Programación en Python para ingeniería

J. París, F. Navarrina & GMNI

Grupo de Métodos Numéricos en Ingeniería

E.T.S. Enxeñería de Camiños, Canais e Portos

Índice

1.- Introducción
- 1.1.- Conceptos generales
- 1.2.- Modos de interacción con Python

2.- Estructura y estilo de programación
- 2.1.- Estilo de codificación y programación
- 2.2.- Estructura de un programa
- 2.3.- Tipos de variables
- 2.4.- Clases y objetos

3.- Operadores
- 3.1.- Operadores aritméticos y lógicos
- 3.2.- Operadores relacionales

4.- Sentencias de control
- 4.1.- Bucles
- 4.2.- Sentencias de control lógicas: if
- 4.3.- Sentencias de control lógicas: match

5.- Ficheros
- 5.1.- Apertura y cierre de archivos
- 5.2.- Lectura y escritura en archivos
- 5.3.- Estándar input y output

6.- Estructuras de datos
- 6.1.- Listas
- 6.2.- Strings (Cadenas de caracteres)

7.- Gestión y utilización de funciones
- 7.1.- Definición y utilización de funciones
- 7.2.- Uso avanzado de funciones

8.- Módulos
- 8.1.- Incorporación de módulos internos
- 8.2.- Incorporación de módulos externos

9.- Módulo NumPy
- 9.1.- Funciones más habituales
- 9.2.- Lectura de datos de archivos txt
- 9.3.- Operaciones con array

10.- Command line
- 10.1.- Información desde el command line

11.- Bibliografía y ejemplos
- 11.1.- Bibliografía y ejemplos

1.1- Conceptos generales

- Es un lenguaje de propósito general
- Es un lenguaje de alto nivel (cercano al usuario)
- Es un lenguaje interpretado
- El intérprete de Python está disponible para todos los sistemas operativos bajo licencia GPL
- Es un lenguaje de programación modular (y existen numerosos módulos estándar disponibles para su uso)
- Fue desarrollado a principios de los 90 por Guido Van Rossum
- Recibe su nombre del programa de TV "Monty Python's Flying Circus" de la BBC
- En 2008 se lanzó la versión 3.0 de Python, que rompió la compatibilidad con versiones anteriores

1.2- Modos de interacción con Python

- Modos de interacción con Python
  - Modo interactivo:
    - ✓ Para acceder al intérprete Python basta con teclear en una terminal: "python"
    - ✓ Para salir del intérprete basta con teclear: "quit()"
    - ✓ Se pueden introducir instrucciones individualmente
    - ✓ Se pueden recuperar instrucciones anteriores de forma similar a una terminal de comandos
  - Modo script:
    - ✓ Las instruccciones se pueden indicar de forma conjunta en un archivo de comandos
    - ✓ La invocación del intérprete se realiza como: "python *archivo_de_comandos.py*"
    - ✓ Si se desea se pueden invocar módulos estándar como: "python -m *module [arg] ...*"
    - ✓ La codificación del archivo de comandos se realiza por defecto en UTF-8, si bien es recomendable utilizar solo la codificacion ASCII por compatibilidad
    - ✓ Los comentarios y líneas de comentarios comienzan con el símbolo #
    - ✓ No se indica el ";" al final de cada instrucción

2.1- Estilo de codificación y programación

Estilo de codificación

- Cada línea corresponde a una instrucción
- Se recomienda usar sangrías de 4 espacios, no tabuladores
- La indentación permite identificar los bloques de código
- Se recomienda no hacer líneas de más de 79 caracteres (para facilitar visibilidad)
- Si una línea ocupa más de 80 caracteres se puede cortar la instrucción con el caracter \ y continuar en la línea siguiente.
- Se recomiendan los comentarios en una sóla línea
- Se recomienda usar espacios alrededor de operadores y luego de las comas, pero no directamente dentro de paréntesis
- Se recomienda utilizar solo caracteres ASCII.

