Guía de iniciación al Python

1. Primero de todo arrancamos el programa de desarrollo integrado más sencillo de Python, que es el IDLE, buscando Python2.x en la lista de programas de Windows.
2. Lo primero que vamos a hacer con el Python son unas sencillas operaciones sobre la consola:

```
>  valor_inicial=10e8
>  interes_por_periodo=0.05
>  numero_de_periodos=30
>  pago_final= valor_inicial*(1+ interes_por_periodo)** numero_de_periodos
>  print pago_final

432194237.515

>  pago_final

432194237.515

>  pago_final_a_interes_simple= valor_inicial*(1+ interes_por_periodo)
>  print pago_final_a_interes_simple

105000000.0

>  j

Traceback (most recent call last):

  File "<pyshell#24>", line 1, in <module>

    j

NameError: name 'j' is not defined

>  1j #ahora si significa la unidad imaginaria (‘#’ indica comentario)

1j

>  complex(0,1) #alternativamente un numero complejo x+jy se representa como
   complex(x,y)
>  a=1.5+0.5j
>  a.real

1.5

>  a.imag

0.5
```

> float(a)

Traceback (most recent call last):

File "<pyshell#27>", line 1, in <module>

float(a)

TypeError: can't convert complex to float; use abs(z)

> abs(a)

1.5811388300841898

> 1+_ #'_' es el valor del último cálculo que ha salido en pantalla

2.5811388300841898

> round(_,2)

2.5800000000000001

Si queremos ver de qué tipo es una variable podemos usar el comando *type*

> a=2
> type(a)
> type(2)
> type(2.)
> type("hola")

3. Ahora vamos a ver cómo trabaja Python con las cadenas de caracteres:

   Hay tres tipos de comillas: ' '; " "; " " " " " "

   Tipo ' ' o tipo " "

   Se pueden usar unas u otras indistintamente pero no se pueden repetir dos veces si son del mismo tipo porque la segunda vez que intentamos abrirlas se interpretará como un cierre de las primeras. Esto se puede evitar "protegiendo" las comillas con una barra \. Prueba a poner

> "minoría"
> "Yo: "minoria absoluta" "
> "Yo: 'minoria absoluta' "
> 'Yo: "minoria absoluta" '
> "Yo: 'minoria absoluta' "
> "Yo: \"minoria absoluta\" "

   Se pueden usar comillas triples " " " " " " (o triples-dobles) para que algunos de estos problemas no ocurran ya que dentro de ellas todas las comillas se pueden usar con plena libertad, e incluso añadir retornos de carro

```
>   “ “ “ Elvis: ‘ I‘m so lonely…lonely’ ” ” ”
>   “ “ “ Elvis: ‘ I‘m so

    lonely…lonely’ ” ” ”
```

Para los demás casos y en cualquier línea de código, no solamente dentro de la definición de una cadena de caracteres, se utiliza una barra \ para dar paso a otra línea

```
>   a=5+ \

6
```

La concatenación de cadenas se puede hacer con un + o una simple yuxtaposición

```
>   “Sucede” + ‘que me canso’
>   ‘caminando por’ ‘la calle yo te vi’
```

Otra manera interesante de introducir cadenas es a través de formatos

```
>   “Rudy %s” %(“cogio su fusil”)
>   “Rudy %i %i” %(42, 42)
>   “Rudy %f %f” %(42, 42)
>   “Rudy %2.3f %2.3f” %(42, 42)
>   “Rudy %E %E” %(42, 42)
```

Para extraer partes de las cadenas su puede utilizar los subíndices. Fíjese que el primer carácter ocupa la posición 0 y que el segundo subíndice indica un “hasta”, sin incluir esa posición.

```
>   word=”Hello”
>   print word[0:2]
>   print word[:2]
>   word[:2]
>   print word[2:4]
>   print word[2:12]
>   print word[2:]
>   word[2:]
>   word[-1:2]
>   word[2]
```

Los retornos de carro dentro una cadena se pueden hacer, como hemos visto, con las triples-dobles comillas o usando \n. Pruébense las siguientes órdenes

```
>   sentence=”hola, \n ¿que tal?”
>   sentence
>   print sentence
```

```
>   sentence=”hola, \n \

¿que tal?”

>   sentence
>   print sentence
>   sentence=”hola, \

¿que tal?”

>   sentence
>   print sentence
>   sentence=”hola, \n \ ¿que tal?”
>   sentence
>   print sentence
```

Sin embargo, anteponiendo una ‘r’ a las comillas hacemos que \n no se interprete como un retorno de carro

```
>   sentence=r”hola, \n ¿que tal?”
>   sentence
>   print sentence
```

Hasta aquí hemos evitado los acentos. Pruébese lo siguiente

```
>   word=”sofá”
>   word
>   print word
```

También se pueden hacer otras operaciones con las cadenas, por ejemplo multiplicarlas

```
>   word=”casa”
>   word*5
```

Las posiciones de las cadenas **no** se pueden redefinir. Inténtese

```
>   word=”casa”
>   word[3]=’i’
```

Los índices negativos solos implican contar desde el final

```
>   word=”building”
>   word[-1]
>   word[-2]
>   word[-2:]
>   word[:-2]
```

4. ***Nombres de variables.*** Los siguientes nombres no se pueden usar porque están reservados por el sistema: and, assert, break, class, continue, def, del, elif, else, except, exec, finally, for, from, global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, while, yield
5. ***Tipos de tablas.*** Existen tres tipos de tablas de datos: los tuplas, las listas y los diccionarios. Los dos primeros se llaman secuencias.

