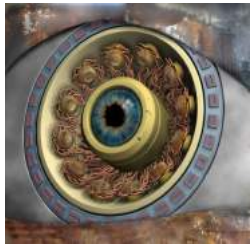


Introducción a Python

Visión por Computador

ETSI Informáticos

Universidad Politécnica de Madrid

Luis Baumela

Departamento de Inteligencia Artificial

POLITÉCNICA

Historia

- Monty Python's Flying Circus
- Guido van Rossum
  Benevolent Dictator For Life
- Diciembre 1989 (0.0)
  Octubre 2000 (2.0)
  Diciembre 2008 (3.0)

Introducción
¿Para qué?

Python is a high-level general-purpose programming language that can be applied to many different classes of problems.

Comes with a large standard library:

- string processing (regular expressions, ...)
- Internet protocols (HTTP, FTP, SMTP, XML-RPC, POP, IMAP, CGI programming)
- software engineering (unit testing, logging, profiling,..)
- operating system interfaces (system calls, TCP/IP sockets,...)

Look at the table of contents for The Python Standard Library to get an idea of what's available

Basic concepts
Citas, comentarios

"A powerful programming language with huge community support."

Nature, vol. 518:125-126, Feb 2015

"It seems to me that Java is designed to make it difficult for programmers to write bad code, while Python is designed to make it easy to write good code"

Magnus Lycka, software consultant

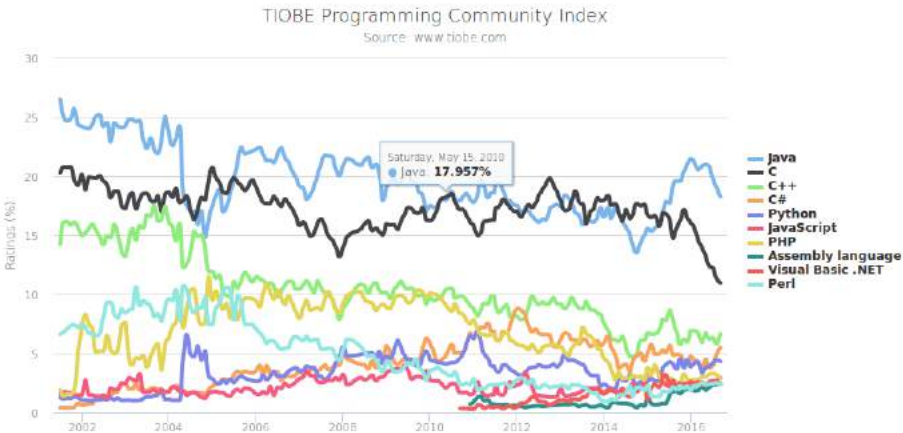
"Python plays a key role in our production pipeline. Without it a project the size of Star Wars: Episode II would have been very difficult to pull off. From crowd rendering to batch processing to composing, Python binds all things together"

Tommy Burnefe, Senior Technical Director, Industrial Light & Magic

"Python has been an important part of Google since the beginning, and remains so as the system grows and evolves. Today dozens of Google engineers use Python, and we're looking for more people with skills in this language."

Peter Norvig, director of search quality at Google, Inc

Introducción
Popularidad



21-09-2016

Introducción
¿Para qué se usa?

- Aplicaciones y web:
  - Maya, MoXonBuilder, Sonimage, Cinema 4D, BodyPaint 3D, Blender, GIMP, Inkscape, Scribus, Paint Shop Pro, GNU GDB, Baflefield 2, CivilizaXon 4, Youtube, Yahoo Maps, Google search engine
- Computación científica:
  - NASA, CERN, National Weather Service, ...

