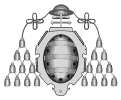

Universidad de Oviedo
Grado en Física

Introducción a la Física Computacional

Rosario Díaz Crespo, Julio Manuel Fernández Díaz

Curso 2015-16

Python como lenguaje algorítmico: Tercera parte

Práctica 6

Python como lenguaje algorítmico: Tercera parte

6.1. Objetivo

Familiarizarse con el uso de funciones y módulos en Python.

6.2. Qué es una función en Python

Una función es un fragmento de código con un nombre asociado que realiza una serie de tareas y devuelve un valor. A los fragmentos de código que tienen un nombre asociado y no devuelven valores se les suele llamar "procedimientos". En Python no existen los procedimientos, ya que cuando el programador no especifica un valor de retorno la función devuelve el valor `None` (nada).

El uso de funciones permite racionalizar el código: como función, programada de manera general, una tarea repetitiva dependiente de algunas variables (denominados "argumentos" de la función), se puede usar para cualquier problema del mismo tipo.

Por ejemplo, podemos diseñar una función que resuelva una ecuación de segundo grado a partir de sus parámetros a, b y c, devolviéndonos como resultado las raíces (reales o complejas). Esta función general nos permite resolver **cualquier** ecuación de segundo grado.

Como veremos más adelante, Python permite agrupar bajo un mismo "paraguas" diversas funciones relacionadas entre sí, en lo que se denominan **módulos**.

6.3. Funciones de conversión de tipos

Hemos visto que los números de diferentes tipos se convierten automáticamente a un tipo común después de la operación aritmética o una comparación. También se pueden cambiar los tipos con las funciones:

- `int(a)`: convierte `a` en un entero.
- `long(a)`: convierte `a` en un entero largo.
- `float(a)`: convierte `a` en un `float`.
- `complex(a)`: convierte `a` en un complejo `a+0j`.
- `complex(a,b)`: convierte `a` en un complejo `a+bj`.

Las funciones anteriores sólo trabajan convirtiendo cadenas en números tan largos como las cadenas de manera que representen un número valido. La conversión de `float` a entero se realiza con un proceso de truncamiento, no por redondeo.

Ejemplo 6.3.1. *Uso de las funciones de conversión de tipos.*

En el modo interactivo de Python escriba:

```
>>> a = 5
>>> b = -3.6
>>> d = '4'
>>> print(int(b))
-3
>>> print(float(d))
4.0
>>> print(complex(a,b))
(5-3.6j)
>>> print(int(d))
4
>>> print(float(d))
4.0
```

Ejercicio 6.3.1. Utilice las funciones de conversión de tipos para convertir los grados Celsius en Fahrenheit mediante la expresión $F = 9/5C + 32$.

6.4. Funciones matemáticas

El núcleo de Python sólo contiene las siguientes funciones matemáticas:

- `abs(a)`: valor absoluto de `a`.
- `max(secuencia)`: el mayor elemento de una secuencia (tupla, lista o cadena de caracteres).
- `min(secuencia)`: el menor elemento de una secuencia.
- `round(a, n)`: redondea `a` con `n` decimales.
- `cmp(a, b)`: devuelve: `-1` si $a < b$, `0` si $a = b$, `1` si $a > b$

La mayoría de las funciones matemáticas se encuentran en el módulo `math` que estudiaremos más adelante.

6.5. Funciones definidas por el usuario

Es posible crear nuevas funciones para resolver problemas concretos. En Python la definición (o declaración) de una función se realiza de la manera siguiente:

```
def nombre_funcion (param1, param2, ...):
   sentencia 1
   sentencia 2
   ...
   return valor_devuelto
```

donde `param1`, `param2`, ..., son los **parámetros** (también denominados **argumentos formales**). Obsérvese también el **indentado** uniforme.

`valor_devuelto`, aparte de un valor numérico, cadena, etc., también puede ser una tupla, lista, etc. Cuando es una tupla no hace falta poner los paréntesis.

