

INTRODUCCIÓN A PYTHON 3

**Universidad Nacional de
Colombia**

**Material de apoyo elaborado como apoyo a la
materia**

Métodos Numéricos

Diego Camilo Peña Ramírez (docente)

Twitter: @nervencid

CONTENIDO

- ¿Que es Python?
- Historia de Python
- Características de Python
- EMPEZAR A UTILIZAR PYTHON
- Aplicación básica “Hola mundo”
- COMENTARIOS
- VARIABLES
- TIPOS DE DATOS
- OBTENER EL TIPO DE VARIABLE
- CONVERSIONES
- COLECCIONES DE TIPOS DE DATOS

CONTENIDO

- DICCIONARIOS
- OBTENER UN CARÁCTER DE UNA CADENA
- OPERACIONES ARITMETICAS
- OPERACIONES LOGICAS
- OPERADOR DE PERTENENCIA 'in'
- SENTENCIA 'for'
- SENTENCIA 'while'
- SENTENCIA 'if'
- FUNCIONES

CONTENIDO

- FUNCIONES DE ORDEN SUPERIOR
- FUNCIONES LAMBDA
- COMPRENSIÓN DE LISTAS
- GENERADORES
- DECORADORES
- MANEJO DE ARCHIVOS
- ESCRIBIR TABLAS EN PYTHON
- EXCEPCIÓNES Y ERRORES
- PROGRAMACIÓN ORIENTADA A OBJETOS

CONTENIDO

- PROGRAMACIÓN ORIENTADA A OBJETOS (HERENCIA)
- PROGRAMACIÓN ORIENTADA A OBJETOS (CLASES DECORADORAS)
- BIBLIOGRAFIA
- SOBRE EL AUTOR Y EL CONTENIDO

¿Que es Pyhton?

Python es un lenguaje de programación interpretado cuya filosofía hace hincapié en una sintaxis muy limpia y que favorezca un código legible.[1]



Historia de Python



Imagen extraída de [1]

Python fue creado a finales de los ochenta² por Guido van Rossum en el Centro para las Matemáticas y la Informática (CWI, Centrum Wiskunde & Informatica), en los Países Bajos, como un sucesor del lenguaje de programación ABC, capaz de manejar excepciones e interactuar con el sistema operativo Amoeba.³
[1]

Historia de Pyhton

El nombre del lenguaje proviene de la afición de su creador original, Guido van Rossum, por los humoristas británicos Monty Python. [1]