Introducción
Algunas características

- Simple y legible

  “ el código se lee muchas más veces de las que se escribe”

  sintaxis minimalista, dispersa, concisa
- Productivo

  el código pythonic es entre 2 y 10 veces más breve que en C/C++/Java
- Multiparadigma

  OOP, estructurado, funcional, ...
- Compacto

  cabe en la cabeza de un humano normal
- Sencillo para principiantes
- Fácil de embeber y muy fácil de extender
- La máquina del tiempo de Guido

Paquetes habituales del entorno

Herramientas habituales del entorno Python:

- **`NumPy` (mathematical arrays)**,
- `SciPy` (linear algebra, differential equations, signal processing and more),
- `SymPy` (symbolic mathematics),
- `matplotlib` (graph `plotting`),
- `Pandas` (data analysis).
- `Cython` (code optimization, “c” interface)
- **`Ipython` (advanced shell and working environment)**

Otras herramientas:

- `OpenCV`, for computer vision
- **`Scikit-Learn`, for machine learning**,
- `Biopython`, for bioinformatics,
- `PsychoPy` for psychology and neuroscience
- `Astropy` for astronomers.

Existen dos versiones activas

- 2.7
- 3.4
  - Las versiones 3.x no son compatibles con 2.x

En este curso utilizaremos la versión 2.7

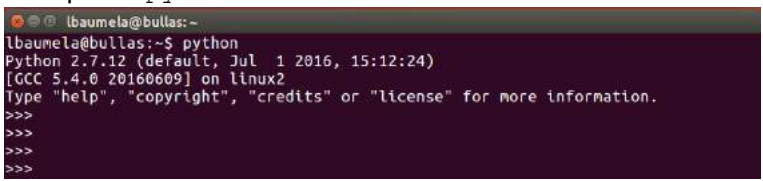
¿Dónde conseguirlo?

- Para Debian y Ubuntu, paquetes
  - python, python-numpy, python-scipy, python-matplotlib, ipython, ~~python-opencv,~~ python-sklearn
- Anaconda
  - Python + herramientas + bibliotecas científicas
  - Para Linux, Mac OS X y Windows
  - No incluye OpenCV, pero se puede instalar

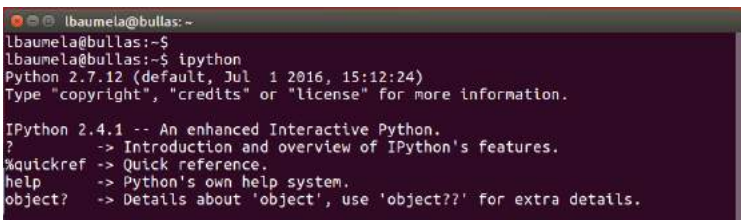
El intérprete de Python

Python es un lenguaje interpretado

- Intérprete `python`



- Intérprete `ipython`



```
x = 34 - 23              # A comment
y = "Hello"              # Another one.
z = 3.45
if z == 3.45 or y == "Hello":
    x = x + 1
    y = y + " World"     # String concat.
print x
print y
```

Sobre los espacios en blanco

Whitespace is meaningful in Python: especially indentation and placement of newlines.

- Use a newline to end a line of code.
- Use \ when must go to next line prematurely.
- No braces { } to mark blocks of code in Python…

Use consistent indentation instead.

- The first line with less indentation is outside of the block.
- The first line with more indentation starts a nested block
- Often a colon appears at the start of a new block.(E.g. for function and class definitions.)

- Start comments with # – the rest of line is ignored.
- Can include a documentation string as the first line of any new function or class that you define.
- The development environment, debugger, and other tools use it: it s good style to include one.

```
def my_function(x, y):
    """This is the docstring. This
    function does blah blah blah."""
    # The code would go here...
```

Numbers: Integers and floats work as you would expect

```
x = 3
print type(x) # Prints "<type 'int'>"
print x       # Prints "3"
print x + 1   # Addition; prints "4"
print x - 1   # Subtraction; prints "2"
print x * 2   # Multiplication; prints "6"
print x ** 2  # Exponentiation; prints "9"
x += 1
print x  # Prints "4"
x *= 2
print x  # Prints "8"
y = 2.5
print type(y) # Prints "<type 'float'>"
print y, y + 1, y * 2, y ** 2 # Prints "2.5 3.5 5.0 6.25"
```

Ojo con la división entera!!

