # Se ejecuta despues de cada test
    def tearDown(self):
        print("tearDown: FINALIZANDO TEST")

    def test_es_equilatero(self):
        print("Ejecutando test1")
        self.triangulo.lado1 = 4
        self.triangulo.lado2 = 4
        self.triangulo.lado3 = 8
        resultado = self.triangulo.es_equiletero()
        self.assertFalse(resultado)
```

Librería *unittest* para hacer pruebas unitarias

*setUp* se ejecuta antes de cada método test

*tearDown* se ejecuta después de cada método test

El nombre de cada método test debe comenzar con *test*



## Pruebas unitarias

- Módulos para testing
  - unittest: para crear los tests
  - Nose: para ejecutar tests de forma automática
  - Coverage: para la medición de la cobertura de código
  - Mock: para crear objetos mocks