   Tuplas: Son secuencias de valores separadas por comas y comprendidas por paréntesis. El elemento n-ésimo se llama con *nombre_de_la_tupla[n]*, y se pueden aplicar las normas para los índices que hemos visto para las cadenas

```
> paises_de_la_UE=(“Alemania”, “Austria”, “Bélgica”, “Bulgaria”, “Chipre”,\
  “República Checa” , “Dinamarca”, “Estonia”, “Finlandia”, “Francia”,\
  “Grecia”, “Hungría”, “Irlanda”, “Italia” ,“Letonia”, “Lituania”,\
  “Luxemburgo”, “Malta”, “Países Bajos”, “Polonia”, “Portugal”, “Rumania”,\
  “Eslovaquia”, “Eslovenia”, “España”, “Suecia”,“Reino Unido”)
> paises_de_la_UE[2:4]
> paises_de_la_UE[24]
> paises_de_la_UE[1]=”Österreich”
```

   Las tuplas se pueden definir también sin paréntesis, aunque no es recomendable como metodología general

```
> r=1,2,3,’ahora’
> x,y,z,t=r #esta es una forma ágil (no recomendable siempre) de desempaquetar
  una tupla
> x
> y
> x,y,z,t
```

   Si se quiere definir una tupla de un elemento, hay que escribir una coma (,) al final de la asignación, ya sea con o sin paréntesis

```
> futuro=’ahora’,
> futuro
> futuro=(‘ahora’)
> futuro
> futuro=(‘ahora’,)
> futuro
```

   Listas: Son secuencias de valores separadas por comas y comrendidas entre corchetes []. Sí se pueden modificar mediante redefinición y *operaciones de clase*.

```
> paises_candidatos=[“Croacia”, “Macedonia”, “Turquía”]
> paises_candidatos[1]= “Hrvatska”
> paises_candidatos.append(“Islandia”)
> paises_candidatos.extend(“Suiza”)   #extend es una operación de clase de una
                                      #lista que toma como argumento otra lista,
                                      #pero no una tupla ni cualquier otra cosa
> paises_candidatos.extend([“Suiza”])
```

Las secuencias, ya sean tuplas o listas, admiten tener elementos de tipos diferentes.

```
> monedas=[("florin","dólar", 100),"€",1.23, [1,2]]
```

Existen otras funciones de clase que hay que explorar: insert, remove, pop, index, count, sort, reverse. Pruébense todas ellas. Para conocer su funcionamiento, basta escribir su nombre seguido de .__doc__

```
> paises_candidatos.insert.__doc__
```

Diccionarios: son como listas indexadas por elementos inmutables (strings o cadenas, números y tuplas) se definen con llaves { }.

```
> primeros_ministros={ } #esta es una manera posible de iniciar una tabla de
                         #cualquier tipo, ya sea una tupla (con( ) ), una lista
                         #(con[ ] ), o un diccionario(con { } )
> primeros_ministros ["Portugal"]="José Sócrates"
> primeros_ministros
> primeros_ministros ["Francia"]=" François Fillon"
> presidentes={"Portugal": "Aníbal Cavaco Silva","Francia":"Nicolas\
  Sarkozy"}
```

Estos índices no numéricos se denominan *keys* y se pueden averiguar como sigue

```
> primeros_ministros.keys() #el output es una lista, ni una tupla ni un diccionario
> primeros_ministros.keys
```

Los valores contenidos en el diccionario se obtienen con

```
> primeros_ministros.values()
```

Otra función de clase interesante es el "has_key"

```
> primeros_ministros.has_key('Francia')
> primeros_ministros.has_key("Francia")
> primeros_ministros.has_key("France")
```

Otra forma sintáctica de definir un diccionario es como sigue

```
> primeras_damas=dict(Portugal="María Alves da Silva",Francia="Carla
  Bruni")        #atención a la ausencia de comillas en las keys en este tipo de
                 #definición de diccionario
> primeras_damas
```

Conjuntos: Son un cuarto tipo de grupos de objetos, que no incluimos en los tres de tablas, ya que tienen un carácter propio: en ellos se pueden realizar operaciones del álgebra de Boole. Se construyen a partir de cualquier tipo de tabla con la orden set(tabla)

```
> conjunto=set([0,1,1j])
> conjunto
> paises_pequeños=(“Liechtestein”, “Andorra”, “Liechtestein”)
> conjunto_paises_pequeños=set(paises_pequeños)
> conjunto_paises_pequeños #se verá que los elementos solamente aparecen una
                           #vez y además ordenados
> set(presidentes)
> set(presidentes.values())
> set(presidentes.keys())
```

Algunas operaciones lógicas son

```
> numeros_primos=set([2,3,5,7,11,13,17,19])
> numeros_impares=set([1,3,5,7,11,13,15,17,19])
> 1 in numeros_primos                    #¿pertenece 1 al conjunto de los
                                         #números primos?
> numeros_primos is numeros_impares      #¿son iguales ambos conjuntos?
> numeros_primos is not numeros_impares  #¿son diferentes ambos conjuntos?
> numeros_primos- numeros_impares        #numeros primos no impares
> numeros_impares- numeros_primos        #numeros impares no primos
> numeros_primos & numeros_impares       #numeros primos e impares
> numeros_primos | numeros_impares       #numeros primos o impares
> numeros_primos ^ numeros_impares       #numeros primos xor impares
```

6. ***Operaciones de control***. Veamos como se formulan en Python las condiciones, los bucles y algunas otras operaciones de control.

   If se apoya en los operadores lógicos: and, or, >, <, = =, ! =, not; la casuística se completa con *elif* y *else*. Después de la condición siempre se escribe una coma.

```
> a=“chaos is not beautiful”
> b=”chaos isn’t beautiful”
> if a = = b :                  # la indentación es muy importante en Python, a

                                # diferencia de C no

      print ‘Falso’             #utiliza llaves y la sintaxis se juzga a partir de la
                                #indentacion, que #es automática en los editores
                                #preparados para el código Python

> b.replace(“isn´t”,”is not”)   #la clase string tiene muchas funciones asociadas
                                #o métodos; para ver un listado de los mismos
                                #basta escribir el  #nombre de una variable
                                #seguido de un punto y pulsar la barra #del
                                #tabulador: aparecerá una lista en la que nos
                                #movemos con #el cursor de arriba y abajo, si
```

```
                                    #utilizáramos el ratón perdemos la lista #y habrá
                                    #que pulsar otra vez la tecla del tabulador
> b
> if a = = b.replace("isn´t","is not"):

        print 'Falso'
```

Los bucles se pueden escribir con

i) while(condicion)

ii) for … in …:

(Ejemplo: for i in [1,2,3]:)

