Temario

2do Parcial día 13 de Noviembre.

- Programación orientada a objetos
- Manejo de errores
- Debugging

Programación orientada a objetos

Es un paradigma de la programación que permite tener códigos flexibles que pueden crecer indefinidamente y reutilizar los códigos desarrollados.

Desde el punto de vista del debugging localiza la propagación de los errores.

Objetos son conjuntos de variables (datos) y de funciones que operan sobre esas variables.

Variable $\rightarrow$ Atributo
Función $\rightarrow$ Método

Elementos de una clase

```
class Auto:
   def __init__(self,color='',posx=0,posy=0):
      # Posicion initial
      self.posx=posx
      self.posy=posy

   def mover(self,dx=0,dy=0):
      self.posx=self.posx+dx
      self.posy=self.posy+dy
```

Se suele usar mayúsculas para nombrar a los objetos.

self: Es una forma de acceder a los atributos y métodos adentro del objeto.

Instanciación de una clase

Se crean instancias de la clase o se definen objetos de la clase al llamar a esta:

```
citroen=Auto(color='amarillo',posx=14.0,posy=2.0)
```

Cada objeto es llamado una instancia de la clase. También se dice que se instancia cuando se crea el objeto.

Cada clase se puede instanciar cuantas veces se quiera, entre éstas son totalmente independientes:

```
clio=Auto(color='gris',posx=20.0,posy=5.0)
peugeot=Auto(color='blanco',posx=0.0,posy=0.0)
```

Creación del objeto

Cuando se hace referencia al objeto Python va a crear la instancia y llama a

```
Auto.__init__
```

Este es el "inicializador" de la clase. Alli se ponen los parametros que se quieren definir por default.

Entonces:

```
clio=Auto(color='gris',posx=20.0,posy=5.0)
```

y

```
clio=Auto.__init__(color='gris',posx=20.0,posy=5.0)
```

son exactamente lo mismo.

self

```
def __init__(self,color='',posx=0,posy=0):
```

Notar la presencia de self como argumento de entrada. Esto es obligatorio en todos los métodos de una clase.

En cada método o función ingresan como argumentos de entrada todos las variables y métodos de la clase. Esto es explicitado en la variable self.

El self nos dice que desde cualquier lugar de la clase podemos acceder a las atributos de la clase y/o los métodos de la clase:

self.posx me dice la posicion del auto. Es decir es como que tenemos variables “modulares” son compartidas por todos los métodos de la clase. Estas son distintas de las variables locales de cada método y de las variables globales.

Referencias a los atributos de una clase

Con el nombre de la instancia punto y el atributo se puede acceder a cualquier atributo de la clase.

```
print ('El citroen es de color: ', citroen.color)
```

En este caso citroen.color es un atributo tipo string de la clase.
También se puede acceder a las funciones de la clase que es un atributo método

```
clio.mover(10,5)
print ('La posicion actual del clio es:',clio.posx,clio.posy)
```

Otro ejemplo: Movimiento uniformemente acelerado

```
class MUR:
    def __init__(self, x0=0.0, v0=None, a=-9.81,xunit='m',tunit='s'):
        self.x0=x0
        self.a = a
        if v0 is not None:
            self.v0=v0
        else:
            quit('Se requiere v0')
#  sys.stderr.write('Se requiere v0'); sys.exit(1)
    def posicion(self, t):
        return self.x0+self.v0*t + 0.5*self.a*t**2
    def velocidad(self, t):
        return self.v0+self.a*t
```

Otro ejemplo: Movimiento uniformemente acelerado

En el programa principal creo o instancio el objeto:

```
Mov=MUR(x0=10,v0=5)
t=linspace(0,100,101)
x=Mov.posicion(t)
v=Mov.velocidad(t)

plt.subplot(2, 1, 1)
plt.plot(t,x)
plt.subplot(2, 1, 2)
plt.plot(t,v)
plt.show()
```

Llamada por defecto: __call__=self.posicion

```
x=Mov(t)
```

No requiero mencionar el método.

Pares ordenados: vectores de dos componentes

```
import copy
class Vector:
    def __init__(self, l):
        self.a=copy.deepcopy(l)#genero nueva variable
    def __add__(self, b): # Esta asociado al +
        return Vector([self.a[0]+b.a[0],self.a[1]+b.a[1]])
        # Genero un nuevo objeto a la salida
        # Se esta autoreferenciando
    def prod_int(self, b):
        return self.a[0] * b.a[0] + self.a[1] * b.a[1]
    def __mul__(self, alpha): # producto por un escalar --> *
        return Vector([self.a[0] * alpha, self.a[1] * alpha])
    def impr(self):
        return '('+str(self.a[0])+','+str(self.a[1])+')'
```

