Introducción a la programación de computadores

mediante el lenguaje Python

Parte I

Á. Ibeas

Buena parte de estas notas están inspiradas en los materiales siguientes:

- J. Campbell, P. Gries, J. Montojo y G. Wilson: **«Practical Programming»**. Pragmatic Bookshelf, 2009.
- `http://docs.python.org`
- Andrés Marzal e Isabel Gracia: **«Introducción a la programación con Python»**. Universitat Jaume I.

Un ordenador es una máquina que permite almacenar información y realizar cálculos de manera automática. Utilizando los recursos que pone a nuestra disposición un procesador, podemos **programar** el ordenador para que lleve a cabo la tarea que nos interesa.

Por ejemplo, si disponemos de las horas de entrada y de salida de miles de vehículos que han atravesado un túnel, resulta muy sencillo encargar a una máquina que calcule parámetros estadísticos relativos a su velocidad.

Si tenemos cientos de fotografías en una tarjeta de memoria, la tarea de comprimirlas a un formato más reducido es un cálculo que podemos automatizar mediante un programa.

La tarea del programador consiste en combinar las operaciones elementales que puede encargar al procesador de manera que obtenga el resultado que necesita.

La labor del programador sería poco eficiente si su programa solo fuera válido para un procesador concreto.

Los **lenguajes de programación de alto nivel** permiten desarrollar programas con una serie de instrucciones básicas, sin tener en cuenta el procesador en que se ejecutarán.

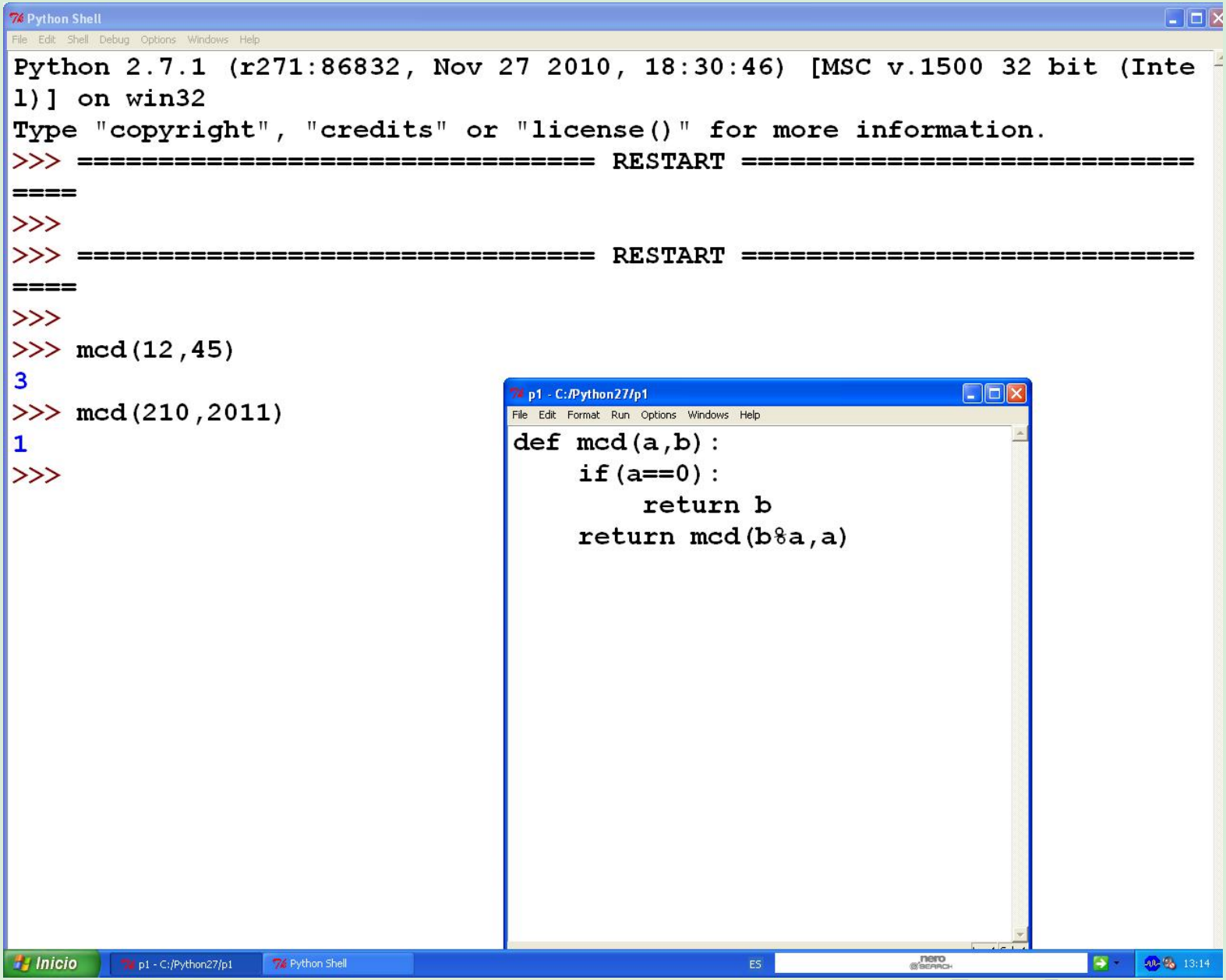
Una vez listo el programa, mediante un proceso de *compilación* o *interpretación* el programa podrá ejecutarse en una máquina concreta.