También podemos encontrarnos con una cadena de texto como primera línea del cuerpo de la función. Estas cadenas se conocen con el nombre de *docstring* ("cadena de documentación") y, como su nombre indica, sirven de documentación de la función:

```
def iden (p1, p2):
   '''
   Esta funcion imprime los dos valores pasados como argumentos
   '''
   print(p1)
   print(p2)
```

Al **definir** una función lo único que hacemos es asociar un nombre al fragmento de código que conforma la función, de forma que podamos ejecutar dicho código más tarde referenciándolo por su nombre. Es decir, a la hora de escribir estas líneas no se ejecuta la función. Para **llamar** (o invocar) a la función (ejecutar su código) se escribiría el nombre de la función a la que queremos llamar seguido de los valores que queramos pasar como argumentos entre paréntesis.

En Python un parámetro puede ser cualquier objeto, incluida una función. Se pueden definir funciones que no tengan parámetros y/o que tampoco devuelvan valores.

Ejemplo 6.5.1. *Uso de funciones definidas por el usuario que no tienen argumentos y que no devuelven ningún valor.*

1. *Escriba en un fichero denominado* `lineapuntos.py` *el siguiente código:*

```
def linea():
    print('__________________')
def puntos():
    print('.....................')
print('Invocamos las funciones linea y puntos')
linea()
puntos()
print('Ahora voy a llamar a la función linea dos veces')
linea()
linea()
print('Ahora voy a llamar a la función puntos tres veces')
puntos()
puntos()
puntos()
```

2. *Guarde el fichero, ejecute el programa y observe la respuesta.*

Ejemplo 6.5.2. *Uso de funciones definidas por el usuario con argumentos que no devuelven ningún valor.*

En el modo interactivo de Python escriba el siguiente código:

```
>>> def iden (p1, p2):
...       print(p1)
...       print(p2)
...
>>> nombre = 'Sara'
```

```
>>> apellido = 'Silva'

>>> iden(nombre, apellido)

Sara

Silva

>>> iden('Rebeca', 'Alvarez')

Rebeca

Alvarez
```

Los argumentos que se pasan cuando se invoca la función se denominan **argumentos actuales** (o **argumentos** a secas). El número de valores que se pasan como argumentos actuales al llamar a la función tiene que coincidir con el número de argumentos formales que la función acepta según la declaración de la función. En caso contrario Python dará un error.

También se pueden usar argumentos con **valores por defecto**, que se indican en la definición de la función.

Ejemplo 6.5.3. *Paso incorrecto de argumentos.*

Siguiendo el ejemplo anterior, escriba:

```
>>> iden('Sara')
```

La respuesta será:

```
Traceback (most recent call last):

File ''<stdin>'', line 1, in <module>

TypeError: iden() takes exactly 2 arguments (1 given)
```

Los argumentos de invocación se asocian a los parámetros de la definición por **posición**.

Ejemplo 6.5.4. *Asociación de parámetros por posición:*

1. *Escriba un fichero que contenga el siguiente código:*

   ```
   def posicion(x0,v0,a,t):
       print float(x0+v0*t+0.5*a*t**2)
   posicion(2,5,2,1)
   ```

   La función interpreta que `x0 = 2`, `v0 = 5`, `a = 2`, `t = 1`.

2. *Ejecute el programa y observe la respuesta.*

3. *Modifique la función para que imprima el valor de la posición con formato de coma flotante con dos decimales, y el valor de cada uno de los parámetros.*

Ejemplo 6.5.5. *Los valores por defecto para los argumentos se definen situando un signo igual después del nombre del argumento y a continuación el valor por defecto. Conviene no poner ningún espacio en este caso entre el nombre del argumento, el signo* `=`*, y su valor por defecto:*

```
>>> def imprimir (texto, veces=1):
...       print(veces * texto)
...
>>> imprimir(''hola'')
hola
```