2.2- Estructura de un programa

Estructura de un programa:

- Los programas en Python se diseñan de forma modular (con funciones)
- El programa principal se escribe al final del archivo después de todas las funciones
- Las funciones deben definirse en el archivo antes de utilizarse (lenguaje interpretado)
- Las instrucciones en el programa principal se escriben desde el inicio de cada fila, sin indentación
- No existen comandos de preproceso estilo #DEFINE de C
- Los módulos externos se incorporan al programa al inicio del mismo o antes de utilizarlos

2.3- Tipos de variables

- Tipos de variables:
  - *int* (variables enteras)
  - *float* (variables reales, tamaño por defecto normalmente doble precisión)
  - *bool* (*True* o *False*)
  - *strings*, cadenas de caracteres (se ven posteriormente en 2)
- No es necesario declarar las variables.
- Python decide por nosotros el tipo de variable.
- En caso de duda se recomienda forzar el tipo con funciones de conversión.
- Se recomienda forzar el tipo mediante funciones de conversión de tipos.
- Funciones de conversion de tipos de variables:
  - *int(x)* $\rightsquigarrow$ Convierte a entera x (sea del tipo que sea)
  - *float(i)* $\rightsquigarrow$ Convierte a real i (sea del tipo que sea)
  - *str(x)* $\rightsquigarrow$ Convierte a string x (sea del tipo que sea)

2.4- Clases y objetos

- Clases:
  - Se definen como:

```
class nombre_clase:
    variable_1
    variable_2
    def f(...):
        ...
```

  - Pueden contener funciones definidas como: *def ...*
  - Se pueden crear objetos de una determinada clase como:

```
x = nombre_clase()      # Crea el objeto x vacío de clase tipo nombre_clase
```

  - Dentro de cada objeto los diferentes atributos (miembros) se identifican como:

```
a = x.variable_1      # Toma el atributo variable_1 del objeto x.
c = x.f(...)      # Aplica la función f del objeto x con los datos (...).
```

3.1- Operadores aritméticos y lógicos

- Operadores aritméticos:
  - Los operadores aritméticos habituales se utilizan normalmente como en cualquier lenguaje de programación
  - Se incorpora la división modular al igual que en Lenguaje C mediante (*%*)
  - Los operadores incrementales de C (*++*, - -) no forman parte de Python
  - Sí se pueden utilizar los operadores incrementales (*+=*, *-=*, **=* y */=*)
- Operadores lógicos:
  - AND: Se utilizan y se escriben directamente con la instrucción "*and*" entre espacios
  - OR: Se utilizan y se escriben directamente con la instrucción "*or*" entre espacios
  - Negación: Se utilizan y se escriben anteponiendo "*not*" al valor o variable booleana

3.2- Operadores relacionales

- Operadores relacionales:
  - Se escriben al igual que en C entre espacios:
  - $<$ (menor estricto) $\longrightarrow$ Ej. $j < 5$
  - $<=$ (menor o igual) $\longrightarrow$ Ej. $j <= 5$
  - $>$ (mayor estricto) $\longrightarrow$ Ej. $j > 5$
  - $>=$ (mayor o igual) $\longrightarrow$ Ej. $j >= 5$
  - $==$ (coincide con) $\longrightarrow$ Ej. $j == 5$
  - $!=$ (no coincide con)$\longrightarrow$ Ej. $j != 5$

4.1- Bucles

- Bucles for

```
for i in range(10):     # las lineas a repetir van indentadas 4 espacios.
                        # la instrucción finaliza con :
    print(i)
    i = 5               # Obsérvese que a pesar de esto el contador seguirá igual
```

  - La forma habitual de establecer secuencias numéricas es con la función *range*:

    range(ini,fin) $\rightsquigarrow$ {ini, ini+1 ..., fin-1}
    range(ini,fin,salto) $\rightsquigarrow$ {ini, ini+salto, ...} # El valor "fin" nunca se alcanza
    range(fin) = {0,1,...,fin-1}

  - El rango de valores también puede ser una comparación con elementos de una lista

```
words = ['cat', 'dog', 'mouse']
for w in words:
    print(w, len(w))
```

  - La sentencia *break* termina el bucle
  - La sentencia *continue* se salta el resto de la iteración y pasa a la siguiente.

