chap04

July 28, 2020

1 Capítulo 4

1.1 Funciones del usuario

A medida que la longitud y complejidad de los programas de computador aumentan, se hace necesario dividir el problema en pequeñas partes (**divide and conquer**). Esto es una buena estrategia, ya que lo hace más modular y más fácil de leer y entender (y de encontrar problemas o *bugs*). Esto se puede hacer creando funciones (`def` o definiciones) dentro de Python, que hacen parte del trabajo. Algunas ventajas de programar así: - Se puede evaluar partes o pedazos del código de manera individual, confirmar que están funcionando correctamente antes de terminar el programa completo. - El código es más modular y fácil de entender. - Es más fácil usar partes del programa en otros programas (sin necesidad de copiar y pegar cada pedazo).

1.2 Funciones dentro del programa

Hazlo tu mismo El código de abajo muestra una función (o `def`) que tiene como objetivo

[8]:
```
# adivine_entero2.py
# Juego donde el usuario adivina un número,
# usando funciones
#

def guess_number(number):
    """ Función para adivinar un número
    Función del usuario que hace el trabajo de
    interactuar con el usuario y evaluar si adivinó el número.
    Entrada: number - un número, que el usuario no conoce

    Salida: guesses - Número de intentos
    """
    guesses  = 1
    guess    = int(input('Adivine un número del 1 al 1000: '))
    while guess!=number:
        guesses = guesses + 1
        if (guess>number):
            print(guess," es muy alto")
        elif (guess<number):
            print(guess," es muy bajo")
        guess = int(input("Adivine otra vez: "))
```

```
    return guesses

#-------------------------------------
# El programa principal
#-------------------------------------

import numpy as np

# Obtenga número aleatorio [1, 1000)
rnum = np.random.randint(1, 1000)

# Llame a la funci´øn que interactua con usuario
nguess = guess_number(rnum)

print("\nExcelente, adivinaste en ",nguess, ' intentos')
```

```
Adivine un número del 1 al 1000: 0
0  es muy bajo
Adivine otra vez: 500
500  es muy bajo
Adivine otra vez: 750
750  es muy bajo
Adivine otra vez: 875
875  es muy bajo
Adivine otra vez: 925
925  es muy alto
Adivine otra vez: 900
900  es muy alto
Adivine otra vez: 880
880  es muy bajo
Adivine otra vez: 890
890  es muy alto
Adivine otra vez: 885
885  es muy alto
Adivine otra vez: 883
883  es muy alto
Adivine otra vez: 882

Excelente, adivinaste en  11  intentos
```

La función que creamos se llama `guess_number`, y se debe definir al principio del programa (en Notebooks, antes de ser usada, por ejemplo en una celda anterior). La función tiene la estructura:

```
def guess_number(number):
    ...
    return guesses
```

La variable de entrada `number`, funciona como un argumento de la función `guess_number`, y el

resultado de la función se dá con la variables `guesses`. Note que debe usar `return guesses` para que el programa principal sepa qué obtiene de vuelta al llamar la función.

La función se comporte igual a cualquier función que ya hemos usado,

```
asen = sin(theta)
```

y note que el nombre de la variable (`number`) que le pongamos al resultado de llamar a la función **no** tiene que ser el mismo que usa la función internamente.

Indentación sigue siendo importante Note que dentro de la función se requiere mantener la indentación.

Comentarios de la función Siempre es buena idea documentar de manera completa cualquier función que uno cree. Esto con el fin de que sea claro lo que hace, las entradas y salidas y así, cualquier persona que lea el código pueda entenderlo. Note las triples commillas, que abren y cierran las funciones. Si uno quiere ver la descripción de la función:

[9]:
```
print (guess_number.__doc__)
```

```
 Función para adivinar un número
    Función del usuario que hace el trabajo de
    interactuar con el usuario y evaluar si adivinó el número.
    Entrada: number - un número, que el usuario no conoce

    Salida: guesses - Número de intentos
```

El programa principal llama a la función a través del comando

```
nguess = guess_number(rnum)
```

La variable `number` dentro de la función tomará el valor que determina el programa al llamar a la función (`rnum`}. Tenga en cuenta sin embargo, que a diferencia de Fortran, si las variables son cambiadas dentro de la función, éstas no cambian en el programa principal. Hay que tener cuidado con esto. Note que el nombre de la variable (de entrada y salida) en los argumentos de la función no tienen que ser iguales (`rnum` - `number`).

