Introducción al lenguage Python

Temario de la Clase 2

- Variables Lógicas. Bifurcaciones. Condicionales.
- Loops. Ciclos.
- Ordenando
- Comprehensions de listas
- Enumerate Zip. Argument Unpacking.
- Funciones/subrutinas en python. (?)

Variables lógicas

Una variable lógica puede tomar dos valores: True o False.

```
>>>  lpreg=True
>>>  type(lpreg)
<type 'bool'>
```

Son de utilidad para switches, configuraciones de si se quiere que el código tenga determinadas características o no de acuerdo al interesado.
Se usan de máscara cuando se trabaja con datos.
Las tres operaciones de variables lógicas mas reconocidas: and, or y not.

```
>>>  lresp=False
>>>  lresp and lpreg
False
```

Operaciones combinadas (OJO con el orden!)

```
>>>  lcom=lresp and (lpreg or not lbe)
>>>  lcom
False
```

Las variables lógicas se asocian a 0 (False) y 1 (True) en python.

Operadores que resultan en variables lógicas

Es la variable `a` igual a la variable `b`? Simbolo utilizado para representarlo: ==

```
>>> lresp = a == b
```

Es la variable a distinta a la variable b? !=

```
>>> lresp = 1!=2
```

Es la variable a mayor a 5? >

```
>>> lresp = a > 5
```

```
>>> lresp = a >= 6
```

Combinación de operaciones:

```
>>> lresp = a >= 6 and a <=10
```

El resultado de todas estas operaciones es una variable lógica. True False

Operadores lógicos para cadena de carácteres

```
>>> s1 = 'bc'
>>> s2 = 'abcde'
```

El operador in pregunta si una cadena se encuentra en la otra:

```
>>> s1 in s2
```

El operador in not pregunta si una cadena no se encuentra en la otra:

```
>>> s1 in not s2
```

El operador is pregunta si una variable es la otra.

```
>>> x0 is y
>>> x0 is None
```

Las variables las puedo definir como 'None'

Nota: `is` da True si las dos variables se refieren al mismo objeto en memoria.

`==` da True si las dos variables son iguales

Ejemplo:

```
In [9]: x=[1,2]
In [10]: y=[1,2]
In [11]: x is y
Out[11]: False
In [12]: x == y
Out[12]: True
```

La instrucción if: condicional

Hay muchas veces en un programa que vamos a querer controlar el flujo, es decir que el programa haga algo si la respuesta es afirmativa y que no lo haga si la respuesta es negativa:

```
>>> syes = input("Desea terminar (s): ")
>>> if syes == 's':
...     print ('Respuesta s=si. Termino el programa')
...     raise SystemExit
```

La estructura de la instrucción if es:
if (variable lógica): Si la variable lógica es verdadera entonces:
(4 espacios en blanco) Hace esto.

Los espacios en blanco, tabulación, son parte de la instrucción. (No endif)

Convención: Se puede usar 2, 4, 8 espacios en blanco. La recomendada para las PEPs es 4.

La instrucción if-else

Si pasa esto, haga algo si no pasa eso hace otra cosa:

```
syes=input("Desea continuar (s/n): ")
if syes == 's':
    print( 'Respuesta s=si continua.' )
else:
    print( 'Cualquier otra respuesta termina. ' )
    raise SystemExit
```

La estructura de la instrucción if-else es:

if (variable lógica): Si la variable lógica es verdadera entonces:

(4 espacios en blanco) Haga esto

else: Si la variable lógica es falsa entonces

(4 espacios en blanco) Haga esto otro

Condicionales anidados: elif

Hay veces que necesitamos varios condicionales anidados.

Para esto existe el elif.

Es una mezcla de else y de if, “de lo contrario si es que pasa esto”

```
a=input('Introduzca un nro: ')
if a == 0:
    print ('El nro es zero')
elif a> 0:
    print ('El nro es positivo')
else:
    print ('El nro es negativo')
```

Nota: Si quieren copiar y pastear para hacer una prueba en `ipython` con los ejemplos de la teoría se puede pastear con `%paste`

Sin embargo el pdf cambia los carácteres ’ y los espacios espacios en blanco.

Varias opciones elif. Ejemplo case.

