Introducción a Python

Temario de la Clase 3

- Funciones.
- Argumentos.
- Variables locales y globales.
- args and kwargs
- Desarrollo de librerías propias.
- Clases y objetos. Una intro

Sintaxis de una función en python

Las funciones se definen con un `def` luego el nombre de la función y entre paréntesis los argumentos de entrada:

```
def funcion_3Dgral(x,y,z,escala=2):
    Instrucciones
    ...
    return v1,v2
```

Se debe tabular a todo el cuerpo de la función.

- Los argumentos de entrada son posicionales y/o nominales.
- Termina con un `return` y las variables de salida.
- Si no hay variables de salida, no es necesario el `return` (fin de tabulación).

Ejemplo de uso de función.

Tenemos que hacer cambios de grados Celsius a Fahrenheit, $F(C) = \frac{9}{5}C + 32$

```
def trans_c2f(tcel):
    ''' Transforma temperatura de Celsius a Fahrenheit'''
    return 9./5.*tcel+32.0
```

Luego en el programa principal llamamos a la función:

```
ta = 10
temp = trans_c2f(ta)
print ( trans_c2f(ta+1) )
sum_temp = trans_c2f(10.0) + trans_c2f(20.0)
```

- Aun cuando la función se utilice/llame una sola vez en el programa principal, conceptualmente un programa es mucho mas claro.
- Aun cuando la función sea de una sola línea también es conveniente.

Las funciones deben ir definidas antes de llamarlas.

Función lambda

Si queremos definir una función muy corta de una línea solamente:

`funcion = lambda argumentos : comandos`

```
>>>  sinsquare = lambda x:math.sin(x*x)
>>>  sinsquare(2.1709)
-0.9999999127092075
```

El nombre de la función es sinsquare y recibe un argumento x.

Si queremos definir una función con dos argumentos de entrada:

```
>>>  sinsquare2d = lambda x,y:math.sin
                   (x*x+y*y)
>>>  sinsquare2d(2,1)
-0.95892427466313846
```

```
>>>  sinvector2d = lambda x,y:(math.
                   sin(x*x+y*y),math.
                   cos(x*x))
>>>  vx,vy = sinvector2d(0.5,0.1)
```

También la de la transformación de temperatura:

```
trans_c2f = lambda t_cel: 9./5.*t_cel+32.0
```

Excepto para funciones adentro de un contexto siempre conviene el `def`, es mas legible.

Múltiples return

Una función puede tener varios return generalmente ligados a bifurcaciones:

```
>>>  def determina_signo(x):
...      if x>0 :
...          return 'positivo'
...       elif x<0
...          return 'negativo'
...       else:
...          return 'cero'
```

Si evaluamos `determina_Signo(10)` el primer argumento es True y por lo tanto se sale de la función sin continuar ejecutando el resto de los comandos.
Si queremos terminar una función

```
>>>  def busca_impares(lista_nros):
...      for i in lista_nros:
...          if i%2 = 1:
...              return i
...      return None
```

Si no hay return lo que se devuelve es un None (la última línea es innecesaria).

Ejemplo

Queremos encontrar el primer numero de una lista que sea divisible por 3.

```
def f ( lista ):
    for x in lista :
        if x %3 == 0:
            break
```

Es correcto? Cambios?

Queremos encontrar todos los números de una lista que son divisibles por 3. Que cambiamos?

Queremos una función que nos devuelva 'Izq' si $x < 0$, 'Medio' $0 \leq x \leq 10$ y 'Der' $x > 10$

```
def f(x):
    if x<0:
        return 'Izq'
    if x>=0:
        return 'Medio'
    if x>10:
        return 'Der'
```

```
def f(x):
    if x> 10:
    return 'Der'
    if x>= 0:
    return 'Medio'
    return 'Izq'
```

¿Cual es la correcta?

Regla: Funciones todo funciones

REGLA DE ORO: programa principales pequeños que llaman a funciones.