Se ha tomado el valor por defecto de la variable `veces`*.*

```
>>> imprimir(''hola'', 2)
holahola
```

Al invocar la función `imprimir` *se ha asignado el valor* `2` *a la variable* `veces`*.*

Es posible definir funciones con un número variable de argumentos. Para ello se usa un parámetro final cuyo nombre debe ir precedido del carácter `*`:

Ejemplo 6.5.6. *Definición de una función con un número variable de argumentos.*

1. *En el modo interactivo de Python escriba el siguiente código:*

   ```
   >>> def varios (par1, par2, *otros):
   ...       for valores in otros:
   ...           print(valores)
   ```

   Recuerde que el identado en la función y en el bucle es necesario.

   Este método funciona creando una tupla (de nombre `otros` *en el ejemplo) en la que se almacenan los valores de todos los parámetros extra pasados como argumentos.*

2. *Compruebe el funcionamiento de la función con un número variable de parámetros escribiendo:*

   ```
   >>> varios(1, 2)
   ```

   En este caso no imprime nada pues la tupla `otros` *está vacía.*

   ```
   >>> varios(1, 2, 3)
   3
   >>> varios(1, 2, 3, 4)
   3
   4
   >>> varios('a', 'b', 'c', 'd')
   c
   d
   ```

También se puede preceder el nombre del último argumento con `**`, en cuyo caso en lugar de una tupla se utilizaría un diccionario. Las claves de este diccionario serían los nombres de los argumentos indicados al llamar a la función y los valores del diccionario, los valores asociados a estos argumentos.

Ejemplo 6.5.7. *En el siguiente ejemplo se utiliza la función* `items` *de los diccionarios, que devuelve una lista con sus elementos, para imprimir los argumentos que contiene el diccionario.*

1. *Escriba en el modo interactivo de Python el siguiente código:*

```
>>> def varios (par1, par2, **mas):
...     for i in mas.items():
...         print i
...
>>> varios(1, 2, tercero=3)
```

2. *Observe la respuesta.*

No todos los cambios que hagamos a los parámetros dentro de una función Python se reflejarán fuera de esta. En Python existen objetos inmutables, como las tuplas, las cadenas y los números, por lo que si intentáramos modificar una tupla pasada como argumento lo que ocurriría en realidad es que se crearía una nueva instancia, por lo que los cambios no se verían fuera de la función (aunque sí dentro de ella).

Ejemplo 6.5.8. *En este ejemplo se hace uso de la función* `append` *de las listas.*

1. *Escriba el siguiente código en el modo interactivo de Python.*

```
>>> def f(x, y):
...         x = x + 3
...         y.append(23)
...         print('dentro de f', x, y)
...
>>> x = 22
>>> y = [22]
```

2. *Observe el resultado al ejecutar:*

```
>>> f(x, y)
>>> print('fuera de f', x, y)
```

La variable `x` *no conserva los cambios una vez se sale de la función porque los enteros son inmutables en Python. Sin embargo la variable* `y` *si los conserva, porque las listas son mutables.*

Para que una función devuelva un valor debemos usar la sentencia return. En el modo interactivo de Python escriba el siguiente código:

```
>>> def sumar (x,y):
...       return x+y
...
>>> print(sumar(2, 3))
5
```

Esta función suma los valores pasados como argumentos y devuelve el resultado como valor de retorno.

Ejemplo 6.5.9. *Funciones que devuelven un valor.*

1. *Escriba en un fichero el siguiente código:*

```
def y_max(v0):
    '''
    Esta función calcula la altura máxima alcanzada en un
    lanzamiento vertical hacia arriba
    '''
    return v0**2/(2*9.81)

velocidad = input('velocidad inicial? ')
print('Para una velocidad inicial de \t%5.2fm/s' % velocidad)
print('la altura máxima alcanzada es \t:%5.2fm ' % y_max(velocidad))
```

2. *Ejecute el programa y observe la respuesta.*

Se pueden devolver varios valores con `return`.