El elif es útil para cuando se le da opciones al usuario [No existe el case].

```
a=float(input('Introduzca un nro: '))
print ('Que desea calcular: ')
opt=float(input('(1) Cuadrado, (2) Raiz cuadrada, (3) Logaritmo:'))
if opt == 1:
    print ('El cuadrado es: ',a**2)
elif opt == 2:
    print ('La raiz es: ',math.sqrt(a))
elif opt == 3:
    print ('El logaritmo es: ',math.log(a))
else:
    print ('Hay solo tres opciones 1,2,3')
```

En estos casos siempre conviene usar un else a lo último para cualquier problema que hubo en el ingreso de los datos (o cuando se esta ejecutando el programa), Entonces estamos avisando de que “No se encontró ninguna opción válida”.

if else en una sola línea.

Operación ternaria:

```
>>> variable = valor_si_True if condicion else valor_si_False
```

Lo primero que se evalua es la condición luego se ve el valor verdadero o el falso de acuerdo al resultado de la condicion.

Ejemplo:

```
>>> is_apple = 'Yes' if fruit == 'Apple' else 'No'
```

En el caso que no hay `else` se hace con la forma tradicional:

```
>>> if condicion: variable = valor_si_True
```

Ejercicio. Encontrar las raíces de una ecuación cuadrática

Se quiere realizar un programa que dada una funcion cuadratica nos encuentre los ceros de la equación:

$$ax^2 + bx + c = 0$$

La solucion de esta ecuacion es:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Es decir que debemos tener en cuenta que hay tres soluciones posibles.

Ejemplo. Encontrar las raíces de una ecuación cuadrática

```
print( 'Determina raices reales de a x^2 + b x + c = 0' )
a=float(input('Introduzca a: ')) # Idem b y c. Da error si no es flotante

rad = b**2 - 4 * a * c
if rad < 0:
    print( 'La ecuacion no tiene raices reales' )
elif rad == 0:
    print( 'La ecuacion tiene una raiz',-b/(2*a) )
else:
    print( 'Tiene dos raices: ' )
    sqr= rad**0.5 /(2*a)
    raex=-b/(2 * a)
    print( 'Radic. Positivo: ',raex + sqr )
    print( 'Radic. Negativo: ',raex - sqr )
```

Bucles con for

La instrucción mas importante para hacer bucles o repeticiones de órdenes es con for. Este se usa para tomar valores de una lista.

for i in lista de valores:

```
>>> a=['a','b','c']
>>> for char in a:
        print char,')'
a )
b )
c )
```

Otra forma muy utilizada es usando la generación de listas con range:

```
>>> for i in range(3):
...     print (i,')'}
0 )
1 )
2 )
```

La cantidad de repeteciones realizadas es la cantidad de elementos que tiene la secuencia: lista, dictionario, etc

Bucles con for

Si quiero terminar el ciclo actual en algun lugar arbitrario `continue`.

Es decir que va a volver a punto de inicio tomar un nuevo elemento de la lista.

Si queremos terminar el `for`, si se cumple alguna condición usamos `break`.

En este caso el flujo continua en la línea después de que se termina el for.

```
for i  in range(100):
     < calculos >
     if error:
          print 'Ocurrio un error en el bucle'
          break
     < mas calculos >
```

Bucles/ciclos/loops: con while.

El comando while hace que la computadora repita una serie de órdenes hasta que se cumpla una condición lógica.

Queremos producir un bucle hasta que se cumpla una condición:

```
i=0
sum=0
while sum<=80:
    print (i)
    i += 1
    sum += i
```

El while es solo para cuando no conocemos el número de ciclos.

Una forma simplificada (pythonica) de poner contadores en python:

```
i+=1
```

esto es exactamente lo mismo que

```
i=i+1
```

¿Cuando uso for y cuando while?

Las instrucciones `for` y `while` hacen lo mismo aunque el `while` requiere una línea mas.

- Si el número de ciclos es fijo y conocido uso el `for`.
- Cuando el número de ciclos depende de una cantidad que tengo que calcular uso el `while`.

El `for` es mas eficiente computacionalmente que el while.
Sin embargo para cálculos matemáticos largos no son son eficientes, ni el `for` y menos aun el `while`.

Cuando estemos trabajando con arreglos de numpy tengo que evitar los for's y while's

Uso del for con diccionarios

En el caso de dictionarios:

```
clientes={'Nombre':['Laura', 'Eugenia'],'Edad':[25,38]}
for kcliente in clientes:
    print(kcliente) # key del dictionary
    print(clientes[kcliente]) # values de la key
```

Se puede ser explícito y utilizar: `clientes.keys()`

Si quiero “ciclar” sobre los valores directamente:

```
for vcliente in clientes.values():
    print(vcliente) # values del dictionary
```

Ejercicio: Secuencia de Fibonacci

El próximo número en la serie de Fibonacci es la suma de los dos últimos, comenzando por 0 y 1.

