Introducción al lenguage Python

Temario de la clase

- Características de python.
- Python como una calculadora.
- Librerías.
- Variables. Tipos de variables
- Condicionales: `if`
- Loops. `for, while`

Que es Python?

- Lenguaje de muy alto nivel (sintaxis comprensible y muy sencilla).
- Lenguaje interprete (No necesita de compilación).
- Lenguaje estructurado. La tabulación es parte de la sintaxis.
- Lenguaje orientado a objetos.

Aplicaciones

Lenguaje utilizado en la mayoría de las aplicaciones desarrolladas por Google.

Youtube esta hecho en python.

Lenguaje utilizado por Industrial Light & Magic en la producción de Star Wars y por DreamWorks Animations.

Bibliografía para el módulo

Tutorial oficial de python:
https://docs.python.org/3/tutorial/index.html

Libros para una introducción:

- Langtangen, H.P., 2016. A primer on scientific programming with Python. Springer-Verlag Berlin Heidelberg.
- Zhang Y. An Introduction to Python and computer programming. Springer Singapore; 2015.

Avanzados:

- Ramalho, L., 2022. Fluent python. O'Reilly Media, Inc..
- Martelli, A., Ravenscroft, A.M., Holden, S. and McGuire, P., 2023. Python in a Nutshell. O'Reilly Media, Inc.

Porque Python?

- Programas muy compactos (3-4 veces mas corto que en fortran o C).
- Programas legibles.
- Lenguaje estructurado y orientado a objetos.
- Es un lenguaje open-source (gnu).
- Muy fácil /de integrar con/integrador de/ otros lenguajes C/Fortran/Java.
- Enorme comunidad de usuarios.

Es el lenguaje dominante para aplicaciones de machine learning .

¿Que es ser pythonico? Zen of python: 10 Mandamientos de python

```
>>> import this
```

Como usar python

- Versiones recomendadas: python 3.8 o superior.
- Anaconda es una plataforma python completa (demasiado) para data science
- Fácil instalación de librerías individuales: `pip`

En una terminal shell/bash. Desde línea de comando:

```
$ python
```

Si se quiere ejecutar en forma remota (en otra computadora)

```
$ ssh usuario@computadora
```

```
$ ssh usuario@10.40.60.207
```

Luego en la terminal remota se puede ejecutar el python:

```
$ python
```

Interprete estandard.

```
$ ipython
```

Interprete mejorado.

Python como una calculadora

Operaciones aritméticas

```
>>> 55+15
>>> -33+12
>>> 5.2/3.1 + 2
>>> 4**2
>>> 4**0.5
```

Orden de las operaciones aritméticas: 1. $**$, 2. $*, /$, 3. $+, -$. Se altera con $()$

```
>>> 5*3**2
>>> (5*3)**2
>>> 5+3**2
>>> (5+3)**2
```

Tipo de variables

```
>>> int(5)
>>> 5/2.0
>>> 5//2
>>> 'Hola'
>>> lpaso=True
```

Tipos de varibles: Enteros. Flotantes. Cadena de caracteres. Lógicas.

Números complejos

```
>>> a=5+2j
>>> type(a)
>>> print a.real
>>> print a.imag
```

Transforma un número flotante/entero en complejo:

```
>>> complex(5.0)
```

Primer programa python

Si al código lo tenemos que guardar en un archivo que denominamos "programa" o "script".

Los nombres de los archivos python tienen extensión .py.

Para hacer un programa se usa cualquier editor: emacs, vi, o nano.

Editemos un archivo:

```
$ nano simple.py
```

```
#!/usr/bin/env python3
pi=3.141592
radio=20.0
per=2*pi*radio
sup=pi*radio**2
print ('El perimetro es: ',per)
print ('La superficie es: ',sup)
```

Ejecución de un programa python

Para ejecutar un programa python lo que hacemos en una terminal shell/bash es:

```
$ python simple.py
```

Si el programa se encuentra en otro directorio se le da el camino completo:

```
$ python /home/pulido/curso/ml/pyt/simple.py
```

Para hacer el programa ejecutable, hacer

```
$ chmod +x simple.py
```

luego:

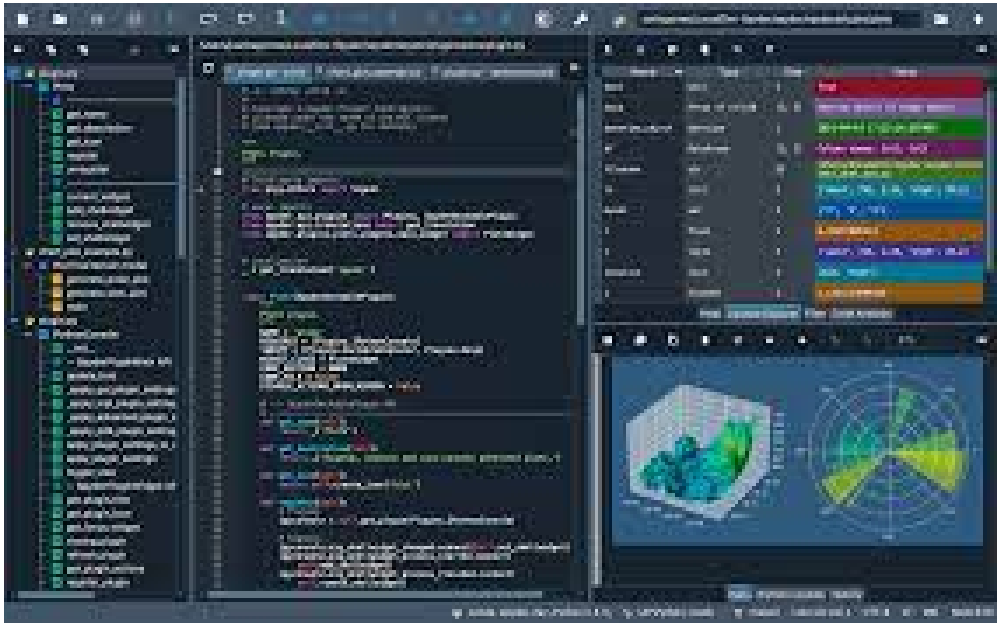
```
$ ./simple.py
```

Para esto es importante incluir como primera línea:

```
#!/usr/bin/env python3
```

Entornos de trabajo

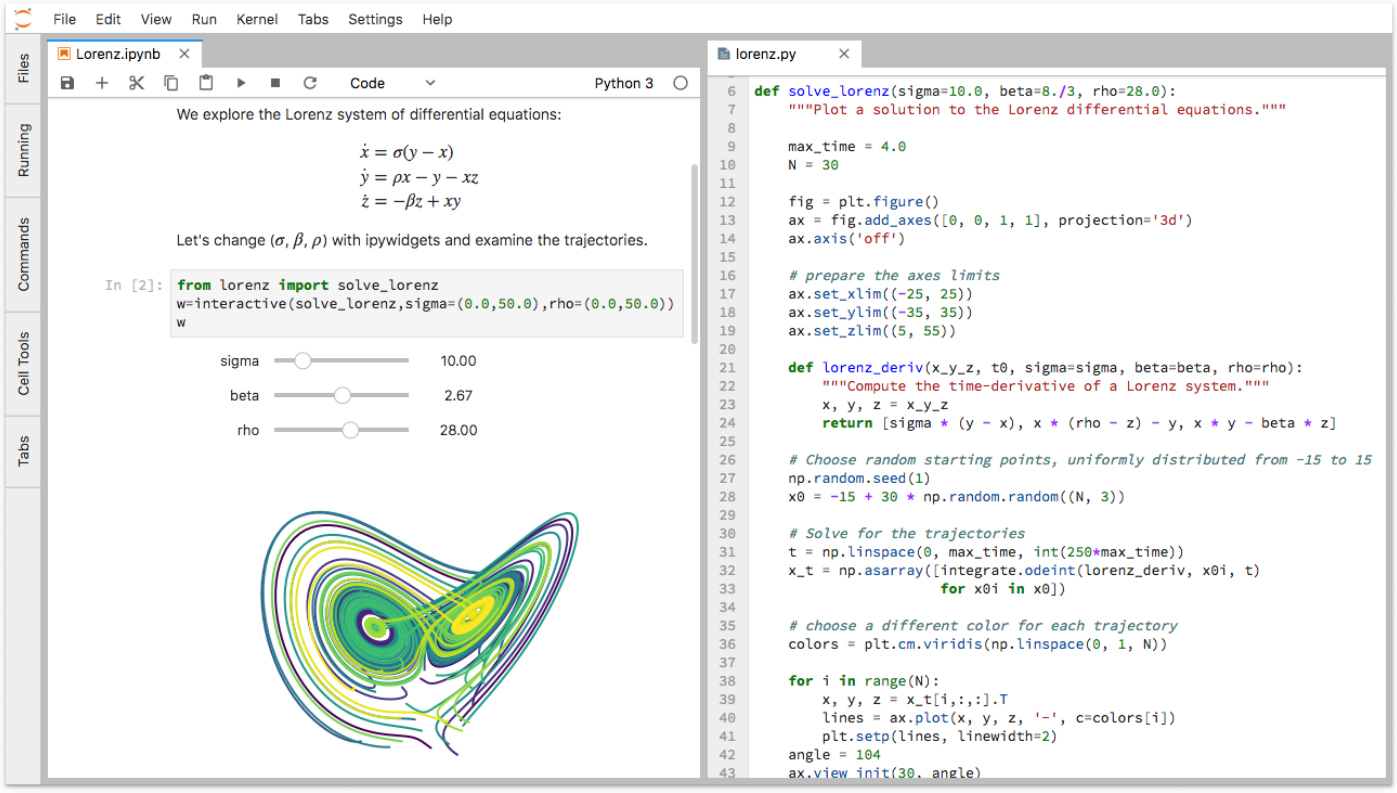
Integrated developed enviroment IDE: editor de textos, interprete/builder, debugger, visualizacion