4.1- Bucles

- Bucles "while"

```
a=1
b=1
while b < 20:
    print(b)
    c=a+b
    a=b
    b=c
```

$\rightsquigarrow$ [1, 2, 3, 5, 8, 13]

  - Los dos puntos indican el final de la condición de repetición del "while"
  - Todas las sentencias afectadas por la instrucción "while" deben ir indentadas con 4 espacios con respecto al while
  - La sentencia *break* termina el bucle
  - La sentencia *continue* se salta el resto de la iteración y vuelve a comprobar la condición de repetición

4.2- Sentencias de control lógicas: if

- Sentencia "if"

```
b = int(input('Escriba b: '))
if b < 20:
    print('b < 20')
elif b < 30:            # La comprobacion else if es opcional
    print('20 <= b < 30')
else:                   # La comprobacion else es opcional
    print('b >= 30')
```

  - La comprobación de la condición finaliza necesariamente con :
  - Las sentencias afectadas por la condición "if" deben ir indentadas con 4 espacios con respecto al if
  - Las sentencias afectadas por las condiciones "elif" y "else" también deben ir indentadas con 4 espacios

4.3- Sentencias de control lógicas: match

- Sentencia "match" (equivalente al switch de C). Sólo a partir de Python 3.10

```
match option:
    case 1:
        print('Option 1')
    case 2:
        print('Option 2')
    case 3:
        print('Option 3')
    case _:
        print('Non specified option')
```

  - Cada "caso" coincidente excluye el resto (a diferencia del switch en C)
  - El caso "_" es la opcion final por defecto

5.1- Apertura y cierre de archivos

- Apertura de archivos con "open"

  ref_archivo = open('nombre_archivo','cd')

  - *ref_archivo* es la variable que identifica el archivo que se va a abrir
  - *nombre_archivo* es un string con el nombre (y ruta, en su caso) del archivo a abrir
  - *cd* son opciones de modo de apertura:
    - ✓ *c* vale "r" ⇝ Lectura, valor por defecto. Error si el archivo no existe
    - ✓ *c* vale "w" ⇝ Escritura. Crea el archivo si no existe
    - ✓ *c* vale "a" ⇝ Añade contenido. Crea el archivo si no existe
    - ✓ *c* vale "x" ⇝ Crea el archivo si no existe. Error si existe
    - ✓ *d* es opcional. Si se añade con el valor "b" el archivo será binario

  Ej.: *finput = open('datos.txt','r')* ⇝ Apertura de 'datos.txt' para lectura
  print('Nombre de archivo: ',finput.name)

- Cierre de archivos con "close"

  Ej.: *finput.close()* # Cierra el archivo con nombre de referencia *finput*

5.2- Lectura y escritura en archivos

- Escritura de datos

  Ej.:

```
foutput = open('resultados.txt','w')

foutput.write('Escribimos un texto')                    # Escribimos un texto
foutput.write('Texto con valor ',v,' intercalado')      # Escribimos texto con un
                                                        # valor v intercalado
foutput.write('Texto y {:5d},{:15.6e}'.format(n,v))     # Escribimos texto con valores
                                                        # intercalados con formato
```

  El formato puede adoptar alguno de los siguientes tipos de formato (entre otros):

  - {:5d} ⇝ Valor entero con 5 dígitos
  - {:15.6e} ⇝ Valor real en notación científica con 15 dígitos y 6 decimales
  - {:s} ⇝ Cadena de caracteres (string)

5.2- Lectura y escritura en archivos

- Lectura de datos (ver apartado de Estructuras de datos)

Ej.: *finput = open('datos.txt','r')*

```
t = finput.readline()        # Lee una linea completa del archivo y
                             # la guarda en el string "t".
tint = t.split()             # Divide el string palabra a palabra y
                             # crea una lista (vector) de strings.
n = int(tint[0])             # Convierte el primer campo de la línea en
                             # un entero y lo guarda en "n".
x = float(tint[1])           # Convierte el segundo campo de la línea
                             # en un real y lo guarda en "x".
```

  - La estructura "finput" dispone de más funciones asociadas que se pueden utilizar. Estas funciones se pueden consultar en la documentación oficial de Python (www.python.org)

5.3- Estándar input y output

- Entrada de datos estándar

  t = input('Mensaje previo') # El mensaje previo es opcional

  - almacena como cadena de caracteres (por defecto) en "t" la entrada de datos
  - Opcionalmente se puede cambiar el tipo de dato como:

  x = float(input('Mensaje previo'))

- Si se desea leer la información de forma binaria se puede utilizar:

  t = sys.stdin.read(b) # Lee "b" bytes de la entrada estándar y
  # los almacena como string en "t".