Ejemplo 6.5.10. *Función que devuelve varios valores.*

Escriba el siguiente código:

```
>>> def cuadrados (x, y):
...       return x**2, y**2
...
>>> cuadrados(2, 4)
```

la respuesta será:

```
(4, 16)
```

Aquí Python crea una tupla cuyos elementos son los valores a devolver y esta tupla es la que se devuelve.

Ejercicio 6.5.1. Modifique el ejemplo 6.5.4 creando una función `mrua(x0,v0,a,t)` que devuelva los valores de posición, velocidad y aceleración para un movimiento rectilíneo uniformemente acelerado.

Ejercicio 6.5.2. Defina una función que calcule las raíces de la ecuación de segundo grado $ax^2 + bx + c = 0$ en función del signo de su discriminante $b^2 - 4ac$. Escriba un programa que pida los tres parámetros a, b y c de la ecuación y que llame a la función para calcular las raíces.

El argumento de una función también puede ser otra función.

Ejemplo 6.5.11. *Uso de funciones como parámetros. En este ejemplo utilizaremos una función que se pasará como parámetro de otra.*

1. *Escriba un fichero* `derivadas.py` *con el siguiente código:*

```
def cuadrado(x):
    return x**2

def finite_diff (f, x, h = 0.0001):
   '''
   aproxima la primera y segunda derivada
   de f en x; h tiene un valor por defecto 0.0001
   '''
   fm, f0, fp = f(x-h), f(x), f(x+h)
   df, ddf = (fp-fm)/(2*h), (fp-2*f0+fm)/h**2
   return df, ddf

x = 0.5
df, ddf = finite_diff(cuadrado, x)
print('Primera derivada = ', df)
print('Segunda derivada = ', ddf)
```

2. *Guarde el fichero, ejecute el programa y compruebe la respuesta.*

En el ejercicio anterior la función `cuadrado` se pasa a `finite_diff` como argumento.

Ejercicio 6.5.3. Modifique el ejemplo anterior para que se muestre la diferencia entre los valores exacto de la primera y segunda derivada y sus soluciones aproximadas.

Si la función tiene la forma de una expresión, puede ser definida de forma sencilla con la sentencia `lambda`, que permite funciones de una sola línea:

```
nombre_funcion = lambda arg1, arg2, ...: expresion
```

Ejemplo 6.5.12. *Uso de la sentencia* `lambda`.

Escriba en el modo interactivo de Python el siguiente código:

```
>>> c = lambda x, y : x**2 + y**2

>>> print(c(3, 4))
```

6.6. Módulos

Las funciones relacionadas entre sí se pueden agrupar en módulos. Hay muchos (cientos) módulos preparados por desarrolladores para realizar tareas relacionadas entre sí. Un módulo también puede contener **submódulos**. En ese caso el módulo principal se denomina **paquete** ('package').

Excepto en el caso de programas muy simples, es conveniente subdividir las tareas en funciones asociadas en módulos.

Ejemplo 6.6.1. *Pidiendo ayuda sobre módulos.*

Escriba en el modo interactivo de Python:

```
>>> help('modules')
```

Nos dará un listado de los módulos de Python desarrollados. Si quiere salir de la ayuda presione la letra `q`*. La ayuda de un módulo concreto se obtiene así:*

```
>>> help(nombre_del_módulo)
```

por ejemplo:

```
>>> help('time')
```

nos da ayuda acerca de funciones sobre tiempo y fecha.

Un programador puede diseñar un módulo juntando funciones en un mismo fichero. El nombre del fichero pasará a ser el nombre del módulo.