Python (/ˈpaɪθən/, /ˈpaɪθɑːn/) es un **lenguaje interpretado**: para ejecutar un programa escrito en este lenguaje, necesitamos un **intérprete** adecuado para la máquina en que lo queremos ejecutar. Este intérprete recorre el código Python traduciendo sus instrucciones a operaciones del procesador.

http://www.python.org/download

Un programa suele tener la forma de un fichero de texto donde se escriben secuencialmente las instrucciones de que consta. Para escribir un programa, necesitamos papel y lápiz o un editor de textos.

Integrated DeveLopment Environment



Entorno interactivo

Si queremos utilizar el intérprete de Python para realizar algunas operaciones que no vamos (en principio) a tener que repetir sistemáticamente, es más sencillo utilizar un intérprete interactivo del lenguaje que escribir cada instrucción en un archivo de texto y ejecutarla.

Para familiarizarnos con el intérprete interactivo de Python, podemos usarlo como una calculadora.

Operadores aritméticos

```
>>> 3+4
7
>>> 45/7
6
>>> 45.0/7
6.4285714285714288
```

¡Atención!

En Python 2, el operador / es distinto según el tipo de sus operandos. Si ambos son enteros, devuelve el **cociente entero** (sin decimales). Contamos con un operador para el **resto**:

```
>>> 45%7
3
>>> 6*7+3
45
```

Tipos de datos

Como hemos visto, cada dato que maneja Python tiene un **tipo** (no es lo mismo 45, cuyo tipo es «número entero», que 45.0). Hay otros tipos de datos además de los números. Tenemos, por ejemplo, las cadenas de texto:

```
>>> "Hola, "+"Pepito"+"."
'Hola, Pepito.'
```

y las listas:

```
>>> ["subir","bajar","parar"][0]
'subir'
>>> ["subir","bajar","parar"][1]
'bajar'
>>> ["subir","bajar","parar"][2][2]
'r'
```

Tipos de datos

Podemos saber cuál es el tipo de un dato mediante la instrucción:

```
>>> type(777777777)
<type 'int'>
>>> type(7777777777)
<type 'long'>
>>> type(7.7)
<type 'float'>
>>> type('7')
<type 'str'>
>>> type([7,"7"])
<type 'list'>
```

Un programa sencillo

En lugar de ir ejecutando instrucciones en el intérprete interactivo, vamos a escribir un fichero de texto como este:

programa.py

```
#!/usr/bin/python

print "¡Aquí estoy!"
```

No nos preocupemos de momento por la primera línea. Con respecto a la segunda... resulta difícil encontrar algún manual para aprender un lenguaje de programación cuyo primer ejemplo no sea «¡Hola, mundo!».

print expresión (, expresión)*

Esta instrucción de Python 2 evalúa las expresiones que involucra según las reglas del intérprete y las imprime por la salida estándar.

```
>>>print 2**0*True+2**2**True+2**5,"Pico","Dobra"
37 Pico Dobra
```

Se suele combinar con el operador % aplicado a cadenas de caracteres:

```
>>> print "Con %s cañones matamos %f moscas" % (10,0)
Con 10 cañones matamos 0.000000 moscas
```

input, raw_input

La instrucción `input` puede considerarse contraria a `print`: permite la comunicación entre programa y usuario en el sentido inverso.

input`([mensaje])`
raw_input`([mensaje])`

La segunda de estas instrucciones devuelve una cadena de caracteres que el usuario ha tecleado. La primera, además, interpreta esa cadena como una expresión de Pyhton.

```
>>> nombre=raw_input("Nombre: ")
Nombre: Emilio
>>> n=input("Edad: ")
Edad: 21
>>> print nombre,"tiene",n,"años."
Emilio tiene 21 años.
```