Para construir un rango se pueden utilizar dos comandos: *range* o *orange*

```
> for i in range(5):

        print 2**i-1     #este tipo de números se llaman de Mersenne

> for i in xrange(5):

        print 2**i-1
```

La diferencia es que *range* crea una lista en memoria y *xrange* simplemente recorre los números indicados, de manera que para bucles en los que el índice recorre muchos valores utilizar *range* es un desperdicio de memoria. Sin embargo, *range* tiene la utilidad de servir para inicializar listas

```
> lista=range(20)
> lista
> barrido=xrange(20)
> barrido
> range(10,20)
> range(10,20,3)
```

Otros comandos de interés son

iii) *break* sale del bucle

iv) *continue* hace que se vaya directamente al siguiente paso del bucle sin pasar por las operaciones que quedan del mismo

v) *pass* no hace nada, se utiliza para clarificar la sintaxis. Por ejemplo:

```
while True:

        pass    #espera activamente una interrupción del teclado
                #(Crtl-C)
```

vi) *else* en el caso de utilizarlo en conjunción con un *for* o con un *while*, indica que se agota la lista del *for* o que se incumple la condición del *while*. Por ejemplo:

```
for n in range(2, 10):

  for x in range(2, n):

    if n % x == 0:      #% es la operación modulo(el resto
                        #después de dividir n entre x aquí)

      print n, '=', x, '*', n/x

      break

    else:

                #Se terminó el bucle sin encontrar ningún factor

      print n, 'es primo'
```

vii) Gestión de excepciones *try* y *except*: en el siguiente ejemplo se intenta ejecutar el código entre *try* y *except*; si se produce un error se comprueba si ese tipo de error (que en este caso Python llama ValueError –hay que sabérselo en este caso) está contemplado en una excepción, en cuyo caso se ejecuta el código correspondiente)

```
while True:

    try:

        x = int(raw_input("Introduce un número: "))

        break

    except ValueError:

        print "¡Huy! No es un número. Prueba de \
        nuevo..."
```

Ya veremos casos más complejos de gestión de errores posteriormente.

7. ***Funciones***. Las funciones son operaciones empaquetadas en un bloque que comienza con el nombre de la función anticipado por el comando def y seguido por un paréntesis que incluye todos los parámetros de entrada y un ":"finalmente devuelve un valor de salida con un *return*. Averíguese qué hace el siguiente programa

```
def euclid(numA, numB):

  while numB != 0:
```

```
        numRem = numA % numB

        numA = numb

        numB = numRem

    return numA
```

Por supuesto resulta útil poder escribir una función en un fichero y ejecutarlo desde ahí. Con el IDLE hay que ir a File y seleccionar New window, lo que lanza el editor de un fichero que todavía no hemos nombrado. Escríbase el código en un fichero con la función de arriba y guárdese el fichero con el nombre de div.py.

Para que Python sepa dónde se encuentra este fichero y así pueda ejecutarlo, vamos a cargar un paquete adicional que nos permite informar al *path* de Python dónde buscar los fichero que vamos creando.

```
> import sys      #import es el commando que nos permite cargar paquetes
                  #adicionales además de los que pertenecen a la librería estándar
                  #y que hemos estado usando hasta ahora
> sys.path        #esto nos dice qué directorios están en el path de Python
> sys.path.append("nombre_del_directorio_donde_hemos_guardado_div.py"
> import div      #nuestro fichero es ahora un paquete nuevo creado por nosotros
                  #y así lo llamamos
> div.euclid(24,6)
> euclid(24,6)
> from div import *     #esta alternative a import permite llamar a las funciones
                        #del paquete que importamos  solamente por su
                        #"apellido", sin anteponer el "nombre"
```

8. ***Alguna cosa más sobre las listas.*** Ya se ha comentado que existen operaciones con listas que no hemos ejemplificado: insert, remove, pop, index, count, sort, reverse. Si no se han probado, hágase ahora empezando por pop y append, que por ejemplo sirven para hacer pilas y colas según se detalla a continuación.
    a. *Pilas*. Se amontonan elementos en una lista y se van sacando los últimos. Ejemplo:
    ```
    > pila = [3, 4, 5]
    > pila.append(6)
    > pila.append(7)
    > pila
    > pila.pop()
    > pila
    > pila.pop()
    > pila.pop()
    > pila
    ```
    b. *Colas*. Los elementos de la lista se extraen empezando por los que están colocados en primer lugar.
    ```
    > cola = ["Panda3D", "Blender"]     #búsquese en Internet qué son
                                        #estos programas
    > cola.append("Inkscape")
    ```

```
> cola.pop(0)
> cola.pop(0)
```

Para eliminar elementos de una lista se puede usar la función *del*

```
> a = [-1, 1, 66.25, 333, 333, 1234.5]
> del a[0]
> del a[2:4]
```

9. ***Herramientas de programación funcional***. Hay tres funciones que son muy importantes al tratar con listas: filter(), map() y reduce().
   a. **filter(función,secuencia)**: devuelve una secuencia (del mismo tipo, si es posible) que contiene aquellos elementos de la secuencia de entrada para los que la función es verdadera (Trae)

```
> def f(x): return x % 2 != 0 and x % 3 != 0
> filter(f, range(2, 25))
```

   b. **map(función, secuencia)**: aplica la función a cada uno de los elementos de la secuencia

```
> def cubo(x): return x*x*x
> map(cubo, range(1, 11))
```

   Si la función toma más de un argumento, se introducen tantas secuencias como el número de los mismos

```
> def ecuacion2g(a,b,c):

      import numpy

      d=numpy.sqrt(complex(b**2-4*a*c))

      return [(-b+d)/(2*a),(-b-d)/(2*a)]

> ecuacion2g(1,1,1)
> map(ecuacion2g,(1,2),(1,3),(1,4))
> map(ecuacion2g,[1,2],[1,3],[1,4])
```

   c. **reduce(función, secuencia)**: aplica la función, que ha de tomar dos argumentos, a los dos primeros elementos de la secuencia y el resultado de la misma se toma como primer parámetro de una segunda aplicación añadiendo el tercer elemento de la secuencia, etc. Es decir, da como resultado la siguiente expresión:

      función(…función(función(función(secuencia[0],secuencia[1]),secuencia[2],secuencia[3])…,secuencia[n-1])

```
> def sumainv(x,y): return 1/(x+y)
> sumainv(1,2)
```

```
> sumainv(1.,2,)
> def sumainv(x,y): return 1/float(x+y)
> sumainv(1,2)
> reduce(sumainv,range(10))
> def suma(x,y): return x+y
> reduce(suma,range(10))          #suma los diez primeros números
```