Ejemplo 6.6.2. *En este ejemplo se crea un fichero* `aritmetica.py` *que puede usarse como módulo. Define dos variables (cero y uno) y dos funciones (suma y resta).*

1. *Escriba el siguiente código en el fichero aritmetica.py:*

```
''' Define diversas funciones y variables '''
uno = 1
cero = 0

def suma (x, y):
    '''
    suma variables
    '''
    return x+y

def resta (x, y):
    '''
    resta variables
    '''
    return x-y
```

2. *Guarde el fichero.*

3. *Para hacer conocido todo lo definido en el fichero* `aritmetica.py` *en el modo interactivo de Python debe escribir:*

```
>>> from aritmetica import *
```

4. *Ahora ya se pueden invocar todas las funciones definidas en el módulo* `aritmetica`.

```
>>>print(uno)
1
>>> print(cero)
0
>>> suma(3, 4)
7
>>> resta(3, 4)
-1
```

5. *Otra forma de hacer conocido todo lo definido en el fichero* `aritmetica.py` *es:*

```
>>> import aritmetica
```

pero de esta forma la invocación de cada función necesita ser precedida del nombre del módulo al que pertenece (en este caso `aritmetica`*), por lo que para invocar ahora a las funciones se debe escribir:*

```
>>> print(aritmetica.uno)
1
>>> print(aritmetica.cero)
0
>>> aritmetica.suma(3, 4)
7
>>> aritmetica.resta(3, 4)
-1
```

La primera manera de utilizar un módulo, `from aritmetica import *`, hace que las variables y funciones del módulo pasen a ser conocidas por su nombre exacto (por ejemplo `suma` y muchas otras más en general, previamente desconocidas en el programa). Esto puede ser peligroso puesto que cualquier variable con el nombre `suma` (por ejemplo) que existiera previamente antes de `from aritmetica import *` desaparezca.

El segundo método, `import aritmetica`, es más seguro, pero su uso es más engorroso. Sólo se perdería una variable `aritmetica` si ya existía previamente. Para evitar teclear tanto al usar las funciones dentro del módulo aritmética se puede usar un "alias": `import aritmetica as ari`. De esa manera se invocaría `ari.suma`, etc. (el alias es `ari`).

También se pueden cargar funciones y variables concretas usando:

```
>>> from aritmetica import cero, suma
```

En este caso ni `uno` ni `resta` están disponibles para usarse.

El **contenido** de un módulo (o sea las variables y funciones que 'exporta') puede ser mostrado con la sentencia `dir(modulo)`. Se puede ver el contenido del módulo del ejemplo anterior escribiendo:

```
>>> import aritmetica
>>> dir(aritmetica)
```

Se pide ayuda sobre el contenido de un módulo mediante `help(modulo)`.

```
>>> import aritmetica
>>> help(aritmetica)
```

la salida que nos proporcionará dicha sentencia será de la forma:

```
Help on module aritmetica:

NAME
      aritmetica - Define diversas funciones y variables

FILE
      aritmetica.py

FUNCTIONS
      resta(x, y)
          resta variables
```

(continúa con más texto)

6.6.1. Variables globales y locales

Las variables definidas en un fichero fuera de una función son **globales** y se conocen a partir de su definición dentro del fichero (incluso dentro de las funciones).

Si escribimos:

```
>>> x = 3
>>> def f (y):
...   print(x, y)
>>> f('a')
```

la respuesta será:

```
(3, 'a')
```

Sin embargo si dentro de una función se asigna una variable de ese nombre se pierde el acceso a la 'vieja'.

Ejemplo 6.6.3. *Uso de variables globales y locales.*

1. *Escriba en un fichero el siguiente código:*

```
x = 3
def g (y):
  print(x, y) # usa la x antes definida
def f (y):
  x = 8 # crea una nueva variable local a la función
  print(x, y) # usa la nueva x; no hay acceso a la otra
g('a')
f('a')
print(x)
```

2. *Ejecute el programa y observe la respuesta.*

Si queremos asignar a una variable un valor dentro de una función se debe usar la sentencia `global`.