Open source: spyder.
Disponible en repositorio python / anaconda.
Propietary: pycharm, visual studio code (vsc)

Mas primitivo:emacs + flycheck + PEP8 + elpy
https://www.emacswiki.org

Cuadernos de trabajo



jupyter notebook: web IDE con codigo + markdown

https://jupyter.org/

No sirve para produccion! Solo para testing/análisis.

Alternativa nerd: emacs + org-babel (literate programming multilanguage)

Link para mas detalles

Literate programming

Comentarios en el programa:

- Para recordar lo que hicimos (nosotros mismos).
- Para explicar lo que hacemos en el código a otro programador.
- Para referenciar el objetivo, que es lo que hace, cuando lo hicimos, cuando lo modificamos, que cosas necesitamos agregarle, etc.

REGLA DE ORO 1: Es esencial que todo programa este comentado hasta el último detalle.

- Los comentarios de una sola línea se hacen con #:

```
# (todo lo que sigue detrás del símbolo python lo interpretará como un #comentario)
```

```
5+8 # suma
```

- Comentarios generales de una línea “hola”
- Comentarios de varias líneas se hacen con: """ (triple comillas)

Estos últimos son utilizados para armar el “help/manual” de las librerías

Comentarios generales de un código

```
""" Calcula el
perimetro y la superficie de una circunferencia.
MP. [2016-08-10]
TODO. Agregar el volumen de una esfera
"""
pi=3.141592
radio=20.0
# Calculo del perimetro
per=2*pi*radio
sup=pi*radio**2 # superficie
print ('El perimetro es: ',perim)
print ('La superficie es: ',sup)
```

REGLA DE ORO 2: Las variables deben tener nombres representativos de la información que contiene e.g. sup (guarda la superficie). Mejor minúsculas.

Entrada de valores

Ejercicio: queremos diseñar un programa para estudiantes de la primaria (que no saben programar) que midan el diámetro de distintos objetos circulares y que el programa les calcule el perímetro y la superficie.
El programa debería pedir que introduzcan el valor del diámetro.
El comando que detendrá la ejecución y pedirá para que el usuario ingrese una cantidad es el input, en general el valor que se ingrese se guardará en una variable:

```
d=input("introduzca el diametro del objeto: ")
```

Cualquier valor ingresado numérico o caracter será guardado como cadena de caracteres (str).

```
In [9]: type(d)
Out[9]: str
```

Entrada de valores

Entonces el programita para los estudiantes de primaria con el input sería:

```
pi=3.141592
d=float(input("introduzca el diametro del objeto: "))
# Calculo del perimetro
per=pi*d
sup=pi*(d/2.)**2 # superficie
print ('El perimetro del objeto es: ',per)
print ('La superficie del objeto es: ',sup)
```

A este programa le faltan chequeos para cuando el usuario introduce algo erroneamente. E.g. el diámetro debe ser positivo.

Print con formato de salida

Supongamos que tenemos un examen y queremos informar los estudiantes presentes

```
print ('La cantidad de asistentes al examen fue de %d alumnos de un curso de %d alumnos. Los ausentes fueron %d' %(nasis,ncurso,ncurso-nasis))
```

Las notas se deben informar con uno o dos dígitos pero el promedio con dos decimales.

```
print ('Perez, Juan          %d' %(nota1))
print ('Sanchez, Pedro       %d' %(nota2))
print ('Promedio             %6.2f' %(0.5*(nota1+nota2)) )
```

%d se utiliza para enteros.