Booleans: Use “and”, “or”, “not”

```
t = True
f = False
print type(t) # Prints "<type 'bool'>"
print t and f # Logical AND; prints "False"
print t or f  # Logical OR; prints "True"
print not t   # Logical NOT; prints "False"
print t != f  # Logical XOR; prints "True"
```

Secuencies: tuples, lists and strings

1. Tuple
   - A simple immutable ordered sequence of items
   - Items can be of mixed types, including collection types
2. Strings
   - Immutable
   - Conceptually very much like a tuple
3. List
   - Mutable ordered sequence of items of mixed types

All three sequence types share much of the same syntax and functionality.

- Tuples are defined using parentheses (and commas).

```
>>> tu = (23, abc , 4.56, (2,3), ‘def ‘)
```

- Lists are defined using square brackets (and commas).

```
>>> li = [ abc , 34, 4.34, 23]
```

- Strings are defined using quotes (“ ,’)

```
>>> st = “Hello World” or ‘Hello World’
```

- Access individual members with []

```
>>> tu = (23, ‘abc’ , 4.56, (2,3), ‘def’ )
>>> tu[1] # Second item in the tuple.
‘abc’
>>> t[1]
abc
>>> t[-3]
4.56
```

Slicing syntax to access sublists

```
nums = range(5)     # range is a built-in function that creates a list of integers
print nums          # Prints "[0, 1, 2, 3, 4]"
print nums[2:4]     # Get a slice from index 2 to 4 (exclusive); prints "[2, 3]"
print nums[2:]      # Get a slice from index 2 to the end; prints "[2, 3, 4]"
print nums[:2]      # Get a slice from the start to index 2 (exclusive); prints "[0,
print nums[:]       # Get a slice of the whole list; prints ["0, 1, 2, 3, 4]"
print nums[:-1]     # Slice indices can be negative; prints ["0, 1, 2, 3]"
nums[2:4] = [8, 9] # Assign a new sublist to a slice
print nums          # Prints "[0, 1, 8, 9, 4]"
```

Slicing operators make return copies of sequences

Note the difference between these two lines for sequences:

```
>>> list2 = list1 # 2 names refer to 1 ref
>>> list2 = list1[:] # Two independent copies, two refs
```

- Lists slower but more powerful than tuples.
  - Lists can be modified, and they have lots of handy operations we can perform on them.
  - Tuples are immutable and have fewer features.

- To convert between tuples and lists use the `list()` and `tuple()` functions:

```
li = list(tu)
tu = tuple(li)
```

Understanding reference semantics

- Assignment manipulates references
  - `x = y` does not make a copy of the object y references
  - `x = y` makes x reference the object y references
- Very useful; but beware!
- Example:

```
>>> a = [1, 2, 3]
>>> b = a
>>> a.append(4)
>>> print b
[1, 2, 3, 4]
```

Understanding reference semantics

- For simple built-in datatypes (integers, floats, strings), assignment behaves as you would expect:

```
>>> x = 3
>>> y = x
>>> y = 4
>>> print x
3
```

- For other data types (lists, dictionaries, user-defined types),assignment works differently.
  - These datatypes are mutable.
  - When we change these data, we do it in place.

A **dictionary** stores (key, value) pairs

```
d = {'cat': 'cute', 'dog': 'furry'}  # Create a new dictionary with some data
print d['cat']       # Get an entry from a dictionary; prints "cute"
print 'cat' in d     # Check if a dictionary has a given key; prints "True"
d['fish'] = 'wet'    # Set an entry in a dictionary
print d['fish']      # Prints "wet"
# print d['monkey']  # KeyError: 'monkey' not a key of d
print d.get('monkey', 'N/A')  # Get an element with a default; prints "N/A"
print d.get('fish', 'N/A')    # Get an element with a default; prints "wet"
del d['fish']        # Remove an element from a dictionary
print d.get('fish', 'N/A') # "fish" is no longer a key; prints "N/A"
```

A **set** is an unordered collection of distinct elements

```
animals = {'cat', 'dog'}
print 'cat' in animals   # Check if an element is in a set; prints "True"
print 'fish' in animals  # prints "False"
animals.add('fish')      # Add an element to a set
print 'fish' in animals  # Prints "True"
print len(animals)       # Number of elements in a set; prints "3"
animals.add('cat')       # Adding an element that is already in the set does nothing
print len(animals)       # Prints "3"
animals.remove('cat')    # Remove an element from a set
print len(animals)       # Prints "2"
```