Si sólo hay un elemento en la secuencia, se devuelve su valor. Si la lista está vacía se devuelve un error. Para evitar este último caso, se puede añadir un tercer argumento que actúa como valor inicial.

```
> reduce(suma,(),0)
```

10. ***Listas autodefinidas***. Aunque nosotros no estamos teniendo en cuenta los cambios introducidos en la versión 3.0, a la espera de que se consoliden, y nos basamos en el estándar 2.5, sí conviene decir que las funciones filter(), map() y reduce() han desaparecido ya que se pueden sustituir adecuadamente –también en Python 2.5- por las listas autodefinidas. Sin embargo, las hemos presentado porque son conceptos en sí mismos, y no solamente funciones, y conviene pensar en términos de ellos a veces. Constan de una expresión seguida de un bucle *for*

```
> aminoacido=[‘ grupo carboxilo’,’ grupo amino’,\
                ‘hidrogeno ‘,’ radical’]
> [elemento.strip() for elemento in aminoacido]
> vec=[1,2,3]
> [3*x for x in vec]
> [3*x for x in vec if x < 2]
> [[x,x**2] for x in vec]
> [x, x**2 for x in vec]
> vec1 = [2, 4, 6]
> vec2 = [4, 3, -9]
> [x*y for x in vec1 for y in vec2]
> [vec1[i]*vec2[i] for i in range(len(vec1))]
> [str(round(355/113.0, i)) for i in range(1,6)]
```

11. ***Comentarios sobre técnicas para hacer bucles***. Al recorrer diccionarios, es posible recuperar la clave y su valor correspondiente a la vez, utilizando el método *iteritems()*

```
> caballeros = {’gallahad’: ’el casto’, ’robin’: ’el valeroso’}
> for k, v in caballeros.iteritems():
```

```
                print k,v
```

Al recorrer una secuencia, se pueden recuperar a la vez el índice de posición y su valor correspondiente usando la función *enumerate()*

```
> for i, v in enumerate(['pim', 'pam', 'pum']):
                print i, v
```

Para recorrer dos o más secuencias en paralelo, se pueden emparejar los valores con la función *zip()*

```
> preguntas = ['nombre', 'misión', 'color favorito']
> respuestas = ['lanzarote', 'el santo grial', 'azul']
> for p, r in zip(preguntas, respuestas):
                print '¿Cuál es tu %s? %s.' % (p, r)
```

Para recorrer una secuencia en orden inverso, hay que especificar primero la secuencia en el orden original y llamar a la función *reversed()*

```
> for i in reversed(xrange(1,10,2)):
                print i
```

Existe un método asociado a toda lista que se denomina *reverse* y que tiene un efecto semejante sobre la lista, pero esta vez sobre la lista misma y no sobre el output

```
> lipidos=['fosfolipidos','glucolipidos']
> reversed(lipidos)
> print reversed(lipidos)
> print lipidos
> lipidos.reverse()
> print lipidos
```

Con la función *sorted()* se hace algo semejante a *reversed()*, pero se produce un ordenamiento en lugar de una inversión. Hágase una prueba.

12. ***Módulos***. Un módulo es un fichero. La manera de incluir la información que contiene sobre funciones y clases (que luego veremos) es utilizando la función *import* de la librería estándar, que ya hemos visto.

```
> import astrología        #astrología sería el nombre de un fichero
                           #que se encuentra en el path del sistema;
                           #aquí obtendremos un error porque dicho
                           #fichero no existe
```

Para conocer qué contiene ese módulo utilizamos la función *dir*

```
> import sys
> dir(sys)
> dir(os)          #si intentamos hacer el dir de un módulo o
                   #de un paquete antes de importarlo se
                   #obtendrá un mensaje de error
> import os
> dir(os)
```

Se pueden importar varios módulos o pauquetes (ahora veremos qué son estos) con un solo import

```
> import numpy,scipy
```

Cada vez que usemos una función o una clase de un módulo hay que anteponerle el nombre de este

```
> sin(pi)
> numpy.sin(numpy.pi)
```

Ya que esto puede resultar bastante incómodo, se suele utilizar el formato *from…import* *, que permite importar los componentes al nivel de jerarquía de llamada superior

```
> from numpy import *
> sin(pi)
```

Un paquete no es más que un conjunto de ficheros y directorios que contienen código Python. Se trata ahora de trabajar con una estructura más compleja y organizada de la que proporciona un solo fichero. Para crear un paquete en un directorio que llamamos, por ejemplo, *sound*, creamos un directorio con ese nombre que contenga un fichero que se llame *__init__.py* (distíngase el doble subrayado, que es el de este y otros muchos casos, del subrayado simple). Este fichero puede estar vacío, pero permite que Python identifique este directorio como soporte de un paquete. Dentro de este directorio creamos, por ejemplo, tres subdirectorios: *formats*, *effects* y *filters*. Cada uno tendrá dentro un fichero *__init__.py*, ya sea vacío o con alguna información de variables o de documentación, por ejemplo. Por ejemplo, créese esta estructura como ejercicio, y con un fichero adicional en cada subdirectorio: waveread.py en formats, echo.py en effects y equalizer.py en filters e introdúzcase alguna función elemental en cada uno, que actúe sobre cadenas alfanuméricas, en lugar de sobre ficheros .mp3, por ejemplo, ya que de momento no sabemos cómo tratarlos. Para que se pueda hacer un *from sound import* *, hay que añadir una variable *__all__* con todos aquellos nombres de las variable, funciones y clases que se quieren incluir en un *from … import* *. Por ejemplo

```
__all__=[“echo”, “surround”, “inverse”]
```

Para saber dónde está ubicado un paquete se puede usar la variable __path__, que se crea automáticamente al importar un paquete

```
> sound.__path__
```

Recuérdese que para que Python pueda encontrar el paquete sound el directorio madre de este directorio ha de estar en el path (recuérdese la orden sys.path.append("…")).