Tipo boolean

- Dos valores: «verdadero» y «falso».

```
>>> 4==5
False
>>> 4<5
True
```

- Python utiliza los enteros 0 y 1 para representar estos valores, dando lugar a situaciones como esta:

```
>>> True+True
2
>>> type(True*False)
<type 'int'>
>>> type(True and False)
<type 'bool'>
```

Operadores lógicos

	and (&)	or (|)
(V,V)	V	V[1]
(V,F)	F	V
(F,V)	F	V
(F,F)	F	F

```
>>> not(False) or True
True
>>> (True | False) & (not(False and True))
True
```

[1] El valor lógico de «o» en la frase «o estás con ellos, o con nosotros» es distinto: es **exclusivo**. Pensamos sin embargo en el sentido de «puedo encender la lumbre si me traes teas o piñas».

Operadores relacionales

```
>>> 4.1>4.09
True
>>> 4<=2*2
True
>>> "Albarracín"<"Zambia"
True
>>> 4!=4
False
>>> 4<>4
False
```

Los dos últimos son sinónimos, pero es preferible no usar el último, que Pyhton 3 ya no interpreta.

Números complejos

Python permite utilizar los operadores de comparación:

```
==   !=   <   <=   >   >=
```

para casi todos los tipos de datos, e incluso entre datos de tipos distintos (devolviendo un resultado poco significativo, en general). Sin embargo, teniendo en cuenta la ausencia de un orden «natural» en el cuerpo de los números complejos, solo permite utilizar las comparaciones == y != para estos datos.

```
>>> complex(1,-2)
(1-2j)
>>> complex(0,1)**2==-1
True
>>> complex(1,0)<complex(0,1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: no ordering relation is defined for complex numbe
```

Variables

Resulta útil utilizar «nombres» para algunos datos que pretendemos utilizar varias veces. Por ejemplo, si queremos calcular el largo de un pedazo de papel que guarde las mismas proporciones que los de la serie A de la norma ISO/DIN:

```
>>> import math
>>> math.sqrt(2)
1.4142135623730951
>>> razon=math.sqrt(2)
>>> 21*razon
29.698484809834998
>>> 29.7*razon
42.002142802480925
```

Variables

Como indica el término que estamos utilizando, el valor de una variable no tiene por qué permancer fijo. Normalmente necesitaremos que cambie.

```
>>> 2*variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'variable' is not defined
>>> variable=5
>>> 2*variable
10
>>> variable="Cristina"
>>> 2*variable
'CristinaCristina'
```

Asignación y comparación

Es importante distinguir entre los operadores = y ==, que ya hemos visto:

- = sirve para asignar un valor a una variable. No es simétrico: a la izquierda se pone el nombre de la variable y a derecha, el valor que toma.
- == compara dos datos y devuelve un valor «bool» (True o False).

```
>>> v=2
>>> v==3
False
>>> v=3
>>> 3==v
True
```

Asignación

No está de más insistir en que el operador asignación = es un operador, y no enuncia la igualdad entre dos términos (lo que es su significado estándar en notación matemática). Otros lenguajes de programación codifican este operador mediante el símbolo :=, que también es habitual en Matemáticas.

Por tanto, la expresión de Python:

```
x=2*x
```

no es una ecuación con la solución única `x=0`. Se trata de una instrucción que asigna a la variable `x` el doble del valor que almacenaba anteriormente.

Variables

No podemos utilizar cualquier cadena de caracteres como **identificador** para denominar una variable. Por ejemplo, parece razonable que Pyhton no permita llamar a una variable `g=h`, ya que si quisiéramos asignarle un valor: `g=h=2`, el interpréte no podría decidir entre crear dos variables (`g` y `h`) con el valor 2 o una sola.

En general, podemos utilizar letras, números y el signo `_`, con las dos restricciones siguientes:

- El primer carácter del identificador no puede ser un número.
- No pueden utilizarse algunas palabras que reserva el intérprete (por ejemplo: **for, break, and**...)

