Capítulo 4

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. Esto es una buena estrategia, ya que lo hace más modular y más fácil de leer y entender (y de encontrar problemas *bugs*). Esto se puede hacer creando funciones 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).

4.1. Funciones dentro del programa

Como ilustración de como introducir una función del usuario, el programa del máximo común denominador, se muestra a continuación:

```
# gcf2.py
# Calculate the greatest common factor (gcf) of two integers
# using a User defined function

# Define user function
def getgcf(x,y):
   amin = min(x,y)
   for j in range(1,amin+1):
      if ( (x%j)==0 and (y%j)==0):
         jmax = j
   z = jmax
```

```
   return z

# Start the main code

for i in range(10):

   intxt = input("Enter two integers (zeros to stop) ")
   a,b = intxt.split()
   a = int(a)
   b = int(b)

   if (a==0 and b==0):
      break
   else:
      gcf = getgcf(a,b)
   print ("Greatest common factor = ",gcf)
```

Ahora se realiza el cálculo del `gcf` dentro de la function `getgcf`. Esto se define dentro del programa (en el encabezado) con

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

Las variables `a` y `b` entran como argumentos de la función `getgcf` y el resultado de la función se da con la variable `z`. Note que debe usar `return z` para que el programa principal sepa que obtiene devuelta al llamar la función. Note además que se requiere mantener la identación dentro de la función.

El programa principal llama a la función a través del comando

```
gcf = getgcf(a,b)
```

Las variables `x` y `y` dentro de la función tomarán los valores que determina el programa al llamar a la función. 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`).

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.

4.2. Funciones con más 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 rsultado 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`.

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

```
# userdist.py
# Program to calculate the distance and azimuth of two points
# on the surface of the Earth.

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.
   #

   # import appropriate functions
   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]

#-----------------------------------------------
# Start main code
#-----------------------------------------------

for i in range(10):
   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)

   if (lat1==0 and lon1==0 and lat2==0 and lon2==0)
```

```
    break

delta,azi = sph_azi(lat1,lon1,lat2,lon2)

print ("del, azi =, ", delta,azi)
```

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 de Python.

También es importante siempre documentar claramente lo que hace cada función, explicando las variables de entrada y de salida, y si la rutina tiene problemas de precisión o cualquier otro potencial problema o limitación. Aunque lo que se muestra para la función como documentación parece ser demasiada información, pero tenga en cuenta que en el futuro Ud. va a tener una gran cantidad de funciones que va a utilizar y es importante entender claramente que hace cada función y cómo se llama desde un programa. Esto puede ahorrarle mucho tiempo en el futuro.

La documentación busca además 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 desde el punto 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.

Finalmente, note que la función trata de ser robusta en casos patológicos, como que los dos puntos tengan las mismas coordenadas (distancia y azimuth es cero) o estén en los polos.

4.3. modules & 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. Por ejemplo, yo he construido a través del tiempo un *module* con funciones para cálculos estadísticos, métodos de Fourier, simulación numérica, etc. Estas funciones las uso constantemente, pero no tengo que cambiarlas.

El siguiente ejemplo muestra el cálculo de la distancia sobre la esfera, pero utilizando un módulo propio.

```
# userdist2.py
# Program to calculate the distance and azimuth of two points
# on the surface of the Earth.
```

```
import clase.sphere_subs as sphere

for i in range(10):
   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)

   if (lat1==0 and lon1==0 and lat2==0 and lon2==0):
      break

   delta,azi = sphere.sph_azi(lat1,lon1,lat2,lon2)

   print ("del, azi =, ", delta,azi)
```

donde se importa el módulo `sphere_subs`, que se encuentra dentro del paquete `clase`.

Otra manera de llamar el módulo, que lo hace más sencillo es

```
from clase.sphere_subs import *
...
...
   delta,azi = sph_azi(lat1,lon1,lat2,lon2)
...
```

permitiendo así llamar la función que se quiere utilizar de manera directa.

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

def sph_azi(flat1,flon1,flat2,flon2):
   # SPH_AZI computes distance and azimuth between two points
   # on the sphere
   #
   ...
```

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

4.3.1. 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.

Cuando un módulo es importado, se pueden usar las funciones dentro del mismo. Puede ser importar módulos internos de Python (como numpy o math), módulos de terceros o módulos propios del usuario.

Se recomienda usar nombres para los módulos cortos, en minúscula y evitando usar símbolos especiales como punto (.) o (?). Evite usar nombres como mi.modulo.py, ya que esto puede interferir en la manera que Python interactúa con los módulos.

En el caso anterior (mi.modulo.py), Python esperaría encontrar un archivo modulo.py en un folder llamado mi (vea el ejemplo anterior). Es mejor usar mi_modulo, aunque esto a veces no es recomendable.

Fuera de estas consideraciones, no se requiere nada especial para que un archivo sea considerado un módulo de Python, sin embargo si es importante recordar el mecanismo de importación de Python de tal forma que su uso sea el adecuado.

Al usar el comando import modu, Python buscará el archivo modu.py en el folder donde se está trabajando, si está disponible. Si no lo está, Python buscará el archivo modu.py en el *path* definido por Python (ver más adelante como cambiar o adicionar directorios al *path*). Si no lo encuentra, un ImportError se despliega.

Si la importación es efectiva, Python ejecutará el módulo línea a línea. Es decir, que si el módulo tiene comandos particulares, éstos serán ejecutados, incluyendo comandos import. Definiciones de funciones y clases serán guardadas en el diccionario del módulo.

Todas las variables, funciones y clases que estén dentro del módulo estarán disponibles para el programa principal utilizando el nombre del módulo (por ejemplo math.pi es una variable del módulo math y equivale a 3.1415...). Este es un concepto fundamental de como trabaja Python.

En otros languajes (como Fortran o C), un comando include file se puede utilizar para que el programa pre-procese el código dentro del archivo file y lo incluye (o copie) en el encabezado del programa principal. La ventaja de usar los módulos en Python, es que no existe el riesgo que dentro del módulo haya una función que sobreescriba una función con el mismo nombre de una función propia de Python.

Sin embargo es posible cargar los nombres de funciones de manera directa, aunque esto puede ser **riesgoso**. Con el comando