Vamos a clarificar ahora el uso de *dir*. Reiníciese la shell del IDLE yendo a Shell en el menú y seleccionando Restart Shell y luego introducir lo siguiente

```
> dir()
> import sys
> dir()
> import numpy
> dir()
> from numpy import *
> dir()
> dir(sys)
> dir(numpy)
```

Las funciones que no necesitan de un import están en __builtins__

```
> dir(__builtins__)
```

Compárese ahora el output de

```
> dir(__name__)
```

con las funciones cuando escribimos

```
> S="alfalfa"
> S.
```

y apretamos seguidamente la tecla del tabulador (para moverse dentro de la ventana emergente úsense los cursores ya que si usamos el ratón perdemos dicha ventana emergente)

13. ***Clases y objetos***. Una clase es la definición de un tipo de objeto. ¿Y qué es un objeto? Un objeto es una entidad de información que contiene *atributos*, que son básicamente de dos tipos: *variables miembro* o datos y *métodos* o funciones. Si empezásemos definiendo clase como una estructura de información, es decir, como un tipo de entidad de información, sin mencionar la palabra objeto, diríamos que un objeto es una *instancia* (un ejemplo o caso concreto) de una clase. Así, *hombre* sería una clase y *Pepe* un objeto.

Creemos nuestra primera clase

```
> class Biblioteca:

      "Una biblioteca es un almacén de libros y revistas"

      n_ejemplares=3

      ejemplares=["Quijote","Amadis","Hamlet"]

      def add(self,libro_nuevo):

            self.ejemplares.append(libro_nuevo)
```

Fíjese que para referirse a sí misma se utiliza el término *self* en la definición. Ahora se trata de crear un objeto

```
> mibiblioteca=Biblioteca()
```

La presencia de los paréntesis es imprescindible para la *instanciación* de la clase en un objeto. Si no se utilizasen

```
> mibiblioteca=Biblioteca
```

estaríamos diciendo que mibiblioteca es una clase idéntica a Biblioteca, pero no un objeto. Para llevar mejor la cuenta de qué es una clase y qué es un objeto se suele adoptar el convenio de que una clase se escribe con la primera letra en mayúsculas. Veamos esto de que mibiblioteca=Biblioteca define una clase y no un objeto

```
> mibiblioteca=Biblioteca()
> mibiblioteca2=Biblioteca
> mibiblioteca.add("Biblia")
> mibiblioteca2.add("Biblia")
```

Efectivamente, esta última orden da error porque no se puede utilizar la clase como un objeto, es decir, *add* define un método dentro de la clase pero no es un método en sí mismo, ni mibiblioteca2 es un objeto (un "self").

```
> mibiblioteca.ejemplares
```

De todas maneras, está clase está mal definida, ¿por qué? (si no se sabe introdúzcase mibiblioteca.n_ejemplares). Corríjase la definición de clase de manera que las variables miembro sean consistentes).

Nótese que lo que hemos dicho arriba es que toda biblioteca ha de tener el Quijote, el Amadís y Hamlet para ser considerada biblioteca ;-)

Otra cosa que conviene aclarar es el uso del self en la definición de los métodos. Siempre conviene ponerlo en la definición de los mismos incluso si no se necesita al objeto mismo dentro del método. Obsérvense los siguientes ejemplos

```
> class MiClase:
        def f(self):
            return "hola"
```

Esta sería la manera correcta

```
> ejemplo_tonto_de_objeto=MiClase()
> ejemplo_tonto_de_objeto.f()
> ejemplo_tonto_de_objeto.f(1)
```

Esta última orden da un error ya que el self no cuenta como argumento de la función miembro o método. Definamos ahora la función sin el self

```
> class MiClase:
        def f():
            return "hola"
> ejemplo_tonto_de_objeto=MiClase()
> ejemplo_tonto_de_objeto.f()
```

Sin embargo, sería posible poner

```
> class MiClase:
        def f(x):
            return "hola"
> ejemplo_tonto_de_objeto=MiClase()
> ejemplo_tonto_de_objeto.f()
```

Lo que está ocurriendo es que Python se cree que *x* es *self*, ya que él no habla inglés e interpreta cada cosa dependiendo de su posición

```
> class MiClase:
        n=4
        def f(x):
            x.n=5
> ejemplo_tonto_de_objeto=MiClase()
> ejemplo_tonto_de_objeto.n
```

```
> ejemplo_tonto_de_objeto.f()
> ejemplo_tonto_de_objeto.n
```

Sin embargo, el estilo correcto es usar el *self*, si no se hace la cosa se complica para tener una buena interpretación de lo que pasa. Por ejemplo, estaríamos tentados de poner

```
> ejemplo_tonto_de_objeto.f(1)
```

que no funciona porque el *x* es el *self* y el self no cuenta como argumento, como hemos dicho más arriba.

La explicación "Una biblioteca es un almacén de libros y revistas", sobre la que no hemos dicho nada es lo que obtenemos si ponemos

```
> mibiblioteca.__doc__
```

Resulta obvio recomendar el uso de

```
> print mibiblioteca.__doc__
```

para ver bien todos los caracteres.

Este mismo principio para crear el __doc__ de una clase vale para crear el __doc__ de una función.