Pares ordenados: vectores de dos componentes

```
a=Vector([10.0,-5.0])
b=Vector([7.,8.])
alpha=5.0
c=a+b # estamos realizando la operacion __add__definida
d=a*alpha # producto por un escalar

print ('Suma:')
print (a.impr()+'+'+b.impr()+'='+c.impr())
print ('Producto:')
print (a.impr()+'*'+alpha+'='+d.impr())
```

Herencia

Los atributos de una clase se pueden heredar por otra clase.

```
class Camioneta(Auto):
   def __init__(self,color='',posx=0,posy=0,carga=0,maxcarga=0):
      Auto.__init__(self,color='',posx=0,posy=0):
      self.carga=carga
      self.maxcarga=maxcarga
   def subo_mercaderia(peso):
      self.carga+=peso
```

```
f100=Camioneta(posx=0,posy=0,carga=0,maxcarga=1000)
f100.mover(dx=10,dy=-5)
f100.subo_mercaderia(100)
f100.subo_mercaderia(80)
```

Notar que estoy utilizando los métodos de la clase padre.

Manejo de errores

Cuando se produce algun error durante la ejecución Python detiene la ejecución y emite un mensaje de Error.

```
>>> 5./0.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: float division by zero
```

El mensaje nos da una traza del error dice el archivo (o los archivos) donde se produjo, la línea y el módulo.
En la línea siguiente se muestra el tipo de error.

Debugging vs control de errores

Cuando se produce un error existen dos posibilidades:

- El error es imprevisto debido a algun problema del codigo. Debemos hacer un debugging.
- El error es posible en la lógica del programa por lo que deberíamos hacer tratamiento en el código a través de Excepciones.

Como en el resto de la programación para el testeo/chequeo y el debugging es esencial la modularización del código.
Si el error esta localizado en 10 líneas de código es muy fácil encontrarlo.
Si esta en 10000 líneas de codigo es imposible.

Debugging el código

Opción mas artesanal: Se pueden introducir print de las variables antes de la línea donde aparece el error para ver lo que esta sucediendo.
Python trae incorporada una librería para el debugging:

```
import pdb
# donde se quiere que comience a rastrear el bug agregar:
pdb.set_trace()
```

Comenzar a compilar python problema1.py
(en modo debugging)

Comandos para el debugging

Comandos esenciales:

n next ejecuta la línea

p a,b,c imprime variables

q si queremos salir sin seguir ejecutando.

c si queremos continuar la ejecución sin seguir debugging

Excepción

Existe una forma de probar si da error y si es asi le decimos que haga algo.

```
try:
    a=1/b
except ZeroDivisionError,a:
    print 'Division por Zero'
```

De esta maneras evitamos el error y se continua con la ejecución. En la programación se debe tener en cuenta la aparición del error. En el ejemplo `a` no esta definido

Otros ejemplos

Chequea que exista el módulo a importar:

```
try: #fortran
    import hdmpfmod
except ImportError:
   quit('Requires compilation of fortran modules')
```

Otros ejemplos

Chequea que exista una variable, sino existe la crea con None:

```
def checkpar(par):
    " Define los parametros indefinidos como None "
    try:
        par
    except NameError:
        par = None
```

Este comando es util para el manejo de diccionarios o atributos de objetos.

Ejemplo

Si una variable no existe me dara un error

```
print gravedad.g
```

Como hago para definirle un valor a la variable solo en el caso en que no lo tenga?

```
checkpar(self.g) # la defino como None
if self.g is None:
   self.g=9.8
```

Excepciones en el ingreso de datos

Para el problema que hemos tenido con el ingreso de datos existe una solución sencilla con el uso de las excepciones:

```
def readInt():
    while True:
        val = raw_input('Ingrese un valor entero: ')
    try:
        val = int(val)
        return val
    except ValueError:
        print val, 'no es un entero'
```

Una forma mas general que me sirve para cualquier dato

Para el problema que hemos tenido con el ingreso de datos existe una solución sencilla con el uso de las excepciones:

```
def readValue(valType,InputMsg,ErrorMsg):
   while True:
      val = raw_input(InputMsg)
      try:
         val = valType(val)
         return val
      except ValueError:
         print val, ErrorMsg
```

Esta función puede ser usada para cualquier tipo de datos (polimorfica).

Control de los errores

Se puede generar una excepción voluntaria en nuestro programa mediante la orden 'raise'.
De esta manera podemos controlar cuando aparezcan errores que sabemos de antemano pueden aparecer.
Cuando le pasamos la excepción le decimos el tipo de error y una cadena de carácteres que indique el contexto:

```
>>> raise ZeroDivisionError,"El denominador se hizo 0"
Traceback (most recent call last):
File "<stdio>", line 1, in <module>
ZeroDivisionError: El denominador se hizo 0
```