Nota

Tenga en cuenta que a diferencia de lo que pasa en otros languajes, si la variable de entrada es cambiada dentro de la función (rnum por ejemplo), esta variable NO cambia en el programa principal (en Fortran esto si pasaba).

Hazlo tu mismo El ejercicio del máximo común divisor (MCD) se puede separar en dos partes. La primera, encargada de la interacción con el usuario (pedirle los números, etc). La segunda, calcular el MCD. La última la podemos poner cpomo una función.

[3]:
```
# gcf2.py
# encuentre el máximo común divisor de dos números enteros
# con una función
#
```

```
def gcf(a,b):
    amin = min(a,b)
    for j in range(1,amin+1):
        if (a%j==0 and b%j==0):
            jmax = j
    return jmax

# Ahora, el programa principal

for i in range(10):
    intxt = input('Digite dos números enteros (ceros para parar) ')
    a,b   = intxt.split()
    a     = int(a)
    b     = int(b)
    if (a==0 and b==0):
        break
    mcd = gcf(a,b)  # cree esta función

    print ("Máximo común divisor = ", mcd)
```

```
Digite dos números enteros (ceros para parar) 2 5
Máximo común divisor =  1
Digite dos números enteros (ceros para parar) 2 8
Máximo común divisor =  2
Digite dos números enteros (ceros para parar) 0 0
```

Note que en este caso, se define la función para calcular el MCD, y recibe dos números de entrada (`x,y`) y devuelve al programa la variable `z` que es el MCD.

```
def getgcf(x,y):
   ...
   return z
```

El número y posición de los argumentos debe coincidir con los de la función, sin embargo note que los nombres de las variables no tienen que ser iguales (`x` = `a`).

1.3 Funciones con múltiples salidas

Las funciones descritas abajo tienen una utilidad limitada, ya que están diseñadas para rgresar al programa una sola variable como resultado. En algunos casos, se requiere de funciones que puedan regresar varias variables como rsulta- do de una función. En Fortran esto se lleva a cabo con subrutinas (subroutine). En Python, la misma función puede retornar varias variables de resultado. En Python, la forma como se devuelven dichas variables puede variar, incluyendo *listas*, *tuples*, etc., pero para facilitar en este curso vamos a utilizar *tuples*.

1.3.1 Distancias en la Tierra

A continuación un ejemplo de una función en Python con utilidad en Geociencias para calcular la distancia y azimuth de dos puntos sobre la superficie de la Tierra.

[1]:
```
def sph_azi(flat1,flon1,flat2,flon2):
    """
    SPH_AZI computes distance and azimuth between two points
    on the sphere

    Inputs: flat1 = latitude of first point (degrees)
            flon2 = longitude of first point (degrees)
            flat2 = latitude of second point (degrees)
            flon2 = longitude of second point (degrees)

    Returns: del = angular separation between points (degrees)
             azi = azimuth at 1st point to 2nd point, from N (deg.)

    Notes:
    (1) applies to geocentric not geographic lat,lon on Earth

    (2) This routine is accurate depending on the precision of the
    real numbers used. Python should be accurate to real(8) precision
    For greater accuracy, perform a separate calculation for close
    ranges using Cartesian geometry.
    """

    # importe modulos necesarios
    import math  as mt
    import numpy as np

    if ( (flat1 == flat2 and flon1 == flon2)
    or (flat1 == 90. and flat2 == 90.)
    or (flat1 == -90. and flat2 == -90.) ):
        delta = 0.
        azi = 0.
        return [delta,azi]

    # Perform calculation
    delta = 0.
    azi   = 0.

    raddeg=mt.pi/180.

    theta1=(90.-flat1)*raddeg
    theta2=(90.-flat2)*raddeg

    phi1=flon1*raddeg
    phi2=flon2*raddeg

    stheta1=mt.sin(theta1)
    stheta2=mt.sin(theta2)
```

```
    ctheta1=mt.cos(theta1)
    ctheta2=mt.cos(theta2)

    cang=stheta1*stheta2*mt.cos(phi2-phi1)+ctheta1*ctheta2
    ang=mt.acos(cang)
    delta=ang/raddeg

    sang=mt.sqrt(1.-cang*cang)
    caz=(ctheta2-ctheta1*cang)/(sang*stheta1)
    saz=-stheta2*mt.sin(phi1-phi2)/sang
    az=mt.atan2(saz,caz)
    azi=az/raddeg

    if (azi < 0.):
       azi=azi+360.

    return [delta, azi]
```

1.3.2 Hazlo tu mismo

Escriba un programa que use la función anterior (`sph_azi`) y solicite al usuario la latitutd y longitud de dos puntos en la Tierra, para imprimir la distancia y azimuth entre los dos puntos.