Muchas veces resulta útil que exista una inicialización de los atributos de los objetos desde que estos se crean. Por ejemplo, si ponemos

```
> class Complejo:
        def __init__(self,parteReal,parteImaginaria):
            self.r= parteReal
            self.i= parteImaginaria
> z=Complejo(1,2)
> z.r
> z.i
```

El inconveniente ahora es que si ponemos

```
> z=Complejo()
```

nos da error. Por tanto, conviene definir unos valores por defecto como sigue

```
> class Complejo:
        def __init__(self,parteReal=0,parteImaginaria=0):
            self.r= parteReal
            self.i= parteImaginaria
> z2=Complejo()
> z2.r
> z2.i
```

De hecho, esta definición por defecto es algo que no habíamos visto cuando aprendimos a usar las funciones pero que también existe para las funciones que no están dentro de una clase.

```
> def suma(x,y=0,z=0):
        """Suma aritmética"""
        return x+2*y+3*z
> suma(1,2,3)
> suma(1)
> suma(1,2)
```

Otro aspecto interesante es que aquellos argumentos que tienen un valor por defecto pueden llamarse a través del nombre de dicho argumento en la definición

```
> suma(1,y=2)
> suma(1,y=2,z=3)
> suma(1,z=3,y=2)
```

Se puede también utilizar una lista como lista de argumentos

```
> def suma2(*args):
        """Suma aritmética"""
        return reduce(suma,args)
```

Pruébense ahora los dos comandos siguientes

```
> suma2([1,2,3,4])
> suma2(*[1,2,3,4])
```

Uno estaría tentado de escribir

```
> def suma3(args):

      """Suma aritmética"""

      return reduce(suma,args)
```

como equivalente de suma2. En este ejemplo no hay problema, excepto que ahora no hay que poner el asterisco al introducir los parámetros

```
> suma3([1,2,3,4])
> suma3(*[1,2,3,4])
```

Sin embargo, es mejor trabajar con el asterisco ya que este es el que permite desempaquetar los argumentos, cosa que no es necesaria en el caso de suma2, ya que el input de *reduce* puede ser una lista. Para ilustrar esto medítese lentamente y en silencio sobre los siguientes comandos

```
> range(3,6)
> args=[3,6]
> range(*args)
> range(args)
```

Volviendo a las clases, se dice normalmente que incorporan tres conceptos muy importantes: polimorfismo, encapsulación y herencia.

Polimorfismo. Es la capacidad de utilizar la misma función sobre argumentos diferentes. Esta idea es ya implementable en una función normal

```
> def pregunta(tabla):

      if isinstance(tabla,tuple):

            print str(tabla)+" es una tupla"

      elif isinstance(tabla,list):

            print str(tabla)+" es una lista"

      elif isinstance(tabla,dict):

            print str(tabla)+" es un diccionario"

      else:

            print str(tabla)+" no es una tabla"
```

Este concepto de polimorfismo en el contexto de las clases permite definir diferentes métodos con el mismo nombre que se aplican a diferentes tipos de variables. Sin embargo, en Python, cuando nosotros programamos, no tenemos que especificar el tipo de variable en la línea de argumentos, de manera que el polimorfismo es algo más bien interno en Python, no algo que usemos al esculpir nuestro código. Por ejemplo "+" es polimórfico en Python en la medida que

```
> 1+2
> "hola"+" que tal?"
```

están ambos definidos.

Encapsulación. Es la posibilidad de incluir en una clase variables y funciones miembro de uso interno, pero que no pueden ser llamadas desde fuera

```
> class Secretive:
        a=2
        __b=3
        def __inaccessible(self):
            print "Bet you can't see me..."
        def accessible(self):
            print "The secret message is:"
            self.__inaccessible()
> a=Secretive()
> a. __inaccessible()
> a. accessible()
> a.a
> a.__b
```

Herencia. Es la capacidad de construir clases a partir de clases, heredando los atributos de aquellas y posiblemente incluyendo otros específicos. Esto se hace poniendo entre paréntesis después del nombre de la clase el nombre de aquella o aquellas clases madre.

```
class nombreClaseDerivada(nombreClaseBase):
    <sentencias>
```

Constrúyase una subclase de la clase biblioteca que además tenga revistas.

Investíguese el funcionamiento de los comandos *issubclass*, .*__bases__* y .*__class__*

14. ***Advertencia: comando reload***. Una vez que hemos importando correctamente un módulo o un paquete, este queda almacenado en memoria. Esto implica pata Python que si hacemos un cambio en el fichero o ficheros que lo componen y volvemos a hacer un import del mismo, estos cambios no se incorporan, ya que Python mira si tiene ese módulo o paquete en memoria y si es así no hace nada, de manera que nuestros cambios no quedan registrados. Para recargar un paquete hay que usar el comando *reload(nombre_del_paquete_o_módulo)*.

15. ***Comando execfile***. Hay una serie de operaciones que realizamos rutinariamente cada vez que arrancamos el Python: carga de paquetes habituales como el sys, el os, el numpy, el scipy o el matplotlib, adición de nuestra(s) carpeta(s) de programas al path. Todas estas órdenes se pueden empaquetar en un *batch file* que podemos ejecutar cada vez que empezamos una sesión Python. Este batch file ha de estar en una carpeta que esté en el path nada más arrancar, típicamente la …\Python25 (C:\Python25, por ejemplo) y puede tener cualquier nombre: no necesita acabar en .py. Un ejemplo sería un fichero llamado startup cuyo contenido fuese

    ```
    import sys,os,numpy,scipy

    sys.path.append("D:\\misprogramas\\favoritos\\Python")
    ```

    y que ejecutaríamos nada más arrancar la terminal de Python con

    ```
    > execfile("startup")
    ```

16. ***Más sobre los módulos y clases: los "espacios nominales"***. Un espacio nominal es una correspondencia o "mapping" entre nombres y objetos. Y un ámbito es la región donde se pueden realizar esas correspondencias. Por ejemplo, *also* significa también en inglés y *por tanto* en alemán. Es decir, que cada idioma sería como un espacio nominal y cada uno de los países correspondientes sería un ámbito. La mayor parte de los espacios nominales se implementan como diccionarios internamente, aunque eso no lo vemos nosotros. Sin embargo, tiene sentido con el ejemplo que hemos puesto de los idiomas, los diccionarios serían los diccionarios que relacionan nombre y objeto (nombre en inglés y realidad física o concepto, en nuestro ejemplo; ojo con la idea: no nombre en inglés y nombre en español, ja, ja). Ejemplos de espacios nominales ya los hemos visto: cuando hacíamos