  Para ello es necesario importar antes el módulo "sys" mediante *import sys*.

5.3- Estándar input y output

- Saída de datos estándar (introduce salto de línea al final por defecto)

```
print('Mensaje texto',variables)   # Se pueden combinar separados por comas
```

Ej.:

```
print('r = ',r)                          # Imprime el texto y el valor de la variable
print('r = {:e}'.format(r))              # La variable se muestra con notación científica
print('r = {:25.16e}'.format(r))         # La variable se muestra con notación científica,
                                         # 25 dígitos y 16 decimales
print('n = {:5d}, r = {:25.16e}'.format(n,r))   # n como entera y r como real
print('Nombre = {:s}'.format(str_name))  # muestra texto y string encadenados
print('Nombre = ' + str_nombre)          # equivalente al anterior
```

- Si se desea escribir la información de forma binaria se puede utilizar:

```
sys.stdout.write('x = {:f}'.format(x))   # Igual que print pero sin salto al final.
```

Para ello es necesario importar antes el módulo "sys" mediante *import sys*.

5.3- Estándar input y output

- Saída de datos estándar (continuación)

Otras opciones de escritura son:

- *print(r'\n r')* # No interpreta los caracteres especiales tipo '\n'.
- *print(f'r = x')* # Intercala el valor de la variable x (indicada entre llaves).

Ejemplo de lectura de stdin línea a línea:

```
import sys
for line in sys.stdin:
    if 'Exit' == line.rstrip():
        break
    print('Linea leida: ****{:s}****'.format(line))
print('Done ')
```

6.1- Listas

Listas (forma general de vectores)

- Se pueden definir como: *lista = [1, 4, 9, 16, 25]*
- Pueden contener valores de tipos diferentes, aunque no es lo habitual
- Se pueden segmentar e indexar al igual que los strings (e.g. *lista[:]*)
- Se pueden concatenar con el símbolo "+"
- Se pueden añadir términos a la lista como: *lista.append(valor)*
- Se pueden eliminar partes de una lista como: *lista[2:5] = []* (elimina componentes 2, 3 y 4)
- *len(lista)* indica el número de elementos de la lista

6.1- Listas

Comandos más habituales de modificación de listas:

- *list.append(x)* → Agrega un ítem al final de la lista *list*
- *list.extend(lista2)* → Extiende la lista *list* agregándole todos los ítems de *lista2*
- *list.insert(i, x)* → Inserta el ítem *x* justo antes de la posición *i* de la lista *list*
- *list.remove(x)* → Quita el primer ítem de la lista *list* cuyo valor sea *x*. Si no existe indica un error.
- *list.clear()* → Elimina todos los elementos de la lista *list*
- *list.count(x)* → Retorna el número de veces que x aparece en la lista *list*
- *del list[a:b]* →

6.1- Listas

Formas matriciales

- Se indican como listas dentro de otra lista.
- Se almacenan por filas.
- Cada fila puede tener diferente número de elementos.

  Ejemplo de matriz:

  matriz = [[11,12], [21,22,23], [31,32,33]]
- Luego se pueden operar como:

```
nfilas = len(matriz)
for i in range(nfilas):
    print(matriz[i][-1])      # Muestra el último valor de cada fila.
```

- La gestión de matrices se simplifica con las librerías del módulo NumPy.