[2]:
```
# usuariodist.py
# Program para calcular la distancia y azimut entre dos
# puntos sobre una esfera.

lat1 = -1.
lon1 = -1.
lat2 = -1.
lon2 = -1.
while (lat1!=0. or lon1!=0. or lat2!=0. or lon2!=0.):
    intxt = input('Enter 1st point lat/lon ')
    lat1,lon1 = intxt.split()
    lat1 = float(lat1)
    lon1 = float(lon1)

    intxt = input('Enter 2nd point lat/lon ')
    lat2,lon2 = intxt.split()
    lat2 = float(lat2)
    lon2 = float(lon2)

    delta,azi = sph_azi(lat1,lon1,lat2,lon2)
    print('Distancia y acimut:  %6.2f  %7.2f '%(delta, azi))

# Bogota - New York
# 4.60 -74.08
```

```
# 40.71 -74.00
# Distancia y acimut:   36.11     0.10

# Bogota - Paris
# 4.60 -74.08
# 48.85 2.35
# Distancia y acimut:   77.63    40.91

# Bogota - Buenos Aires
# 4.60 -74.08
# -34.60 -58.37
# Distancia y acimut:   41.90   160.50
```

```
Enter 1st point lat/lon 4.60 -74.08
Enter 2nd point lat/lon 40.71 -74.00
Distancia y acimut:   36.11     0.10
Enter 1st point lat/lon 4.60 -74.08
Enter 2nd point lat/lon 48.85 2.35
Distancia y acimut:   77.63    40.91
Enter 1st point lat/lon 4.60 -74.08
Enter 2nd point lat/lon -34.60 -58.37
Distancia y acimut:   41.90   160.50
Enter 1st point lat/lon 0 0
Enter 2nd point lat/lon 0 0
Distancia y acimut:    0.00     0.00
```

En este caso, los valores de `lat/lon` son pasados a la función `sph_azi`, la cual devuelve las variables `delta` y `azi`. Es importante siempre poner nombres a las funciones que sean claras y que no repitan nombres de rutinas o variables ya existentes en Python.

1.3.3 Ojo con la documentación

La documentación de las funciones es **fundamental**. Debe ser claro que hace cada función, explicar las variables de entrada y salida, y si la rutina tiene problemas de precisión o cualquier otro problema o limitación es bueno documentarlo.

Puede que la documentación de esta función parezca larga y tediosa, pero con el número de funciones que Uds. van a trabajar, este tipo de documentación es ideal.

La documentación busca que Ud. o cualquier otra persona pueda utilizar la función correctamente sin necesidad de mirar el código. Por ejemplo, es claro que el azimuth es calculado del putno 1 al punto 2, y no al contrario. Explicar además las limitaciones de la función puede además ayudar a que se use de manera erronea. En distancias muy cortas, el programa puede tener limitaciones y devuelve un resultado que no es correcto y el usuario puede no entender esto.

Por último, note que la función intenta ser robusta en casos patológicos, como cuando los dos puntos son iguales, o cuando están en los polos (distancia y azimuth es cero).

1.3.4 Comentarios

Para calcular la distancia entre dos puntos en la Tierra, el código completo incluye las dos celdas, la que tiene la función `sph_azi` y programa o script principal. Ambos códigos deben estar juntos para que el programa completo funcione. Qué pasa si Ud. quiere usar la función `sph_azi` en el futuro? Le tocaría buscar este programa, copiar y pegar la función para poder usarla. Esto no es muy eficiente.

1.3.5 Hazlo tu mismo

Para evitar re-escribir mil veces la misma función, es posible escribir un *script* donde se puede poner la función deseada, de tal forma que el programa principal pueda `import` la función deseada, muy similar a cuando hacemos

```
import numpy as np
```

La tercera ley de Kepler gobierna el movimiento de un satélite alrededor de su planeta (o sol). El archivo `kepler.py` es un archivo de texto plano, que contiene una sola función (definición) y que calcula el periodo de la órbita (duración de un giro alrededor del sol) de un satélite (planeta, luna, ...) de acuerdo a la distancia con respecto al sol (planeta, etc) de acuerdo a la ecuación

$$GM_{sol} = \frac{4\pi}{T^2} a^3$$

donde G es la constante gravitacional, T es el periodo, a es el eje semi-major de su órbita (ver el archivo `kepler.py`).