Es mucho mas fácil programar con funciones:

- Las funciones son unidades básicas independientes, esto nos permite reciclarlas, ej. en cualquier programa que necesite transformar temperatura puedo utilizar la función trans_c2f.
- Además tiene muy bien definido todo lo que necesita para funcionar (Argumentos de entrada) y cuales son los resultados (variables de salidas).
- El debugging de una función de pocas líneas es mucho mas sencillo (Se aisla del resto).
- Las variables que se utilizan en las funciones son locales. Veamos...

Variables locales. Nacen y mueren en la función

Las variables que se definan en una función son variables locales, es decir se crean para la función, pero luego en el return se destruyen y no afectan el resto del programa.

```
ta = 10
def trans_c2f(tcel):
    factor=9./5.
    ta=factor*tcel+32.0
    return ta
F1 = trans_c2f(10)
print 'Variable ta: ',ta
print 'Variable factor: ',factor
```

Que da el print de ta? el valor que se calcula en la función o el que esta en el programa principal?
Que da el print del factor?

Cambio de valor en variables globales?

¿Las variables que se definan fuera de la función pueden ser utilizadas en la función?

```
factor = 5.
def trans_c2f(tcel):
    factor = 9./5.
    ta=factor*tcel+32.0
    return ta
F1 = trans_c2f(20)
print 'Variable ta: ',ta
print 'Variable factor: ',factor
```

¿Qué da el print del factor?

¿Pueden describir cual es la diferencia con el caso anterior? ¿Cúal es la lógica?

Cambio de valor en variables globales

Para redefinir una variable adentro de la función se utiliza:

```
factor = 5.
def trans_c2f(tcel):
    global factor # aqui avisamos que estamos usando
                  # la variable global "factor"
    factor = 9./5. # aqui NO esta creando una variable local
                   # pero redefiniendo la variable global
    ta=factor*tcel+32.0
    return ta
F1 = trans_c2f(ta)
print 'Variable ta: ',ta
print 'Variable factor: ',factor
```

Que da el print del factor?

Pueden describir cual es la diferencia con el caso anterior?

Funciones internas

Si se quiere compartir las variables de una función con otra, evitando de esta manera el exceso de argumentos la puedo poner como una función interna:

```
>>>  def f(x,y):
...      def g(x):
...          return a*x*(y+1)
...      a=5
...      z=g(x+1) + y
...      return z
```

- Notar que tanto a como y estan definidas en f y son utilizadas en la función interna g.
- La función g pasa a ser una variable local de f, no se puede acceder desde “afuera” (otras funciones).
- Si quiero redefinir una variable de f se usa `nonlocal`

Las funciones son objetos de primera clase

- Se crean en tiempo de ejecución
- Se pueden asignar a una variable o elemento de una estructura (eg. lista).
- Se pueden pasar como argumento en una función
- Se pueden devolver como resultado de una función

```
  >>>  def factorial(n):
...      '''Dado n nos da n! '''
...      return 1 if n < 2  else n * factorial(n-1)
>>>  fac=factorial
>>>  fac(4)
24
```

Donde ubicamos a las funciones?

Si estamos escribiendo en un solo archivo al programa principal y las funciones:
Las funciones deben ir antes del llamado a éstas.

```
# Primero Funcion
def trans_c2f(tcel):
    factor=9./5.
    ta=factor*tcel+32.0
    return ta
# Llamando a la funcion desde el programa principal
F1 = trans_c2f(10.)
```

Esto es así porque python es un interprete, entonces necesita tener “interpretada” la función, antes que se la quiera utilizar, de lo contrario no sabría lo que es “trans_c2f”.

Archivo con un principal y varias funciones

Si tenemos un varias funciones y un main.

Usar un main es una buena practica incluso si solo son funciones que van a ser llamadas desde otros mains.

En esos casos los mains son mas que nada para probar a las funciones o ejemplos de uso.

El archivo transforma_temperaturas.py contendría:

```
def trans_c2f(tcel):
    return 9./5.*tcel+32.0
def trans_f2c(tf):
    return 5./9*(tf-32.0)

if __name__ == '__main__':
    trans_c2f(23)
```

Esto permite utilizar el transforma_temparaturas.py de dos maneras:

1. Desde ipython hago: `>>> %run transforma_temperaturas.py` o desde terminal: `$ python transforma_temperaturas.py`
2. Desde otro script hago:
   `import transforma_temperaturas as transf` y luego uso
   `transf.trans_c2f(23); transf.transf_f2c(120)`

Donde ubicamos a las funciones?. Librerías

- Las funciones pueden ir en un archivo aparte en el mismo directorio: una librería
- También pueden ir en un subdirectorio pero en este caso necesito agregar un archivo vacio:

  __init__.py

  Esto le avisa a python que es un directorio python y eventualmente se puede utilizar este archivo para inicializar librerías del subdirectorio.