6.2- Strings (Cadenas de caracteres)

Strings

- Las cadenas de caracteres (*strings*) se encierran entre comillas simples o dobles (indistintamente)
- Las cadenas de caracteres pueden contener varias líneas si se definen mediante triple comilla
- Las cadenas de caracteres colocadas una a continuación de la otra se concatenan automáticamente. Con variables no funciona de este modo.
- Variables tipo string y textos pueden concatenarse con "+"
- Se puede hacer referencia a posiciones dentro de un string o a porciones del mismo como:
  - string[4:6] $\rightsquigarrow$ Caracteres desde el 4 (inlcuido) hasta el 6 (excluido)
  - string[4:] $\rightsquigarrow$ Caracteres desde el 4 hasta el final del string
  - string[:4] $\rightsquigarrow$ Caracteres desde el inicio hasta el 4 (excluido)
  - string[-2] (segundo caracter desde el final)
- Los string no se pueden modificar parcialmente.
- Para modificarlo se compone un nuevo string con los cambios incorporados

7.1- Definición y utilización de funciones

Definición de funciones

- Las funciones se definen como:

  def nombre_funcion*(*lista_argumentos*):*

  - Todas las instrucciones de esa función deben ir con el correspondiente sangrado
  - El paso de argumentos se hace por valor de la referencia:
    - ✓ De objetos y listas se envían copias de las referencias.
    - ✓ De variables se envían copias de los valores (los cambios no permanecen)
  - La llamada a la función desde el programa principal se realiza como:

    valores_retorno = nombre_funcion*(*lista_argumentos*)*

- La función tiene que estar definida en el archivo de comandos antes de ser llamada.

7.1- Definición y utilización de funciones

- Definición de las funciones
  - Las funciones se definen mediante el comando "def" como:

    def nombre_funcion*(*lista_de_variables*):*

  - Y a continuación se indican los comandos propios de la función.
  - Todas las sentencias de la función deben ir indentadas 4 espacios.
  - Al final de la función si existen valores de retorno se indican como:

    return valor1, valor2, etc.

    Ejemplo:

```
def cuadrado(x):
    a = x * x
    return a
    .
    .
    .
z = cuadrado(x)
```

  - La sentencia *return* puede omitirse si no hay valores de retorno.

7.2- Uso avanzado de funciones

- Funciones con valores por omisión (por defecto):
  - Es opcional especificar los valores. Si no se espefican se adoptan los valores por defecto

```
def ask_ok(prompt, retries=4, reminder='Por favor, intentelo de nuevo!'):
    while True:
        ok = input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
        if retries ¡ 0:
            print('Respuesta de usuario invalida')
        print(reminder)
```

7.2- Uso avanzado de funciones

- Esta función se puede utilizar como:
  - *ask_ok('Desea salir ahora?')* que si la respuesta no es válida repite la pregunta por defecto 4 veces mostrando el mensaje: "Por favor, intentelo de nuevo!"
  - *ask_ok('Desea salir ahora?',2)* que si la respuesta no es válida repite la pregunta 2 veces mostrando el mensaje: "Por favor, intentelo de nuevo!"
  - *ask_ok('Desea salir ahora?',2, 'Respuesta incorrecta')* que si la respuesta no es válida repite la pregunta 2 veces mostrando el mensaje: "Respuesta incorrecta"
- Los valores por omisión (por defecto) se asignan una sola vez cuando se crea la función.

8.1- Incorporación de módulos internos

Incorporación de módulos internos:

- Los programas Python pueden incorporar funcionalidades a través de módulos internos
- Los módulos internos se añaden mediante la función `import` como:
  import modulo_interno
- La importación se suele hacer al principio del programa
- Los módulos incorporados se comportan como objetos formados tanto por variables como por funciones:
  - *resultado = modulo.funcion(valores)*
  - *modulo.valor_propio*
- Algunos de los módulos propios más habituales de Python son:
  - sys, math, random

8.2- Incorporación de módulos externos

Incorporación de módulos externos (librerías):

- Los programas Python pueden incorporar funcionalidades a través de módulos externos.
- Los módulos externos son ficheros (*.py) y se añaden mediante `import` como: *import modulo_ext*
- La importación se suele hacer al principio del programa.
- El intérprete busca el módulo en el sistema siguiendo este orden:
  - Comprueba si es un módulo interno del sistema (listados en listed in *sys.builtin_module_names*)
  - Comprueba si hay un archivo *modulo_ext.py* en la lista de carpetas indicada en *sys.path*
  - La lista *sys.path* incluye la propia carpeta de ejecución del script.
- Los módulos incorporados se comportan como objetos formados por funciones:
  - *resultado = modulo.funcion(valores)*
- Algunos de los módulos externos más habituales de Python son:
  - Matplotlib, SciPy, NumPy, ...