Un programa sencillo

programa.py

```
#!/usr/bin/python

nombre=raw_input("¿Cómo se llama Ud.?\n")
print("Bienvenido a la aplicación, "+nombre+".\n")
```

Operadores aritméticos de asignación

Los lenguajes de programación suelen disponer de símbolos para representar operaciones artiméticas que no son comunes al escribir matemáticas.
En Pyhton, la expresión `a(operador)=b` es, en general, equivalente a `a=a(operador)b`.

```
>>> a=3;a+=4;a
7
>>> a%=5;a
2
>>> b="Cadena ";c="expansible";b+=c;b
'Cadena expansible'
```

Sintaxis

Pyhton, como cualquier otro lenguaje de progrmación, especifica una **sintaxis** para su código. Como en la mayoría de los lenguajes,

Por lo general, cada instrucción se escribe en una línea:

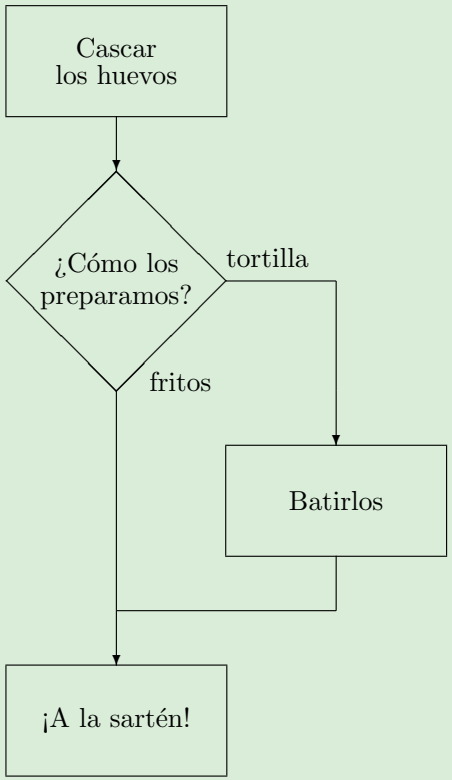
```
>>> texto="torrelavega"
>>> texto.capitalize()
'Torrelavega'
```

También pueden incluirse varias instrucciones en una misma línea, utilizando el separador ;:

```
>>> altura=25; cota=320; nieva=(altura>=cota); nieva
False
```

La programación resultaría demasiado rígida si tuviéramos que restringirnos a enumerar una sucesión de instrucciones que se ejecutaran una detrás de otra. A veces, queremos «seguir caminos distintos» en función de alguna circunstancia:

Diagrama de flujo



Estructuras de control

Para modificar la ejecución lineal de las instrucciones, los lenguajes de programación cuentan con instrucciones denominadas estructuras de control. El ejemplo anterior involucra una bifurcación que se recorre o no en función de una condición. Un fragmento del código Python correspondiente podría ser:

```
if(tortilla): print("Batir los huevos")
print("¡A la sartén!")
```

También podemos prescribir alguna instrucción para el caso en que la condición no se satisfaga:

```
if(tortilla): print("Batir los huevos")
else: print("Bien de aceite")
print("¡A la sartén!")
```

Bloques

En los ejemplos anteriores, cada «bifurcación» comprende una única instrucción. Para incluir más, podemos optar entre separarlas con punto y coma o por agruparlas dentro de un «bloque»:

```
if(tortilla):
    print("Batir los huevos")
    print("Mezclar con ingrediente al gusto")
else:
    print("Bien de aceite");
print("¡A la sartén!")
```

Sintaxis

En muchos lenguajes de programación, la norma consiste en encerrar las instrucciones a agrupar mediante llaves o paréntesis. Además, para facilitar la lectura del programa (por parte del hombre, que no de la máquina), suele recurrirse a sangrar los bloques.

Una característica de Python es que hace de esa costumbre (el sangrado —«indentation» en ingles—) la norma y la pertenencia de una línea a un bloque se determina según su margen: no existen delimitadores para los bloques.

La sentencia condicional `if` admite una estructura más compleja que la dual `if/else`.

```
if condición: bloque
( elif condición: bloque)*
[ else: bloque]
```

Pueden incluirse tantas sentencias `elif` como ne necesite: se ejecuta únicamente el bloque correspondiente a la primera condición que se verifica, o el correspondiente a la instrucción `else` (en caso de estar presente), si no se verifica ninguna.

```
#Criterios de divisibilidad

def divisible(m,n):
    if n==2:
        c=cifra_menos_significativa(m)
        return c%2==0
    elif n==3:
        s=suma_de_las_cifras(m)
        return divisible(m,3)
    elif n==5:
        c=cifra_menos_significativa(m)
        return c==0 or c==5
    elif n==7:
        ...
    else:
        return m%n==0
```

range(`[comienzo]`,`parada`,`[paso]`)

Se utiliza típicamente en los bucles `for`.