%f para flotantes. %(digitos).(decimales)f

%s para una cadena (string).

f-strings

Una forma nueva (python >=3.6) y mucho mas conveniente de imprimir variables con texto son las f-strings:

```
print (f'La cantidad de asistentes al examen fue de {nasis} alumnos.')
print (f'de un curso de {ncurso} alumnos.')
print (f'Los ausentes fueron {ncurso-nasis}')
```

Con formato:

```
print (f'Perez, Juan  {nota1:d}')
print (f'Sanchez, Pedro  {nota2:d}')
print (f'Promedio  {(0.5*(nota1+nota2)):6.2f}' )
```

Interprete: help y exit

Si queremos consultar información de un comando:

```
>>> help(input)
```

```
>>> help(print)
```

Para salir del interprete:

```
>>> quit()
```

Para terminar un programa en el medio:

```
raise SystemExit
```

Alternativa (es lo mismo):

```
sys.exit()
```

Tipos de variables: listas

Una lista es un conjunto de variables de cualquier tipo separados por comas y delimitado por corchetes:

```
>>> lista=[25,60.4,'edad y peso','domicilio']
```

Para acceder a un elemento de la lista:

```
>>> print ( lista[1] )
```

Para acceder a varios elementos de la lista:

```
>>> print ( lista[1:3] )
```

Cantidad de elementos de la lista:

```
>>> print ( 'Longitud: ',len(lista))
```

Si quiero cambiar la edad en la lista:

```
>>> lista[0]=26
```

Tipos de variables: tuplas

Las tuplas son secuencias de objetos como las listas pero no se pueden cambiar (no mutables)

```
>>> tupla=(25,60.4,'edad y peso')
```

si engordo y quiero cambiar el peso:

```
>>> tupla[1]=61.6
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

La razón de su existencia es que son mucho mas eficientes que las listas.

Objetos por todos lados

Las variables son objetos en python

```
a=15.0
dir(a) # metodos disponibles para los flotantes
a.is_integer()
```

Las cadenas de caracteres, las listas y las tuplas son secuencias (y objetos).

- Las secuencias son iterables.
- Las operaciones sobre las secuencias se aplican a todas por igual. Incluso a otros containers (e.g. diccionarios)

Operaciones con listas y strings

Hay un conjunto de funciones “built-in” (nativas) que son aplicables a las secuencias: listas, tuplas y strings:

`len(seq), max(seq), min(seq), sum(seq)`

donde `seq` una secuencia (e.g. lista).

Mapa:

`map(funcion,lista)`: aplica la función `funcion` a todos los elementos de la lista `lista`

```
>>> lista = ['Ricardo', 'RODOLFO', 'sergio' ]
>>> list(map (str.lower,lista))
['ricardo', 'rodolfo', 'sergio']
```

Operaciones con listas

range(<inicio,>fin<,salto>): crea una lista de enteros, desde 0 de uno en uno (el fin esta excluido!).

```
>>> print ( range(4) )
[0, 1, 2, 3]
```

```
>>> print ( range(2,10,2) )
[2, 4, 6, 8]
```

Rebanadas - slicing

`Var[]` Corchetes aplican a secuencias

`Var[<inicio,>fin<,salto>]`

```
a=list(range(10))
```

Las secuencias en python3 tienen "lazy evaluation"

Formas posibles:

```
a[2:5]
a[2:]
a[:5]
a[:-2]
a[::2]
```

Y si hacemos: a[::-2] ?

Operaciones con listas

```
>>> z = []
```
Crea una lista

lista.append(elemento) Agrega elementos a una lista

```
>>> z = [1,2.02]
>>> z.append(800.8)
>>> z
[1, 2.02, 800.8]
```

lista.reverse(): invierte la lista

```
>>> z.reverse()
>>> print ( z )
[800.8, 2.02, 1]
```

lista.remove(x): elimina el primer elemento que coincide con x de la lista

lista.pop(j): elimina el elemento j-esimo de la lista

Operaciones con cadenas de caracteres

Subcadenas [i:j] (slicing de strings):

```
>>> sa='cadena'
>>> print ( sa[2:4] )
de
```

Transforma un número en cadena, str

```
>>> a=1239
>>> b=str(a)
>>> print ( b[2:4] )
39
```

Concatena una cadena: +

```
>>> nombre='Juan'
>>> apellido='Perez'
>>> nombre_completo=nombre+' '+apellido'
>>> print ( nombre_completo )
Juan Perez
```

¿Qué produce con * en las strings? e.g. `a=5*'z'`

Operaciones con cadenas de caracteres

Cambia a mayúsculas: .upper()

```
>>> a=’casa’
>>> print ( a.upper() )
CASA
```

Reemplaza un caracter o cadena de caracteres: .replace(viejo,nuevo)

```
>>> print ( a.replace(’a’,’o’) )
coso
```

Busca un caracter o cadena de caracteres: .find(’o’)

```
>>> a=’casona’
>>> print ( a.find(’o’) )
3
```

Existen mas de 30 métodos para las cadenas, ver con `>>> dir(a)`

Tipos de variables/objetos: Diccionarios

Almacenan un conjunto de variables u objetos.

- Se accede a sus elementos a partir de claves principales (keys).
- La clave sirve para identificar al elemento.
- Se definen con llaves (en lugar de corchetes que es para las listas).