Iterando sobre los tipos de datos

For loop:

- Lists

```
animals = ['cat', 'dog', 'monkey']
for animal in animals:
    print animal
# Prints "cat", "dog", "monkey", each on its own line.
```

```
animals = ['cat', 'dog', 'monkey']
for idx, animal in enumerate(animals):
    print '#%d: %s' % (idx + 1, animal)
# Prints "#1: cat", "#2: dog", "#3: monkey", each on its own line
```

- Dictionaries

```
d = {'person': 2, 'cat': 4, 'spider': 8}
for animal in d:
    legs = d[animal]
    print 'A %s has %d legs' % (animal, legs)
# Prints "A person has 2 legs", "A spider has 8 legs", "A cat has 4 legs"
```

Iterando sobre los tipos de datos

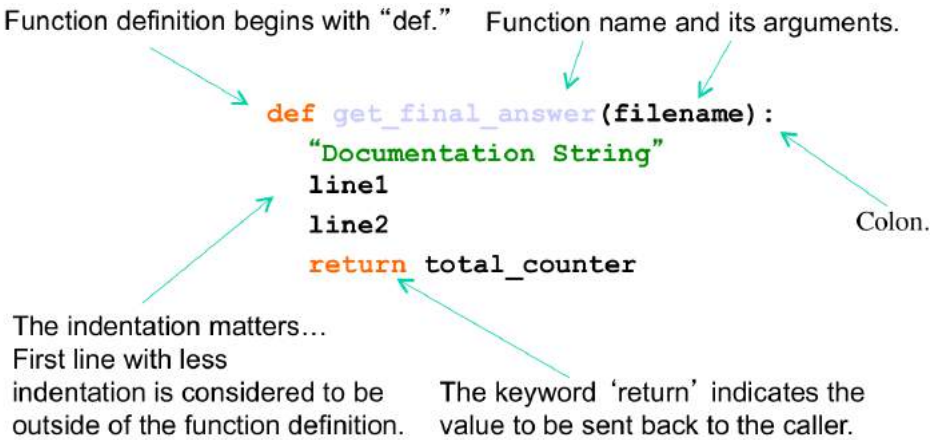
List comprehensions:

- Generate a new list by applying a function to every member of an original list.
- Extensively used in Python
- Sintaxis

```
[ <expression> for <name> in <list> if <filter> ]
```

```
nums = [0, 1, 2, 3, 4]
squares = [x ** 2 for x in nums]
print squares   # Prints [0, 1, 4, 9, 16]
```

```
nums = [0, 1, 2, 3, 4]
even_squares = [x ** 2 for x in nums if x % 2 == 0]
print even_squares  # Prints "[0, 4, 16]"
```


Function definition begins with "def."
Function name and its arguments.
def get_final_answer(filename):
"Documentation String"
line1
line2
return total_counter
Colon.
The indentation matters…
First line with less
indentation is considered to be
outside of the function definition.
The keyword 'return' indicates the
value to be sent back to the caller.

- The syntax for a function call is:

```
>>> def myfun(x, y):
        return x * y
>>> myfun(3, 4)
12
```

- There is no *function overloading* in Python.
- Functions can be used as any other data type. They can be
  - Arguments to function, return values of functions, assigned to variables, parts of tuples, lists, etc
- Parameters in Python are *Call by Assignment*.

  Sometimes acts like “call by reference” and sometimes “like call by value”
  - Mutable datatypes (list, set, dict,...) : Call by reference.
  - Immutable datatypes (int, str, float, ...) : Call by value.

if Statements

```
if x == 3:
    print "X equals 3."
elif x == 2:
    print "X equals 2."
else:
    print "X equals something else."
print "This is outside the 'if'."
```

Note:

- Use of indentation for blocks
- Colon (:) after boolean expression

while sentence

```
x = 3
while x < 10:
    x = x + 1
    print "Still in the loop."
print "Outside of the loop."
```

- You can use the keyword `break` inside a loop to leave the while loop entirely.
- You can use the keyword `continue` inside a loop to stop processing the current iteration of the loop and to immediately go on to the next one.