Para utilizar una librería nuestra en un script es equivalente a las de python. Le tenemos que decir que la importe:

```
import libreria
```

En este caso python busca por el archivo libreria.py sin .py!. Es decir el nombre de la librería esta asociado al nombre del archivo.

Si es en un subdirectorio:

```
import dir.libreria
```

Donde busca python a las librerías que importamos?

La variable sys.path, tiene la lista de variables en las cuales python va a buscar la librería. Esta compuesta por:

1. El directorio de trabajo actual
2. Los directorios definidos en la variable PYTHONPATH
3. Directorios de las librerías base de python

Puedo definir directorios de busqueda incorporandolos a la variable de entorno `PYTHONPATH`

Para ver que tiene definido hacer `$ echo $PYTHONPATH` o

```
>>>  !echo $PYTHONPATH
```

Para definir nuevos directorios:

```
$ export $PYTHONPATH:\home\pulido\assm\pyt
```

También los puedo definir en el propio script:

```
import sys; sys.path.insert(0, '../models/')
import [modelo]
```

Importación de nuestras ibrerías

Supongamos que guardamos en el archivo transformaciones.py las funciones trans_c2f y trans_f2c.

En el programa principal transforma.py hacemos:

```
#!/usr/bin/env python
"Transforma de grados centigrados a fahrenheit y viceversa"
import transformaciones as tr

print 'Que desea calcular: '
opt=input('(1) Celsius a Fahrenheit, (2) Fahrenheit a Celsius:')
if opt == 1:
    tcel=input('Ingrese Temperatura en Celsius')
    tfah=tr.trans_c2f(tcel)
    print 'Temperatura de ',tcel,' Celsius es ',tfah,' Fahrenheit'
elif opt == 2:
    tfah=input('Ingrese temperatura en Fahrenheit')
    tcel=tr.trans_f2c(tfah)
    print 'Temperatura de ',tfah,' Fahrenheit es ',tcel,' Celsius'
```

La importación por convención es al comienzo del archivo!.

Si esta en un directorio funciones tenemos que hacer

```
import funciones.transformaciones as tr
```

Número de argumentos variables: *args

Ya vimos lo que hace *var individualiza una lista o tupla.

Esto es de gran utilidad para funciones que no sabemos cuantos argumentos de entrada tienen ya que puede cambiar de acuerdo al contexto.

`args` es una tupla iterable por lo que podemos recorrer sus elementos. Ejemplo:

```
>>> def my_fn(primero, *args):
...     print("Primer argumento :", primero)
...     for arg in args:
...         print("Siguente *args :", arg)
...
>>>  my_fn('Intro', 'a', 'Python')
Primer argumento : Intro
Siguiente *args : a
Siguiente *args : Python
>>>  my_fun(1,2,3,4,5) # ahora van 5 numeros
```

Tambien podemos usarlo en el llamado de la funcion: `>>> myFun(nro,*lista)`

```
>>> lista=[1,2,3]
>>> myFun(*lista)
Primer argumento : 1
Siguiente *args : 2
Siguiente *args : 3
```

Número de argumentos nominales variables: **kwargs

¿Que sucede en el caso de argumentos nominales? En ese caso tenemos a **kwargs que es un diccionario de los argumentos nominales.
El nombre del argumento es la clave en el diccionario.

```
def function(**kwargs):
    for key, value in kwargs.items(): # recorro items del diccionario
        print(f" El valor de la clave {key} es {valuee}")

function(nombre="Damian", domicilio="Rivadavia 953", edad=23)
```

Tambien podemos usar a **kwargs como argumentos nominales de entrada.

```
  def function(nombre=None, domicilio=None,edad=0):
        print(f" El nombre es {nombre}")
        print(f" Edad: {edad}")

caso={'nombre':'Damian', 'domicilio':'Rivadavia 953', 'edad':23}
function(**caso)
```

Ejemplo evaluador de funciones arbitrarias

Desarrollar una función cuyo primer argumento de entrada sea una función python y el resto de los argumentos, uno o dos, van a ser argumentos de esta función de entrada.

```
def funcion(fn,*args):
    return fn(*args)
```

Probar a esta función con funciones de entrada math.sqrt, math.sin, atan2, pow.