- La Tierra está a 150 millones de km del Sol. Confirme su periodo.
- Un satélite espía está a aproximadamente 200 km de la superficie de la Tierra, cual es el periodo de su órbita?

[3]:
```
import kepler

d_sun   = 150e6 # distancia al sol, en km
T_earth = kep.kepler_3ra(d_sun)
T_days  = T_earth/86400

print('Periodo de órbita terrestre %5.1f días ' %(T_days))

Me        = 5.972e24 # kg
d_sat     = 6371+200 # distancia a la Tierra, en km
T_sat     = kepler.kepler_3ra(d_sat,Me)
T_smin    = T_sat/60

print('Periodo de órbita espía %5.1f minutos ' %(T_smin))
```

```
Periodo de órbita terrestre 366.7 días
Periodo de órbita espía  88.4 minutos
```

Explicación El archivo `kepler.py` contiene la función que hace el cálculo (similar a `sph_azi` o a `np.sim()`), pero Python necesita importarlo. Asumiendo que Python pueda encontrarlo (por

ejemplo si el archivo está en el mismo folder), éste se puede importar con `import kepler`, sin el `.py`. Para llamar la función se debe llamar el modulo y la función deseada `kepler.kepler_3ra`.

1.4 Modulos y Packages propios

Uno de los aspectos más importantes de generar funciones propias en Python, es poderlas utilizar en cualquier programa sin necesidad de pegarlas en cada programa. Esto permite tener una sola copia de la función y mantener una sola copia de la misma y no se requiere tenerla en el programa principal.

En el ejemplo anterior `kepler.py` se conoce como un **module** en Python. Es un archivo que contiene una o mas definiciones (puede tener muchas funciones, definiciones de variables, etc.) y que al importarla carga todas las funciones que han dentro del archivo.

A medida que pasa el tiempo, uno va construyendo una librería de funciones que usa habitualmente. Por ejemplo, yo tengo funciones o módulos para cálculos estadísticos, métodos de Fourier, simulación numérica, etc. Estas funciones las uso constantemente, pero no tengo que cambiarlas, sólo necesito llamarlas. Es lo mismo que siempre llamar numpy para calcular el seno de una variable.

1.4.1 Dónde guardar sus funciones?

En el ejemplo anterior, el módulo `kepler.py` se encuentra en el folder donde está corriendo Python. Si Ud corre Python desde otra ubicación del computador, python no encontrará la función y mosrtrará un error.

Note que Python *sabe* donde encontrar `numpy`, no hay que decirle donde está. Es una buena práctica tener un lugar donde se van a poner todos los módulos y paquetes que Ud va a desarrollar. Esto es buena idea porque - No va a tener 10 copias de la misma función - Puede tener organizado por temas sus funciones - Puede usar sus funciones desde cualquier lugar del computador

La único que toca hacer, es **enseñarle** a Python donde encuentra sus modulos.

El *Path* Cuando Python busca módulos siguiendo el path predefinido en el sistema, incluyendo el directorio donde está corriendo Python en ese momento. Esto se puede encontrar usando los comandos:

[1]:
```
import sys
print(sys.path)
```

```
['/Users/gprieto/Dropbox/gprieto/classes/unal/geoinformatica/python/chap04',
'/Users/gprieto/Dropbox/gprieto/classes/unal/geoinformatica/python/chap04',
'/Users/gprieto/Dropbox/gprieto/python/Modules',
'/Users/gprieto/Dropbox/gprieto/python/Modules/noisepy/src',
'/Users/gprieto/opt/anaconda3/lib/python37.zip',
'/Users/gprieto/opt/anaconda3/lib/python3.7',
'/Users/gprieto/opt/anaconda3/lib/python3.7/lib-dynload', '',
'/Users/gprieto/opt/anaconda3/lib/python3.7/site-packages',
'/Users/gprieto/opt/anaconda3/lib/python3.7/site-packages/aeosa',
'/Users/gprieto/opt/anaconda3/lib/python3.7/site-packages/IPython/extensions',
'/Users/gprieto/.ipython']
```

Es decir que si el usuario quiere crear módulos propios o paquetes propios, éstos deberían estar ubicados en alguno de los folders en el *path*. En muchos casos estos folders son del sistema y no se recomienda cambiarlos o adicionarles archivos.