Python is “all objects”

- A class is a special data type that defines how to build a certain kind of object.
  - The class also stores some data items that are shared by all the instances of this class.
  - Instances are objects that are created which follow the definition given inside of the class.
- Define a method in a class by including function definitions within the scope of the class block.
  - There must be a special first argument `self` in all method definitions which gets bound to the calling instance
  - There is usually a special method called `__init__` in most classes

Example class creation and usage code

```
class Greeter(object):

    # Constructor
    def __init__(self, name):
        self.name = name  # Create an instance variable

    # Instance method
    def greet(self, loud=False):
        if loud:
            print 'HELLO, %s!' % self.name.upper()
        else:
            print 'Hello, %s' % self.name

g = Greeter('Fred')  # Construct an instance of the Greeter class
g.greet()            # Call an instance method; prints "Hello, Fred"
g.greet(loud=True)   # Call an instance method; prints "HELLO, FRED!"
```

- Use classes & functions defined in another file.
- A Python module is a file with the same name (plus the .py extension)
- Three formats of the command:

```
import <somefile>  [ as <str> ]
from <somefile> import *
from <somefile> import <name>
```

- What's the difference?

  What gets imported from the file and what name refers to it after it has been imported.

- `import <somefile>`

  Everything in somefile.py gets imported. To refer to something in the file, append the text “somefile.” to the front of its name:

  ```
  somefile.className.method(“abc”)
  somefile.myFunction(34)
  ```

- `import <somefile> as <str>`

  Everything in somefile.py gets imported. To refer to something in the file, append the text “str.” to the front of its name:

  ```
  str.className.method(“abc”)

  str.myFunction(34)
  ```

Modules

- `import <somefile> [ as <str> ]`
- `from <somefile> import *`

  Everything in somefile.py gets imported. To refer to anything in the module, just use its name.

  Everything in the module is now in the current namespace.

  Caveat! Using this import command can easily overwrite the definition of an existing function or variable!

```
className.method("abc")
myFunction(34)
```

- `import <somefile>`
- `from <somefile> import *`
- `from <somefile> import <className>`

  Only the item className in somefile.py gets imported. After importing className, you can just use it without a module prefix. It's brought into the current namespace.

  Caveat! This will overwrite the definition of this particular name if it is already defined in the current namespace!

Numpy is the core library for scientific computing in Python.
It provides:

- a high-performance multidimensional array object,
- tools for working with these arrays.

A numpy array is a grid of values, all of the same type

- It is **indexed by a tuple** .
- The number of dimensions is the **rank** of the array;
- the **shape** of an array is a tuple of integers giving the size of the array along each dimension

We can initialize numpy arrays from nested Python lists, and access elements using square brackets.

```
import numpy as np

a = np.array([1, 2, 3])  # Create a rank 1 array
print type(a)            # Prints "<type 'numpy.ndarray'>"
print a.shape            # Prints "(3,)"
print a[0], a[1], a[2]   # Prints "1 2 3"
a[0] = 5                 # Change an element of the array
print a                  # Prints "[5, 2, 3]"

b = np.array([[1,2,3],[4,5,6]])   # Create a rank 2 array
print b.shape                     # Prints "(2, 3)"
print b[0, 0], b[0, 1], b[1, 0]   # Prints "1 2 4"
```

Array indexing: **Slicing**

```
import numpy as np

# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Use slicing to pull out the subarray consisting of the first 2 rows
# and columns 1 and 2; b is the following array of shape (2, 2):
# [[2 3]
#  [6 7]]
b = a[:2, 1:3]

# A slice of an array is a view into the same data, so modifying it
# will modify the original array.
print a[0, 1]   # Prints "2"
b[0, 0] = 77    # b[0, 0] is the same piece of data as a[0, 1]
print a[0, 1]   # Prints "77"
```