```
from math import *
```

se importa de manera directa las funciones del módulo `math`, como el coseno y el seno (ver ejemplos en el capítulo anterior). Note que en el ejemplo anterior, se puede hacer algo similar con

```
from clase.sphere_subs import *
```

de tal forma que la función `sph_azi` esté disponible de manera directa.

Esto es considerado una *mala práctica* **en Python. Usar el comando `import *` hace que entender el código sea más difícil y saber las dependencias de las funciones usadas menos compartimentalizado.**

Usar comando como

```
from math import sin
```

es una forma de indicar la función específica que se quiere importar y ponerle un nombre global. Esto puede ser mejor que usar `import *` ya que muestra explicitamente que es importado, la única ventaja es que al llamar la función se ahorra en digitación. Obviamente, si se requiere cargar miles de funciones en un programa, el programa sería muy complicado de leer.

Algunos ejemplos de como importar módulos

```
[...]
from modu import *
[...]
x = sqrt(4)  # Is sqrt part of modu? A builtin? Defined above?
```

No recomendado.

```
from modu import sqrt
[...]
x = sqrt(4)  # sqrt may be part of modu, if not redefined in between
```

Mejor pero no ideal

```
import modu
[...]
x = modu.sqrt(4)  # sqrt is visibly part of modu's namespace
```

la opción más recomendada.

El objetivo del curso y de una buen estilo de programación no es escribir códigos cortos, sino códigos que sean legibles, claros y fáciles de utilizar por terceros. Legibilidad significa evitar exceso de texto o llenar espacio con comandos sin separación, pero tampoco significa que se deba llegar al extremo de oscurecer el código.

4.3.2. Los Packages

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 sub-sub-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 cópdigos 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`). Esto fue lo que se hizo en el programa `userdist2.py`

```
import clase.sphere_subs as sphere
...
delta,azi = sphere.sph_azi(lat1,lon1,lat2,lon2)
...
```

4.4. Ajustando el *path* para módulos propios

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:

```
>>> import sys
>>> print(sys.path)
```

con resultados en mi caso

```
>>> [’’,
’/opt/local/.../Versions/3.5/lib/python35.zip’,
’/opt/local/.../Versions/3.5/lib/python3.5’,
’/opt/local/.../Versions/3.5/lib/python3.5/plat-darwin’,
’/opt/local/.../Versions/3.5/lib/python3.5/lib-dynload’,
’/opt/local/.../Versions/3.5/lib/python3.5/site-packages’]
```

Es decir que si l 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.

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

```
import sys
sys.path.append(’/path/to/dir’)
print(sys.path)
```

y así el *path* quedará

```
>>> [’’,
’/opt/local/.../Versions/3.5/lib/python35.zip’,
’/opt/local/.../Versions/3.5/lib/python3.5’,
’/opt/local/.../Versions/3.5/lib/python3.5/plat-darwin’,
’/opt/local/.../Versions/3.5/lib/python3.5/lib-dynload’,
’/opt/local/.../Versions/3.5/lib/python3.5/site-packages’
’/path/to/dir’]
```

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 direcci?øn 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 el GUI del sistema.

Problemas

4.1. Modifique el programa del problema 3.3 para calcular el mínimo común múltiplo (least common multiple en inglés) con una función propia dentro del programa.

4.2. Modifique el programa del problema 3.3 para calcular el mínimo común múltiplo (least common multiple en inglés) con una función propia en un módulo propio. Ponga el módulo en el folder donde está trabajando.

4.3. Cree un paquete, con los sub folders que Ud. desee para poner todas las rutinas del curso y que Ud. pueda utilizar en el futuro. Como ejemplo, en mi caso yo tengo:

```
/Dropbox/Dropbox/gprieto/python/Modules
      -> clase/
            -> __init__.py
            -> sphere_subs.py
            -> ...
      -> spectra/
            -> mtspectra.py
            -> ..
```

y en mi archivo `.bashrc` tengo

```
export PYTHONPATH="/Dropbox/Dropbox/gprieto/python/Modules"}
```

lo que equivale a tener dos paquetes `clase` y `spectra`.

4.4. Cree una función propia (dentro de su `package` propio o dentro del programa) en Python que calcule el volumen y circunferencia de una esfera, si el usuario proporciona el radio. Haga un programa principal que solicite el radio al usuario, y haga que éste sea estable, que no acepte números negativos, y que imprima el resultado.