Matematicamente esta serie es infinita. Pero por supuesto en el programa vamos a tener que limitar el número de ciclos.
¿ Como detenemos el proceso? O después de un numḿero fijo de ciclos o cuando el término de la serie llega a un valor máximo deseado.

Respuesta: Secuencia de Fibonacci

```
nciclos = 10
num1 = 0
num2 = 1

for i in range(nciclos):
    print(num2, end=" ")
    num1,num2 = num2, num1+num2
```

```
1 1 2 3 5 8 13 21 34 55
```

```
ntot = 100
num1 = 0
num2 = 1

while num2 < ntot:
    print(num2, end=" ")
    num1,num2 = num2, num1+num2
```

```
1 1 2 3 5 8 13 21 34 55 89
```

Notar el uso de la tupla. Pythonico!

Evita el uso de una tercera variable para intercambiar valores:

```
a,b = b, a
```

```
temp=a
a=b
b=temp
```

Puede utilizarse para asignar múltiples variables a la vez:

```
pi,tk,g = 3.14,-273.15,9.80
```

Ejercicio. Tablas de multiplicación.

Queremos que un estudiante de la primaria practique las tablas de multiplicación. Preguntamos cuanto es una multiplicación de dos números y luego debemos chequear si el resultado es el correcto.

Respuesta Tablas de multiplicación. Ejemplo bucles anidados con for.

```
ierror=0
for i  in range(1,10):
    for j in range(1,10):
        cadena='Cuanto es:  '+str(i)+'x'+str(j)+' ? '
         res=int ( input(cadena) )
         if (res != i*j):
            ierror+=1
            print 'Has cometido ',ierror,' errores'

if  (ierror < 3):
    print 'Te felicito. Podes ir a jugar'
else:
    print 'Te quedaste sin futbol.'
```

Tenemos una cantidad de 100 ciclos en total!

Bucles anidados. Ejemplo while-for

```
import random
retry=True
ierror=0
while  retry:
    for i  in range(10):
         j=int(random.uniform(1,10))  # genera numeros aleatorios flotantes
                                                [1,10)
         k=int(random.uniform(1,10))
         cadena='Cuanto es:  '+str(j)+'x'+str(k)+' ? '
         res=int ( input(cadena) )
         if (res != i*j):
            ierror+=1
            print 'Has cometido ',ierror,' errores'

    if  (ierror < 3):
        print ('Te felicito. Podes ir a jugar')
        retry=False
    else:
        print 'Intentalo nuevamente.'
```

¿Cuantos ciclos tenemos ahora?

Enumerate

Hay veces que ademas del ciclado en la lista necesitamos tener un índice del número de ciclo:

```
enumerate(secuencia, start=0)
```

Esta función da dos salidas, el índice de la secuencia y su elemento.

Supongamos que a una lista de flotantes queremos multiplicar/adicionar el cuadrado de su ubicación:

```
nros=[5,10,21,57]
res=[]
sum=0
for i,nro in enumerate(nros):
    sum+=nro * i**2
    res.append(nro+i**2)
```

Pythonico

```
nros=[5,10,21,57]
res=[]
sum=0
for i in range(nros):
    sum+=nros[i] * i**2
    res.append(nro[i]+i**2)
```

No tan lindo.

El enumerate es un caso particular de los generadores que veremos en la próxima clase.

Ejercicio: Transformación de un número binario a decimal

Queremos transformar con un número binario ingresado en un número decimal en base 10 y controlando la entrada.
La fórmula para hacer la transformación:

$$n_d = \sum_{i=0}^{n-1} \text{dig\_bin}\, 2^i$$

Rta. Transformación de un número binario a decimal

```
l_no_bin=True
while (l_no_bin):
    nro_binario = input('Introduzca el numero binario: ')
    nro_decimal=0
    for i,digito_bin  in enumerate(nro_binario[::-1]):
        if (digito_bin=='0' or digito_bin=='1'):
            nro_decimal += int(digito_bin) * 2**i
        else:
            print('No es un binario. Reintente')
            break
    if (i == len(nro_binario)-1): # ingreso bien el numero salgo del while
        l_no_bin = False

print(f'El nro binario {nro_binario} corresponde a {nro_decimal}')
```

zip

`zip` genera una lista de tuplas a partir de un conjunto de listas.

```
>>> print(list(zip('abcdefgh',range(5))))
[('a', 0), ('b', 1), ('c', 2), ('d', 3), ('e', 4)]
```

La lista de tuplas termina cuando se acaba una de las listas.