    ```
    > dir()
    ```

    veíamos que los llamados nombres internos (que tienen una correspondencia en el llamado ámbito interno o ámbito universal, algo así como las señales de tráfico que son iguales en todos los países) se encuentran en __builtins__

    ```
    > dir(__builtins__)
    ```

    que contiene, entre otras, la función *abs*

    ```
    > abs(-4)
    ```

    Otro ejemplo lo hemos visto en nuestra creación de paquetes o de clases. En el capítulo de paquetes y módulos creamos una carpeta que se llamaba *sound* y que gracias a la presencia de un fichero __init__.py quedaba caracterizada como paquete. Una vez

importado ese paquete es posible usar el "vocabulario" de ese módulo, siempre y cuando nos refiramos a los ficheros por su nombre completo, que es como decir que identifiquemos el espacio nominal o "idioma". Así pues si queremos usar una función que esté en un fichero *–op_basicas.py*, por ejemplo- que está dentro de la carpeta sound, tenemos que importar también el fichero con

```
> import sound.op_basicas
```

donde vemos que no hemos puesto solamente

```
> import op_basicas
```

porque op_basicas pertenece al ámbito de sound y no se entiende fuera de él. Si dentro de este fichero tenemos una función *set_vol* (def set_vol(value)) por ejemplo, para poder usarla ahora tendríamos que poner

```
> sound.op_basicas.set_vol(3)
```

Como vemos los import nos permiten cargar el espacio nominal en la memoria de Python (recuérdese el método de aprendizaje de los personajes de The Matrix) pero no que los podamos llamar sin anteponer el nombre de su espacio nominal *sound.op_basicas* (en nuestro ejemplo idiomático es como idioma.dialecto: castellano.canario.gua_gua), ya que si así lo hiciésemos podría tratarse de una función o en general un nombre de algo que tiene significados diferentes en ámbitos diferentes. Por eso Python no tolera este uso del lenguaje y no reconoce los nombres de las cosas si no especificamos su ámbito (no se pone a buscar en los espacios nominales a ver si hay algo que se llama así).

Sin embargo, es posible utilizar un *from* <a> *import* <b>, como hemos visto. Para ello hay que definir una variable __all__=["nombre_fichero1","nombre_fichero2", …,"nombre_carpeta1","nombre_carpeta2",…] en el __init__.py, de manera que cuando hacemos un import del paquete que tiene este __all__ en su __init__.py, esta variable __all__ se inicializa con esos valores y hace que cuando demos al tabulador después del nombre del paquete ya importado seguido de un punto nos aparezcan los nombres de esos módulos (=ficheros) o subpaquetes (=carpetas). Pero nos permite algo más, y es lo que decíamos al principio de este párrafo: nos permite usar un *from* nombre_del_paquete *import* *, donde * se interpreta como la lista __all__. ¿Qué efecto tiene esto? Son dos efectos diferentes, teniendo en cuenta de si se trata de un módulo (=fichero) o subpaquete(=carpeta). Si se trata de un módulo, una vez que hemos hecho el *from* nombre_del_paquete *import* * se puede usar ese módulo sin anteponer el nombre del espacio nominal, es decir, en nuestro caso de sound, podríamos usar *op_basicas.set_vol(3)* sin necesidad de poner *sound.* delante. Sin embargo, sí hay que poner *op_basicas.* delante de *vol(3)*. Esto se debe a que hemos importado op_basicas, pero no hemos eliminado la necesidad de anteponer *op_basicas.*, solamente la de anteponer *sound.*. Para eliminar esta necesidad deberemos poner *from sound.op_basicas import* * (!), y no *from op_basicas import* *, como quizás esperábamos. Esto se debe a que lo que hemos importado el espacio nominal pero no el ámbito (este es un punto extremadamente sutil que podemos ver si introducimos las siguientes líneas después de reiniciar la shell

```
> pi
> from numpy import *
> pi
```

```
> numpy
> import numpy
> numpy
```

Es decir, *que from numpy import ** significa *from ambito_de_numpy import todo_lo_que esta_en_el_espacio_nominal_de_numpy*.

Otra pregunta que nos podríamos hacer ahora es la siguiente: ya que después de poner from numpy import *, podemos acceder a *pi* directamente sin poner *numpy.pi* (es decir, que hemos importado *pi* de *numpy*, aunque no *numpy* misma hasta que hacemos *import numpy*, como queda reflejado arriba), ¿dónde está definido *pi*? Pues pi tiene que estar en __init__.py, y dentro de __init__.py en al menos dos sitios: en su lugar de definición y al principio de todo en el fichero dentro de la definición de __all__

```
__all__=[…,”pi”,…]
```

Sin embargo, si vamos a la carpeta de numpy haciendo

```
> import numpy
> numpy.__path__
```

y mirando dentro del __init__.py que hay en esa carpeta veremos que esto no aparece tan claro. Eso se debe a que la implementación de numpy, al ser más compleja, es más opaca: hay referencias a otros paquetes dentro de __init__.py que a su vez cargan más definiciones y además muchas funciones y elementos están compilados para ganar velocidad, según lo que dijimos en la presentación introductoria. No obstante, nosotros, escribiríamos así nuestra versión de numpy o los paquetes que hagamos: añadiendo variables y funciones fundamentales en el __init__.py.

17. ***Ficheros y directorios***. La forma más sencilla de abrir ficheros y de escribir en ellos es como sigue. Abramos por ejemplo un fichero para escritura

```
> file=open(‘dale_tu_un_path_y_un_nombre’,’w’)
> print f
```

El modo puede ser ‘w’, de escritura, ‘r’, de lectura, o ‘a’, también de escritura pero sin sobreescribir lo que había antes, sino añadiendo nuevo material al fichero. La escritura y lectura binarias se consiguen con ‘wb’,’rb’ y ‘ab’, para los casos recién especificados para ASCII. Si ponemos ‘r+’, habremos abierto en modo lectura + escritura y ambas operaciones serán posibles. Completa este ejercicio inicial añadiendo algo de texto al fichero a través del método write del objeto file

```
> file.write(“escribe tú algo”)
```