En mi caso, el primer folder es el folder actual

`/Users/gprieto/Dropbox/gprieto/classes/unal/geoinformatica/python/chap04`

el segundo es donde pongo mis módulos

`/Users/gprieto/Dropbox/gprieto/python/Modules`

Este segundo Ud todavía no lo tendra, y toca mostrarle a Python.

Lo que se recomienda es crear un folder propio, en algún lugar donde el usuario pueda editar los archivos. Para que Python pueda encontrarlos, se debe adicionar el folder al *path*. Esto se puede hacer de dos maneras. La primera:

[3]:
```
sys.path.append('/Users/gprieto/Desktop')
print(sys.path)
```

```
['/Users/gprieto/Dropbox/gprieto/classes/unal/geoinformatica/python/chap04',
'/Users/gprieto/Dropbox/gprieto/classes/unal/geoinformatica/python/chap04',
'/Users/gprieto/Dropbox/gprieto/python/Modules',
'/Users/gprieto/Dropbox/gprieto/python/Modules/noisepy/src',
'/Users/gprieto/opt/anaconda3/lib/python37.zip',
'/Users/gprieto/opt/anaconda3/lib/python3.7',
'/Users/gprieto/opt/anaconda3/lib/python3.7/lib-dynload', '',
'/Users/gprieto/opt/anaconda3/lib/python3.7/site-packages',
'/Users/gprieto/opt/anaconda3/lib/python3.7/site-packages/aeosa',
'/Users/gprieto/opt/anaconda3/lib/python3.7/site-packages/IPython/extensions',
'/Users/gprieto/.ipython', '/Users/gprieto/Desktop']
```

Sin embargo esta opción sólo altera el `path` durante la ejecución del programa que lo use. La próxima vez que Python sea ejecutado, el `path` vuelve a su `default`.

La opción recomendada es la de crear un folder donde se pondrán todos los paquetes y módulos para su uso futuro. Para adicionar el folder de manera permanente el folder en el path, se debe adicionar la dirección del folder al `PYTHONPATH`. En sistemas operativos OS y Linux, esto se hace en el archivo `.bashrc` así:

```
export PYTHONPATH="${PYTHONPATH}:/my/other/path"
```

En otros ambientes tipo Unix, esto se puede adicionar al archivo `.bashrc`, `.profile`, `.cshrc` o cualquiera que sea el archivo de inicio (startup script) dependiendo de la shell que se use. En Windows, esto se puede hacer a través del GUI del sistema.

1.5 Usando mis *Modules* propios

La forma de llamar los módulos es similar a la utilizada para numpy. Tenga en cuenta que lops archivos `.py` se llaman módulos, y los folder que contienen 1 o más módulos y 1 o más folders, se llaman `packages`.

Por ejemplo, mi *path* busca modulos en `.../python/Modules`. En ese folder, hay un folder `clase` y dentro de éste, un módulo `sphere_subs.py`.

[3]:
```
import clase.sphere_subs as sph
print(sph.__doc__)
```

```
sphere_subs.py
   included in Package clase/
   /Dropbox/Dropbox/gprieto/python/Modules/clase

SPHERE_SUBS is set of Python function definitions to compute distances
and angles on a spherical Earth.  All angles are in degrees.
Latitude used on standard maps is geographic latitude; this may be
converted to the geocentric latitude used in these routines
by using the SPH_GEOCENTRIC function.  Longitude input to
these routines may be from either -180 to 180 or from 0 to 360.
Longitude returned from these routines will be from 0 to 360.

Based on Fortran subroutines from Peter M. Shearer

Last Modified
German A. Prieto
May 2017


Contains definitions of
sph_loc  - finds location of second point on sphere, given range
           and azimuth at first point.
sph_dist - computes angular separation of two points on sphere
sph_azi  - computes distance and azimuth between two points
           on the sphere
sph_mid  - finds midpoint between two surface points on sphere
           and azimuth at midpoint to second point
```

[4]:
```
print(sph.sph_mid.__doc__)
```

```
  SPH_MID finds midpoint between two surface points on sphere
  and azimuth at midpoint to second point

  Requires:  SPH_AZI, SPH_LOC

  Inputs:  flat1  =  latitude of first point (degrees)
           flon1  =  longitude of first point (degrees)
           flat2  =  latitude of second point (degrees)
           flon2  =  longitude of second point (degrees)
```

```
   Returns: delta  =  angular separation between points (degrees)
            flat3  =  latitude of midpoint (degrees)
            flon3  =  longitude of midpoint (degrees)
            azi    =  azimuth at midpoint to first point

   calls sph_azi, sph_loc
```