Una aplicación es el for con múltiples listas:

```
for nombre,direccion in zip(nombres,direcciones):
    print(nombre,'vive en: ',direccion)
```

Puedo trabajar con tres listas (o n-listas) a la vez:

```
  res = list(zip('abcdefgh',range(5),range(2,10,2)))
```

Desempaquetado de tuplas o listas.

El intercambio/swaping de valores de variables, es un ejemplo de desempaquetado.

```
a,b = b, a
```

```
a,b = (b, a)
```

Si tengo una lista o tupla y las quiero desempaquetar uso el "*" asterisco:

```
>>>  a=[5,7,8]
>>>  print(*a)
5 7 8
```

```
>>>  a=(5,7,8)
>>>  print(*a)
5 7 8
```

Dejaron de ser una lista/tupla y pasaron a ser tres elementos!

Supongamos que tenemos una lista de tuplas y queremos generar tuplas de los elementos de las tuplas. Ejemplo: tengo una lista de tuplas con los nombres y domicilios y quiero tener una tupla de los nombres y otra de los domicilios.

```
>>>  nombres_y_domicilios = [('Juan','Belgrano 122'),('Pedro','San Juan 2820')
                                        ,('Alejandro','Rivadavia 22')]
>>>  nombres,domicilios = zip(*nombres_y_domicilios)
>>>  print(nombres)
 ('Juan', 'Pedro', 'Alejandro')
```

Un uso significativo del desempaquetado es con argumentos de funciones que lo veremos en la próxima clase.

Ordenando listas

Si queremos ordenar a los items de una lista en forma ascendente.

- Si lo que quiero es reemplazar la lista con la lista ordenada uso `lista.sort()` .
- Si lo que quiero es generar una nueva lista con los elementos ordenados `lista_ordenada=sorted(lista)`

Retorna una lista ordenada:

```
>>> notas=[5,4,8,7,3]
>>> notas_ordenadas = sorted(notas)
>>> print( notas_ordenadas )
 [3, 4, 5, 7, 8]
```

Ordena y reemplaza la lista:

```
>>> notas=[5,4,8,7,3]
>>> notas.sort()
>>> print ( notas )
 [3, 4, 5, 7, 8]
```

Si se quiere puedo ordenarlas en forma descendente:

```
>>> notas.sort(reverse=True)
```

El comando también funciona con cadenas:

```
lista=['a','z','o','d','p']
```

Estos comandos son de utilidad cuando se requiere el uso del `for` en la lista.

Quiero listar los nombres ordenados alfabeticamente:

```
for nombre in nombres.sort():
    print(nombre)
```

Comprensión de listas

Genero listas a través de un for en una línea:

```
>>> cuadrados_nros = [x**2 for x in range(5)]
```

Los corchetes son porque estoy generando una nueva lista. Los elementos de la lista son lo que siguen y el for es lo que define las iteraciones.

Puedo combinar un if también con el for:

```
>>> impares_nros = [x for x in range(5) if x % 2 == 0]
```

Puedo usar también for anidados:

```
>>> tablero = [(x,y) for x in range(10) for y in range(10)]
```

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).

Las funciones deben ir antes de llamarlas.

Llamada a la función.

- Los argumentos de entrada posicionales son obligatorios y su orden esta pre-establecido.
- Los argumentos de entrada nominales son opcionales, podemos intercambiar el orden y tienen definido un valor por default (en el caso de que no se llame con ese argumento nominal).

```
x,y,z=5,6,7
vv1,vv2 = funcion_3Dgral(x,y,z)
vv1b,vv2b = funcion_3Dgral(x,y,z,escala=3)
```

Podemos tener funciones sin argumentos de entrada y/o sin argumentos de salida:

```
def aviso():
    print('problemas en el codigo')
```

En ese caso la llamo directamente: `aviso()`

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)
```

Notar todas las formas que tenemos para “llamar” a la función. En esencia es como cualquier otra variable.

- 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.

Función en una sola línea.

Función en una línea:

```
  funcion = lambda x: math.sin(x**2)/x
```

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

Con varios argumentos de entrada:

```
superficie = lambda x,y: math.sin(x*x+y*y)
```

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...