```
>>> x={'nom','dom'}
```

Asignación:

```
>>> x={'nom':'Sergio','dom':'Libertad 5400'}
```

```
>>> x['nom']='Sergio'; x['dom']='Libertad 5400'
```

Funciones para diccionarios: `len(), x.keys(), x.values()`

Mas: `>>> a=; dir(a)`

Información sobre el método: `>>> help(a.popitem)`

Uso de librerías

```
>>> import math
>>> math.log(5.0)
```

Si queremos renombrar (por ejemplo para que el nombre sea mas corto):

```
>>> import math as m
>>> m.log(5.0)
```

Si queremos importar solo un código que esta dentro de una librería:

```
>>> from math import log,sin
>>> log(5.0)
```

Notar que con esto reconoce el comando directamente. No es aconsejable en programación pero si para hacer pruebas.

Prohibido: `from math import *.`

Librerías específicas

- Para evaluar funciones matemáticas: math, cmath
- Librería cálculos numéricos: numpy wrappers a: BLAS, LAPACK, fft
- Librería de graficación: matplotlib
- Librería científica (integración, EDOs, procesamiento de señales): scipy BLAS, LAPACK, fft, etc
- Librerias de manipulación de datos: pandas
- Lectura y escritura de datos cientificos: hdf5, netcdf
- Librerías de machine learning: scikit-learn, pytorch, tensorflow

Uso de librerías. os

Si queremos ejecutar algun comando de terminal shell o bash, esta la librería os.

```
>>> import os
>>> os.system('ls')
```

- Si queremos chequear si existe un archivo: isfile

```
>>> import os
>>> os.path.isfile('prueba.tex')
```

- Si queremos chequear si existe un directorio o un archivo: exists

```
>>> os.path.exists('prueba.tex')
```

- Si queremos chequear si existe un directorio: isdir

```
>>> os.path.isdir('/home/programacion/readme')
```

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 codigo tenga determinadas caracteristicas o no de acuerdo al interesado.

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
```

Operadores que resultan en variables lógicas

Es la variable a igual a la variable b? ==

```
>>> 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 (en muchos contextos es similar a ==, pero mas pythonic porque es legigle).

```
>>> x0 is 5
>>> x0 is None
```

Las variables las puedo definir como 'None'

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=raw_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 end)

La instrucción if-else

Si pasa esto, haga algo si no pasa eso haga 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

Varias opciones 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')
```

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”.

Ejemplo. Encontrar las raíces de una ecuación cuadrática

Describa un procedimiento mediante expresiones matemáticas y pasos a seguir para obtener los ceros de una función cuadrática.

```
print ('Determina raices reales de a x^2 + b x + c = 0')
a=float(input('Introduzca a: ')) # Idem b y c

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/ciclos/loops: con while.

El comando while hace que la computadora repita una serie de órdenes hasta que se cumpla una condición lógica.
Para producir un bucle hasta que se cumpla la condicion:

```
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
```

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 )
```

Elementos e indice:

```
a=['a','b','c']
for i,car in enumerate(a):
        print (i,char,')')
```

Dos listas:

```
for nombre,direccion in
            zip(nombres,direcciones):
```

Bucles con for

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

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

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

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
```

Si quiero “ciclar” sobre los valores directamente:

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

Bucles anidados. Ejemplo

Queremos que un estudiante de la primaria, Manuel mi hijo, practique las tablas de multiplicación.

```
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.'
```

Bucles anidados. Ejemplo papá bueno

```
import random
retry=True
ierror=0
while  retry:
    for i  in range(10):
         j=int(random.uniform(1,10))   # random.normal(
         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.'
```

Transformación de un número binario a decimal

Queremos transformar con un numero binario ingresado y controlando la entrada.

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

```
l_no_bin=True
continue_again = 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):
        l_no_bin = False

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

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.

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.

Funcion en una línea: `funcion =lambda x:math.sin(x**2)/x`

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

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? ¿C'ual 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?