- *range(a,b)* devuelve la lista $[a, a+1, a+2, \ldots, b-1]$.
- *range(a,b,Δ)* devuelve una lista que comienza por a, incrementando cada elemento en Δ, hasta llegar a b (sin incluirlo).

```
>>> range(-7,3)
[-7, -6, -5, -4, -3, -2, -1, 0, 1, 2]
>>> range(4,71,9)
[4, 13, 22, 31, 40, 49, 58, 67]
```

for `variable` **in** `lista: bloque`

```
>>> for x in range(1,11):
...     print("9x%d=%d"%(x,9*x))
...
9x1=9
9x2=18
9x3=27
9x4=36
9x5=45
9x6=54
9x7=63
9x8=72
9x9=81
9x10=90
```

while

while `condición: bloque`

El bloque se ejecuta repetidamente mientras se cumpla la condición.

```
semana=["lunes","martes","miércoles","jueves","viernes","sá
i=1; d=1
while(i<=31):
    print(semana[d]+" "+i)
    i=i+1
    d=(d+1)%7
```

Funciones

Son **llamadas a funciones** las instrucciones mostradas a continuación, que ya hemos utilizado:

```
>>> type(777777777)
<type 'int'>
>>>print('¡Aquí estoy!')
¡Aquí estoy!
>>> complex(1,-2)
(1-2j)
>>> math.sqrt(2)
1.4142135623730951
>>>nombre=raw_input("¿Cómo se llama Ud.?\n")
```

Una función se invoca escribiendo su nombre o identificador seguido de los **argumentos** de la función (que pueden ser varios, uno o ninguno) entre **paréntesis**.

Los ejemplos anteriores son funciones «predefinidas» en el lenguaje Python.

Modularidad

Un principio útil para diseñar un programa consiste en dividir la tarea que se quiere alcanzar en procesos más sencillos e independientes que, combinados, permiten resolver el problema.

Es interesante poder definir nuevas funciones para construir a partir de ellas un programa complejo.

Definición de funciones

```
>>>  def suma_de_cifras(n):
...     aux=str(n)
...     suma=0
...     for i in range(0,len(aux)):
...         suma+=int(aux[i])
...      return suma
...
>>> for x in range(0,94,16):
...      suma_de_cifras(x)
...
0
7
5
12
10
8
```

Definición de funciones

Si durante la ejecución de una función se alcanza la instrucción `return`, esta ejecución se termina y el intérprete utiliza el valor indicado.

```
>>> def f(x):
...     if(x>5):
...         return
...     return([])
...
>>> type(f(5))
<type 'list'>
>>> type(f(6))
<type 'NoneType'>
```

Recursión

¡Una función puede llamarse —recurrir— a sí misma!

factorial.py

```
def factorial(n):
    return factorial(n)
```

En el ejemplo anterior, una llamada a la función `factorial` desencadenaría un bucle formalmente infinito incapaz de devolver ninguna respuesta. Sin embargo...

factorial.py

```
def factorial(n):
    if n==0:
        return 1
    return n*factorial(n-1)
```

Módulos

En Python, la palabra «módulo» se utiliza para denominar a un conjunto de funciones (u otros fragmentos de código). Generalmente, un módulo proporciona utilidades que se pueden aprovechar en varios contextos.

El programador que utiliza el módulo no necesita estar al tanto de los detalles de implementación del código que contiene: le basta conocer con precisión **qué hacen** las herramientas del módulo (su especificación).

Cuando personas distintas participan en la escritura de un programa, es imprescindible complementar el código con comentarios que faciliten su lectura (la **documentación**). En general, es una práctica aconsejable: un código sin ningún texto aclaratorio puede ser muy difícil de comprender incluso (al cabo de poco tiempo) para quién lo programó.

Módulos

Las funciones matemáticas predefinidas en Pyhon pueden quedarse cortas para aplicaciones sencillas. Resultaría laborioso y muy poco eficiente que cada usuario tuviera que programar algorítmos para el cálculo de raíces cuadradas, logaritmos,... en caso de necesitarlos. Ya hemos hecho uso del módulo `math`.

```
>>> from math import cos,sin
>>> cos(0);sin(0)
1.0
0.0
>>> sin(pi)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pi' is not defined
>>> from math import pi
>>> cos(pi)
-1.0
```

Módulos

Podemos «importar» todo el contenido del módulo mediante la instrucción:

```
from math import *
```

Haciendo esto, puede presentarse algún problema, como que estemos sobreescribiendo sin darnos cuenta una función que habíamos definido. Contamos con una alternativa, con la que tenemos que especificar el módulo que contiene los «recursos» que queremos utilizar:

```
>>> import math
>>> cos(0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'cos' is not defined
>>> math.cos(math.pi)
-1.0
```