1.5.1 Módulos con muchas funciones

El archivo que contiene la definición de la función es `sphere_subs.py` y tiene la función sph_azi y puede tener otras definiciones de funciones. El encabezado del programa es:

```
""" sphere_subs.py
    included in Package clase/
    /Dropbox/Dropbox/gprieto/python/Modules/clase

 SPHERE_SUBS is set of Python function definitions to compute distances
 and angles on a spherical Earth.  All angles are in degrees.
 Latitude used on standard maps is geographic latitude; this may be
 converted to the geocentric latitude used in these routines
 by using the SPH_GEOCENTRIC function.  Longitude input to
 these routines may be from either -180 to 180 or from 0 to 360.
 Longitude returned from these routines will be from 0 to 360.

 Based on Fortran subroutines from Peter M. Shearer

 ...
"""
```

Los **módulos** son entonces archivos de python `.py` con una o muchas definiciones de funciones, clases, etc.

1.6 Los `Module`

Los `module` de Python son una de las principales capas de abstracción disponibles y son una forma natural para guardar definiciones de funciones que se usan de manera continua en Python. Estas capas permiten separar los códigos en partes relacionando datos y funcionalidad.

Por ejemplo, una capa o módulo de un programa puede enfocarse en la interacción con el usuario, otro módulo realiza manipulación de datos (cargar datos) y otro hace los cálculos matemáticos requeridos. Entonces, todas las funciones que hacen una parte del trabajo se agrupan en un solo archivo `.py`, las funciones de carga de datos en otro `.py` y finalmente las funciones que hacen cálculos, en un tercer `.py`. Para poder usar cada uno de los módulos, se deben importar en el programa principal con el comando `import`.

2 Los `Package`

En proyectos grandes, o para tener una librería de funciones, un programador busca agrupar funciones con diferentes objetivos. Python usa un sistema de paquetes, que es simplemente la extensión

de módulos a un directorio. En resumen, un paquete, es un folder con uno o más módulos dentro.

Cualquier directorio con un archivo `__init__.py` es considerado por Python un paquete. Los módulos dentro del paquete pueden ser importados por un programa de manera similar a los módulos individuales. El archivo `__init__.py` en principio tiene información y definición del contenido del paquete. Sin embargo, el archivo `__init__.py` puede estar vacio (no me pregunten porqué).

Un archivo modu.py en un directorio pack/ puede ser importado con el comando

```
import pack.modu
```

Este comando buscará un archivo `__init__.py` en el folder pack/, y ejecutará todos los comandos en el archivo. Después buscará el archivo `modu.py` y ejecutará sus comandos. Después de esto, todas las variables, funciones y clases definidad en `modu.py` estarán disponibles bajo el nombre pack.modu.

Es común ver que el archivo `__init__.py` tiene muchos comandos. Cuando un projecto es complejo y grande, puede tener varios sub-paquetes y subsub- paquetes en una estructura de folders larga y profunda. En este caso, importar una función dentro de la sub-estructura, implicaría ejecutar muchos `__init__.py` durante la carga de folder, sub-folder y sub-sub-folder.

Es normal dejar el archivo `__init__.py` vacio (sin náda escrito) e incluso esto es considerado como una buena práctica. Esecialemente es considerado una buena práctica si los módulos dentro de los paquetes y sub-paquetes no requieren compartir código (recuerde que Python ejecuta código linea por linea, por lo que el orden en el que se importan los módulos importa.

Finalmente, para evitar tener códigos muy cargados de texto, si uno quiere importar un módulo que se encuentra en un árbol de folders complejo, por ejemplo

```
import pack1.subpack2.subsubpack3.modu
```

y se quiere correr una función dentro del módulo, se necesitaría llamar la función

```
x = pack1.subpack2.subsubpack3.modu.sin(x)
```

lo cual hace muy el código muy difícil de leer. Una mejor opción en este caso sería

```
import pack1.subpack2.subsubpack3.modu as mod
...
x = mod.sin(x)
```

donde las funciones dentro del módulo se llaman con un encabezado más corto (`mod`).

```
import clase.sphere_subs as sph
...
```