Ejemplo 6.6.4. *Uso de variables globales.*

1. *Cree un fichero con el siguiente código:*

```
x = 3
def f (y):
  global x # esta x no es nueva, sino la definida antes
  x = 8
  print(x, y)
f('a') # imprime (8, 'a')
print(x) # imprime 8
```

2. *Ejecute el programa y observe la respuesta.*

En el ejemplo anterior con la función se ha cambiado el valor de la variable global, se dice entonces que existe un *efecto lateral* y debe tenerse especial cuidado.

Se denomina *función pura* aquélla que no tiene efectos laterales (no usa variables globales, ni paso por referencia ni maneja ficheros).

Ejercicio 6.6.1. Con la función `finite_diff` definida en el ejemplo 6.5.11 cree una nueva función que calcule la velocidad y la aceleración de un móvil con movimiento rectilíneo a partir de la función que define la posición con respecto al tiempo de dicho móvil. Considere para la posición una función polinómica del tiempo. Para realizar el ejercicio cree un módulo llamado `cinematica_rectilineo` que contenga todas las funciones necesarias.

6.7. Algunos módulos estándar

Python viene con una biblioteca de módulos estándar. Algunos módulos son internos al intérprete y proporcionan acceso a las operaciones que no son parte del núcleo del lenguaje pero se han incluido por eficiencia o para proporcionar acceso a primitivas del sistema operativo, como las llamadas al sistema. El conjunto de dichos módulos es una opción de configuración. Algunos de estos módulos son:

- `math`: la mayoría de las funciones matemáticas en Python se encuentran en este módulo.
- `time`: nos da información sobre tiempo y fecha.
- `sys`: nos permite interaccionar con el sistema operativo.
- `os`: nos permite interaccionar con el sistema operativo con más funciones que el anterior.

6.7.1. Módulo `math`

En Python la mayoría de las funciones matemáticas se encuentran en el módulo `math`. Existen tres formas de acceder a las funciones dentro del módulo:

- Cargado todas las funciones del módulo con la sintaxis:
  ```
  from math import *
  ```
- Seleccionando las funciones que queremos usar del módulo:
  ```
  from math import func1, func2
  ```
- Habilitando el módulo (método utilizado por la mayoría de los programadores) de la forma:
  ```
  import math
  ```

Dentro de `math` tenemos las funciones: `acos`, `acosh`, `asin`, `asinh`, `atan`, `atan2`, `atanh`, `ceil`, `copysign`, `cos`, `cosh`, `degrees`, `erf`, `erfc`, `exp`, `expm1`, `fabs`, `factorial`, `floor`, `fmod`, `frexp`, `fsum`, `gamma`, `hypot`, `isinf`, `isnan`, `ldexp`, `lgamma`, `log`, `log10`, `log1p`, `modf`, `pow`, `radians`, `sin`, `sinh`, `sqrt`, `tan`, `tanh`, `trunc`

y las variables:

```
e = 2.7182818284590451
```

```
pi = 3.1415926535897931
```

Ejemplo 6.7.1. *Uso del módulo* `math`. *En el siguiente ejemplo se define una función* `maxmin(f, a,b, n=1000)` *que devuelve los valores máximo y mínimo de la función f(x) (evaluada en n puntos) en el intervalo (a,b).*

1. *Escriba en un fichero el siguiente código:*

```
def maxmin(f, a, b, n=1001):
  h = (b-a)/float(n-1) # el 'paso'
  X = [a+i*h for i in range(n)] # comprensión de lista
  Y = [f(x) for x in X] # comprensión de lista
  return max(Y), min(Y)

from math import cos, pi

print maxmin(cos, -pi/2, 2*pi)
```

2. *Ejecute el programa y observe la respuesta.*

Ejercicio 6.7.1. Modifique el ejemplo 6.5.11 para que calcule la primera derivada de las siguientes funciones:

1. $f(x) = e^x$ en $x = 0$,
2. $f(x) = e^{-2x}$ en $x = 0$,
3. $f(x) = \cos x$ en $x = 2\pi$,
4. $f(x) = \ln x$ en $x = 1$.