9.1- Funciones más habituales

Funciones más habituales del módulo NumPy

- Antes de empezar importamos el módulo NumPy:

  import numpy as np # Lo renombramos por comodidad como np

- Y ya podemos utilizar funciones específicas de NumPy:
  - *ones(shape,dtype)* ⇝ "Array" de unos formado por "shape" elementos.

    m_real = np.ones(shape=(2,3),dtype='float64')
    m_real = np.ones((2,3),'float64') # Equivalente al anterior
    m_ent = np.ones(shape=(3,4),dtype='int32')

  - *zeros(shape, dtype)* ⇝ "Array" de ceros compuesto de "shape" elementos.

    m_real = np.zeros(shape=(2,3),dtype='float64')
    m_real = np.zeros((2,3),'float64') # Equivalente al anterior
    m_ent = np.zeros(shape=(3,4),dtype='int32')

  - *m_ent = np.array(range(1,20), dtype='int64')*
  - *m_real = np.array([x**2 for x in range(20)], dtype='float64')*
  - *meshgrid(x,y)* ⇝ Genera una malla 2D combinando dos arrays x, y.

9.2- Lectura de datos de archivos txt

Carga de datos de un archivo:

matriz = np.loadtxt('ArchivoDatos.txt', delimiter = ' ')

- De un array existente podemos obtener información con:
  - *matriz.ndim* ⇝ Proporciona el número de dimensiones de nuestro array.
  - *matriz.dtype* ⇝ Es un objeto que describe el tipo de elementos del array.
  - *matriz.shape* ⇝ Devuelve la dimensión del array (para matrices en una tupla)

9.3- Operaciones con array

Operaciones con "array":

- Si los arrays se generan con funciones de NumPy las operaciones son muy sencillas:
  - Suma: *c = a + b # Suma de "arrays" (matrices o vectores)*
  - Producto escalar: *c = np.dot(a,b) # Producto algebraico de "arrays"*
  - Producto vectorial: *c = np.cross(a,b) # Producto vectorial de "array"*
  - Producto componente a componente: *c = a*b # Producto elemento a elemento*
  - Transpuesta: *c = np.transpose(a) # Transpuesta del "array"*
  - Copia: *mc = np.copy(ma) # Copia ma en mb*

9.3- Operaciones con array

Operaciones con "array":

- Componente a componente con bucles:

```
for f in M:      # f es una fila de M (lista)
    for x in f:      # x es la componente de la fila f
        print(x, end=' ')
    print()

            for i, f in enumerate(M):      # i = indice fila, f = fila de M (lista)
                for j, x in enumerate(f):
                    M[i, j] *= x       # Eleva al cuadrado cada componente
            print(M)

for j in range(m2.shape[1]):      # Gestion de datos por columnas (ineficiente)
    for i in range(m2.shape[0]):       # m2.shape[0] indica el numero de filas de m2
        print(m2[i,j])
    print()
```

10.1- Información desde el command line

Incorporación de información del command line:

- La incorporación de información desde el command line es muy similar a la de C.
- Se utiliza de forma incluso más sencilla, pero opaca.
- En la terminal se ejecuta como: `python prog.py campo_1 campo_2`
- Para ello es necesario incorporar el módulo "sys" como: *import sys*
- *sys.argv[i]* devuelve un string con el campo *i* del command line:
  - sys.argv[0] $\rightsquigarrow$ nombre del programa Python en ejecución (e.g. `prog.py`)
  - sys.argv[1] $\rightsquigarrow$ primer campo del command line (e.g. `campo_1`)
- El número de campos introducidos se puede obtener mediante:

  argc = len(sys.argv)

11.1- Bibliografía y ejemplos

La información de este breve manual se ha extraído de:

- La web oficial del Python: https://www.python.org
- La web oficial del módulo NumPy: https://numpy.org

Los ejemplos de aplicación se han obtenido de diversos ejemplos de aplicación en múltiples páginas web y de desarrollo propio

Se pueden encontrar ejemplos en el apartado de Software de la página web:

https://caminos.udc.es/info/asignaturas/grado_itop/503/index.html

That's All Folks !!!