Array indexing: **Slicing**

```
import numpy as np

a = np.array([[1,2], [3, 4], [5, 6]])

# An example of integer array indexing.
# The returned array will have shape (3,) and
print a[[0, 1, 2], [0, 1, 0]]  # Prints "[1 4 5]"

# The above example of integer array indexing is equivalent to this:
print np.array([a[0, 0], a[1, 1], a[2, 0]])  # Prints "[1 4 5]"

# When using integer array indexing, you can reuse the same
# element from the source array:
print a[[0, 0], [1, 1]]  # Prints "[2 2]"

# Equivalent to the previous integer array indexing example
print np.array([a[0, 1], a[0, 1]])  # Prints "[2 2]"
```

Array indexing: **Boolean indexing**

```
import numpy as np

a = np.array([[1,2], [3, 4], [5, 6]])

bool_idx = (a > 2)  # Find the elements of a that are bigger than 2;
                    # this returns a numpy array of Booleans of the same
                    # shape as a, where each slot of bool_idx tells
                    # whether that element of a is > 2.

print bool_idx      # Prints "[[False False]
                    #          [ True  True]
                    #          [ True  True]]"

# We use boolean array indexing to construct a rank 1 array
# consisting of the elements of a corresponding to the True values
# of bool_idx
print a[bool_idx]  # Prints "[3 4 5 6]"

# We can do all of the above in a single concise statement:
print a[a > 2]     # Prints "[3 4 5 6]"
```

Array data types

- Every numpy array is a grid of elements of the same type.
- Numpy provides a large set of numeric datatypes that you can use to construct arrays.

```
import numpy as np

x = np.array([1, 2])  # Let numpy choose the datatype
print x.dtype         # Prints "int64"

x = np.array([1.0, 2.0])  # Let numpy choose the datatype
print x.dtype             # Prints "float64"

x = np.array([1, 2], dtype=np.int64)  # Force a particular datatype
print x.dtype                         # Prints "int64"
```

Array maths

- Basic mathematical functions operate elementwise on arrays, and are available both as operator overloads and as functions.

```
import numpy as np

x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

# Elementwise sum; both produce the array
# [[ 6.0  8.0]
#  [10.0 12.0]]
print x + y
print np.add(x, y)

# Elementwise difference; both produce the array
# [[-4.0 -4.0]
#  [-4.0 -4.0]]
print x - y
print np.subtract(x, y)

# Elementwise product; both produce the array
# [[ 5.0 12.0]
#  [21.0 32.0]]
print x * y
print np.multiply(x, y)
```

Array maths

- Algebraic array multiplications are performed with the `dot` function.

```
import numpy as np

x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])

v = np.array([9,10])
w = np.array([11, 12])

# Inner product of vectors; both produce 219
print v.dot(w)
print np.dot(v, w)

# Matrix / vector product; both produce the rank 1 array [29 67]
print x.dot(v)
print np.dot(x, v)

# Matrix / matrix product; both produce the rank 2 array
# [[19 22]
#  [43 50]]
print x.dot(y)
print np.dot(x, y)
```

Broadcasting

- Allows numpy to work with arrays of different shapes when performing arithmetic operations.
- Provides a means of vectorizing array operations. It does this without making needless copies of data and usually leads to efficient algorithm implementations.

Approach:

1. NumPy compares their shapes element-wise. It starts with the trailing dimensions, and works its way forward.
   Two dimensions are compatible when
   - they are equal, or
   - one of them is 1
2. The size of the resulting array is the maximum size along each dimension of the input arrays.

Broadcasting

```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = x + v  # Add v to each row of x using broadcasting
print y  # Prints "[[ 2  2  4]
         #          [ 5  5  7]
         #          [ 8  8 10]
         #          [11 11 13]]"
```

Broadcasting

```
import numpy as np

# Compute outer product of vectors
v = np.array([1,2,3])  # v has shape (3,)
w = np.array([4,5])    # w has shape (2,)
x = np.array([[1,2,3], [4,5,6]])
```

```
# Add a vector to each column of a matrix
# x has shape (2, 3) and w has shape (2,).
# If we transpose x then it has shape (3, 2) and can be broadcast
# against w to yield a result of shape (3, 2); transposing this result
# yields the final result of shape (2, 3) which is the matrix x with
# the vector w added to each column. Gives the following matrix:
# [[ 5  6  7]
#  [ 9 10 11]]
print (x.T + w).T
```