Use h= 0.01. En cada caso, haga que el programa escriba la diferencia entre el valor exacto de la derivada (conocido analíticamente) y el resultado dado por la derivada numérica.

Ejercicio 6.7.2. La integral de una función $f(x)$ en un intervalo $[a, b]$ puede realizarse en primera aproximación dibujando una línea recta que una los puntos $(a, f(a))$ y $(b, f(b))$, y calculando el área del trapecio bajo la línea de la forma:

$$\int_a^b f(x)\,\mathrm{d}x \simeq \frac{b-a}{2}[f(a) + f(b)]$$

Defina una función `integral(f, a, b)` que devuelva el valor aproximado de la integral mediante la ecuación anterior de las siguientes funciones:

1. $\int_0^\pi \cos x\,\mathrm{d}x$,
2. $\int_0^\pi \sin x\,\mathrm{d}x$,
3. $\int_0^{\pi/2} \sin x\,\mathrm{d}x$

En cada caso, haga que el programa escriba la diferencia entre el valor exacto de la integral (conocido analíticamente) y el resultado dado por la integral numérica.

Ejercicio 6.7.3. Se puede mejorar facilmente la aproximación de la integral utilizada en el ejercicio anterior aproximando la función $f(x)$ mediante una primera línea que una $(a, f(a))$ con el punto medio $(c, f(c))$ entre a y b y otra segunda línea que una el punto medio $(c, f(c))$ con $(b, f(b))$. El punto medio c es igual a $\frac{1}{2}(a+b)$. El área bajo estas dos líneas es el área de dos trapecios que nos proporcionarán una nueva expresión aproximada de la intergral de $f(x)$. Obtenga la nueva expresión aproximada de la integral e implemente una nueva función en Python que calcule de nuevo las integrales aproximadas de las funciones del ejemplo anterior.

6.7.2. Módulo `time`

Este módulo nos proporciona funciones para el manejo de fechas y del tiempo. Algunas interesantes son:

- `time.time()` nos devuelve el número de segundos desde un instante inicial (dependiente del sistema operativo).
- `time.ctime(x)` nos devuelve una cadena con la fecha que corresponde al tiempo `x` en segundos. Si no se indica `x` se usa la fecha y hora actuales.
- `time.sleep(x)` suspende la ejecución durante `x` segundos.
- `time.clock()` nos devuelve el tiempo de ejecución de CPU hasta ese momento del intérprete de Python.

Ejemplo 6.7.2. *Ejemplo de uso de* `time.clock()`

1. *Escriba en un fichero con el nombre pruebacos.py:*

```
import time, math
t = time.clock()
for i in range(1000000):
   math.cos(i)
dt = (time.clock()-t)/1000000
print('un coseno tarda%e s de CPU'% dt)
```

2. *Ejecute el programa:*

```
$ python pruebacos.py
```

3. *La respuesta será:*

```
un coseno tarda 5.200000e-07 s de CPU
```

6.7.3. Módulo `sys`

Este módulo nos proporciona funciones para interaccionar con el sistema operativo. Algunas interesantes son:

- `sys.platform`: es una cadena con el nombre del sistema operativo.
- `sys.exit(n)`: acaba el programa, devolviendo un número entero `n` al sistema operativo como código de finalización.
- `sys.argv`: es una lista con los argumentos que hemos utilizado al llamar al intérprete de Python. Sirve para pasar información a nuestro programa. El primer elemento de la lista es siempre el nombre de nuestro programa.

Ejemplo 6.7.3. *Ejemplo de uso de* `sys.argv`

1. *Tecleemos el fichero* `param.py`*:*

```
import sys
for i, a in enumerate(sys.argv):
  print(i, a)
```

2. *Ejecute:*

```
$ python param.py 1.2 'y=2*log(x)' 'mi nombre'
```

3. *la respuesta será:*

```
(0, 'param.py')
(1, '1.2')
(2, 'y=2*log(x)')
(3, 'mi nombre')
```