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 posición del auto.

- Tenemos variables “modulares” que 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)

pl.subplot(2, 1, 1)
pl.plot(t,x)
pl.subplot(2, 1, 2)
pl.plot(t,v)
pl.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

`super().__init__()` da acceso a los métodos en una superclase desde la subclase/clase iha la cual hereda todo
La subclase contiene todos los atributos y métodos que tiene la superclase. Ahorro enorme de coding. Reciclado de códigos.

```
class Madre():
    def __init__(self,in_f,out_f):
        self.in_f=in_f
        self.out_f=out_f
        self.inicializacion(in_f,out_f)
    def inicializacion(self,in_f,out_f):
        self.parametros = randn(in_f,out_f)

class Hija(Madre):
    def __init__(self,in_f,out_f):
        super().__init__(*args)
        print(self.parametros.shape)
        # necesito generar nuevos paraemtros aleatorios
        Madre.inicializacion(in_f,out_f)
```

Se puede heredar de múltiples clases: `class Hija(Madre,Tia)`. Tiene orden de precedencia la primera si hay atributos o métodos que se superponen.

Ejemplo geometría

```
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

    def perimeter(self):
        return 2 * self.length + 2 * self.width

class Square(Rectangle):
    def __init__(self, length):
        super().__init__(length, length)
Lote_belgrano=Square(25)
Lote_moreno=Square(35,15)
print('Superficie Belgrano: ',Lote_belgrano.area())
print('Superficie Moreno: ',Lote_moreno.area())
```

Ejemplo Pytorch Red Neuronal FC

```
class Linear(Module):
    r"""Applies a linear transformation to the incoming data: :math:`y = xA^T + b`.
    """
    def __init__(self, in_features: int, out_features: int, bias: bool = True,
                 device=None, dtype=None) -> None:
        factory_kwargs = {'device': device, 'dtype': dtype}
        super().__init__()
        self.in_features = in_features
        self.out_features = out_features
        self.weight = Parameter(torch.empty((out_features, in_features), **factory_kwargs))
        if bias:
            self.bias = Parameter(torch.empty(out_features, **factory_kwargs))
        else:
            self.register_parameter('bias', None)
        self.reset_parameters()

    def reset_parameters(self) -> None:
        init.kaiming_uniform_(self.weight, a=math.sqrt(5))
        if self.bias is not None:
            fan_in, _ = init._calculate_fan_in_and_fan_out(self.weight)
            bound = 1 / math.sqrt(fan_in) if fan_in > 0 else 0
            init.uniform_(self.bias, -bound, bound)

    def forward(self, input: Tensor) -> Tensor:
        return F.linear(input, self.weight, self.bias)
```

Ejemplo de herencia en torch

Supongamos quiero adaptar la clase de pytorch para que cuando hace la inferencia/propagacion hacia adelante le aplique una funcion de activacion.

```
class ActLinear(torch.NN.Linear):
    r"""Applies a linear transformation with activation function`.
    """
    def __init__(self, in_features: int, out_features: int, act=F.Relu, bias: bool = True,
                 device=None, dtype=None) -> None:
        super().__init__( in_features, out_features, bias=bias, device=device, dtype=device )
        self.act = act # activation function

    def forward(self, input: Tensor) -> Tensor:
        return self.act( F.linear(input, self.weight, self.bias))
```

Puedo acceder a los métodos y variables definidas en la madre.

No requiero rehacerlos a menos que necesiten de modificaciones

```
def __init__(self,*args,act=F.Relu [atributos-opcionales], **kwargs):
    super().__init__(*args, **kwargs) # inicializo con los mismos atribu
```