Ahora cerramos el fichero en escritura,

```
> file.close()
```

lo abrimos con un editor y escribimos más líneas de prueba o bien utilizamos un ‘a’ con Python y open, lo volvemos a abrir en lectura y lo leemos como sigue

```
> file=open('dale_tu_un_path_y_un_nombre','r')
> file.read()
> file.close()
> file=open('dale_tu_un_path_y_un_nombre','r')
> file.readline()
> file.readlines()
> file.close()
```

Otra vez

```
> file=open('dale_tu_un_path_y_un_nombre','r')
> file.read(4)
> file.read(3)
> file.read(1)
> file.seek(5)
> file.read(2)
```

file.seek admite también un segundo argumento que por defecto es 0 e indica a partir de dónde se cuentan los bytes para situar el lector o cabeza de escritura del fichero. Así, este segundo argumento puede ser 0 (=principio del fichero), 1 (=punto actual donde se había quedado el lector o la cabeza de escritura) o 2 (=final del fichero). Practíquese al gusto con estas posibilidades.

18. ***Módulo pickle.*** Este módulo permite hacer un *estibado* o volcado de todo un objeto Puython en un fichero, de manera que podemos ir guardando resultados intermedios de un programa para que si lo detenemos podamos recuperarlo posteriormente y podamos reanudar nuestros cálculos posteriormente. Pruébese a construir una clase Mibiblioteca como la de antes y un cierto objeto de la misma, que llamaremos *mirincon*, por ejemplo. Sobre ella realizamos la siguiente operación

```
> import pickle
> fichero=open(nombre_de_un_path+fichero_entre_comillas,'w')
> pickle.dump(mirincon, nombre_de_un_path+fichero_entre_comillas)
> fichero.close()
```

La manera de recuperarlo será escribiendo posteriormente

```
> fichero=open(nombre_de_un_path+fichero_entre_comillas,'w')
> mirinconrecuperado=pickle.load(nombre_del_fichero)
> fichero.close()
```

Como se ve, ni pickle.dump ni pickle.load abren o cierran ficheros. Simplemente guardan objetos, potencialmente muy complejos aunque también pueden ser tan sencillos como una string o cadena (una string es un objeto de la clase String), de una manera reconocible para Python.

Pruébese ahora a ver qué se ha grabado en el fichero abriéndolo con un editor.

19. ***Numpy.*** Numpy nos permite realizar operaciones matemáticas, que es algo que de momento no hemos hecho realmente excepto en algún ejemplo donde precisamente tuvimos que importarlo.

    Lo primero es ver la importancia de los arrays, porque lo que hemos visto hasta ahora son listas y tuplos, cuya operación matemática no es numérica

```
>  a=[1,2,3]
>  a*3
>  a+4
>  from numpy import *
>  a = array( [ 10, 20, 30, 40 ] )
>  a
>  b = arange( 4 )
>  b
>  c = linspace(-pi,pi,3)
>  c
>  d = a+b**2
>  x = ones( (3,4) )
>  x
>  z= zeros( (3,4) )
>  z
>  y = arange(12)
>  y
>  y.shape=3,4
>  y
>  3*a
>  a+y
>  a[2:4] = -7,-3
>  for i in a:
        print i
>  x = ones( (3,4) )
>  x[1,2] = 20
>  x
>  a = arange(10).reshape(2,5)
>  a
```

Averígüese cómo funcionan y practíquese con los métodos .ndim, .shape, .size, .dtype, .itemsize y .data.

Hay muchas maneras de crear arrays. Nosotros hemos visto alguna arriba. Podemos hacerlo también a partir de una secuencia

```
> a = array( [2,3,4] )
> a
> type(a)
> a = array(1,2,3,4)   # WRONG
```

Si queremos crear un array vacío podemos usar empty

```
> z=empty( (2,3) )
> z
```

Investíguense las diferencias entre arange y linspace

```
> arange( 0, 2, 0.3 )
> linspace( 0, 2, 9 )
```

El uso de las funciones en numpy es como estamos acostumbrados

```
> x = linspace( 0, 2*pi, 100 )
> y=sin(x)
```

Veamos ahora como juntar vectores para formar matrices

```
> a = floor(10*random.random((2,2)))
> a
> b = floor(10*random.random((2,2)))
> b
> vstack((a,b))
> hstack((a,b))
```

También podemos descomponer una matriz en filas o columnas de varias maneras. Una de ellas es

```
> a = floor(10*random.random((2,12)))
> a
> hsplit(a,3)
> hsplit(a,(3,4))
```

Investíguese el funcionamiento de la función ix_()

```
> a = array([2,3,4,5]
> b = array([8,5,4])
> c = array([5,4,6,8,3])
> ax,bx,cx = ix_(a,b,c)
> result = ax+bx*cx
> result[3,2,4]
> a[3]+b[2]*c[4]
```

Veamos ahora algo de álgebra matricial

```
> from numpy import matrix
> from numpy import linalg
> A = matrix( [[1,2,3],[11,12,13],[21,22,23]])
> x = matrix( [[1],[2],[3]] )
> y = matrix( [[1,2,3]] )
> print A.T
> print A*x
> print A.I
> print linalg.solve(A, x)
> p
```

Para más información visítese la página

http://docs.scipy.org/doc/numpy/reference/

Por ejemplo, mírese cómo jugar con las FFTs.

20. ***Matplotlib***. Matplotlib es una de las librerías gráficas disponibles para Python. Nosotros nos vamos a centrar en un conjunto de comandos disponibles dentro de la sublibrería matplotlib.pylab, que siguen el patrón de las posibilidades gráficas de Matlab.

```
> import matplotlib.pyplot as plt
> plt.plot([1,2,3])
> plt.ylabel('some numbers')
> plt.show()
```

Para cerrar la ventana gráfica utilícese plt.close(). Otro ejemplo:

```
> mu, sigma = 2, 0.5
> v = random.normal(mu,sigma,10000)
> plt.hist(v, bins=50, normed=1)
```

```
> plt.show()
> (n, bins) = histogram(v, bins=50, normed=1)
> plt.plot(bins, n)
> plt.show()
```

Para más ejemplos visítese la página

http://matplotlib.sourceforge.net/users/pyplot_tutorial.html