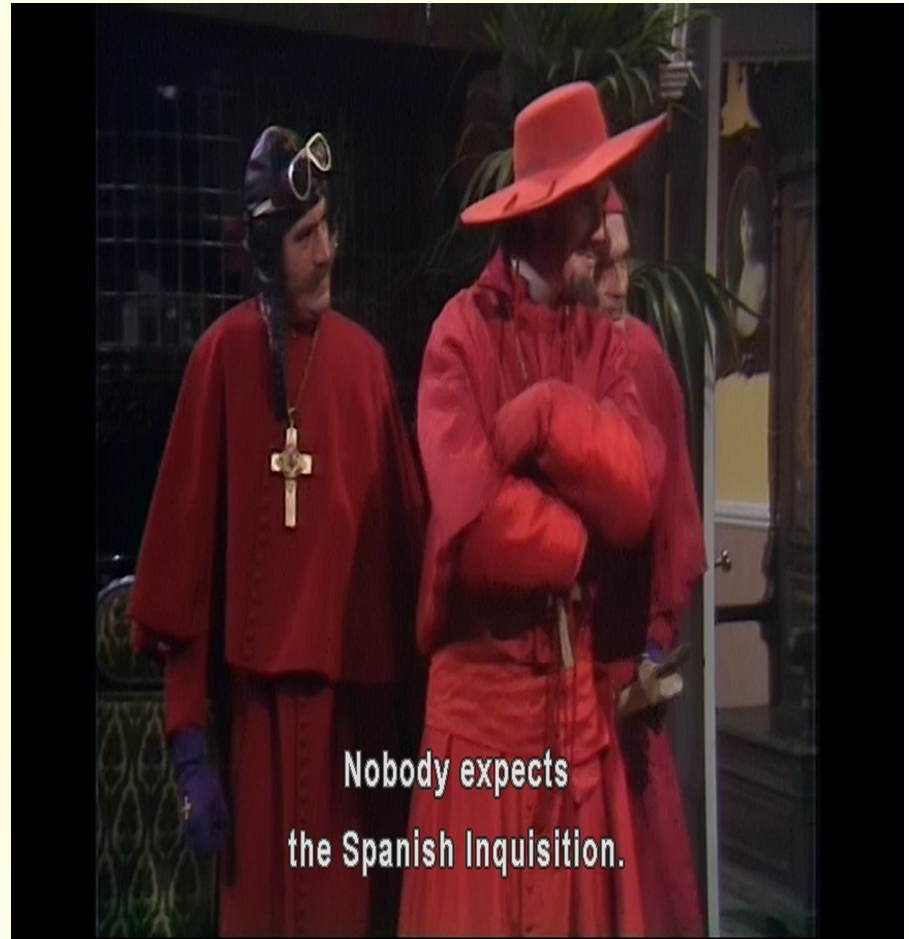


Imagen extraída de [2]

Características de Python

- Simple
- Sencillo de Aprender
- Libre y Fuente Abierta
- Indentado
- Lenguaje de Alto Nivel
- Portable
- Interpretado
- Orientado a Objetos

Características de Python

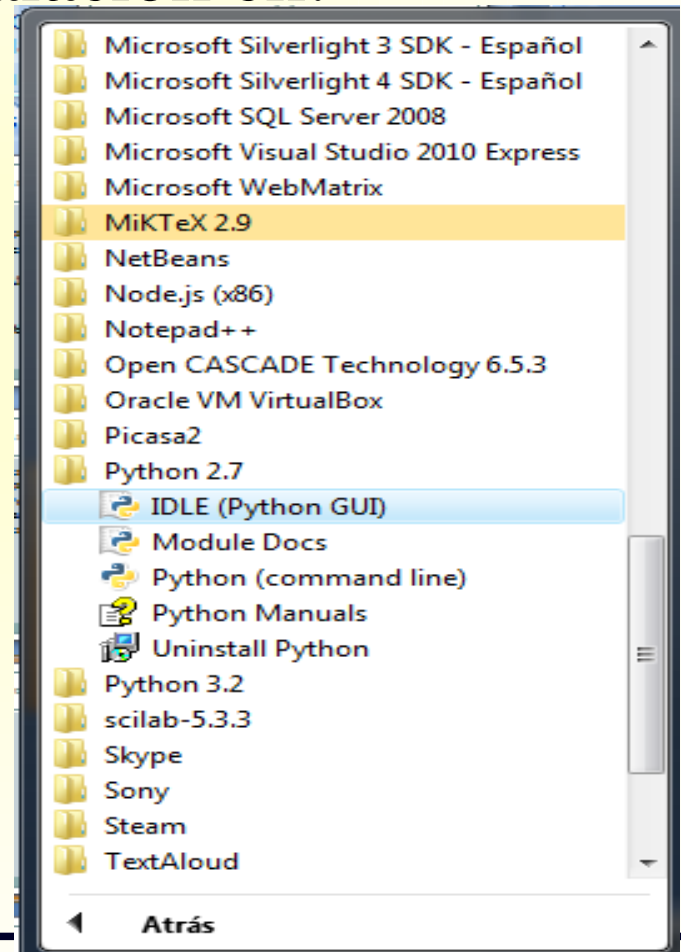
- Ampliable (Permite combinar fragmentos con otros lenguajes de programación).
- Incrustable (Permite insertar código en otros lenguajes para dar facilidades de scripting).
- Librerías Extendidas

EMPEZAR A UTILIZAR PYTHON

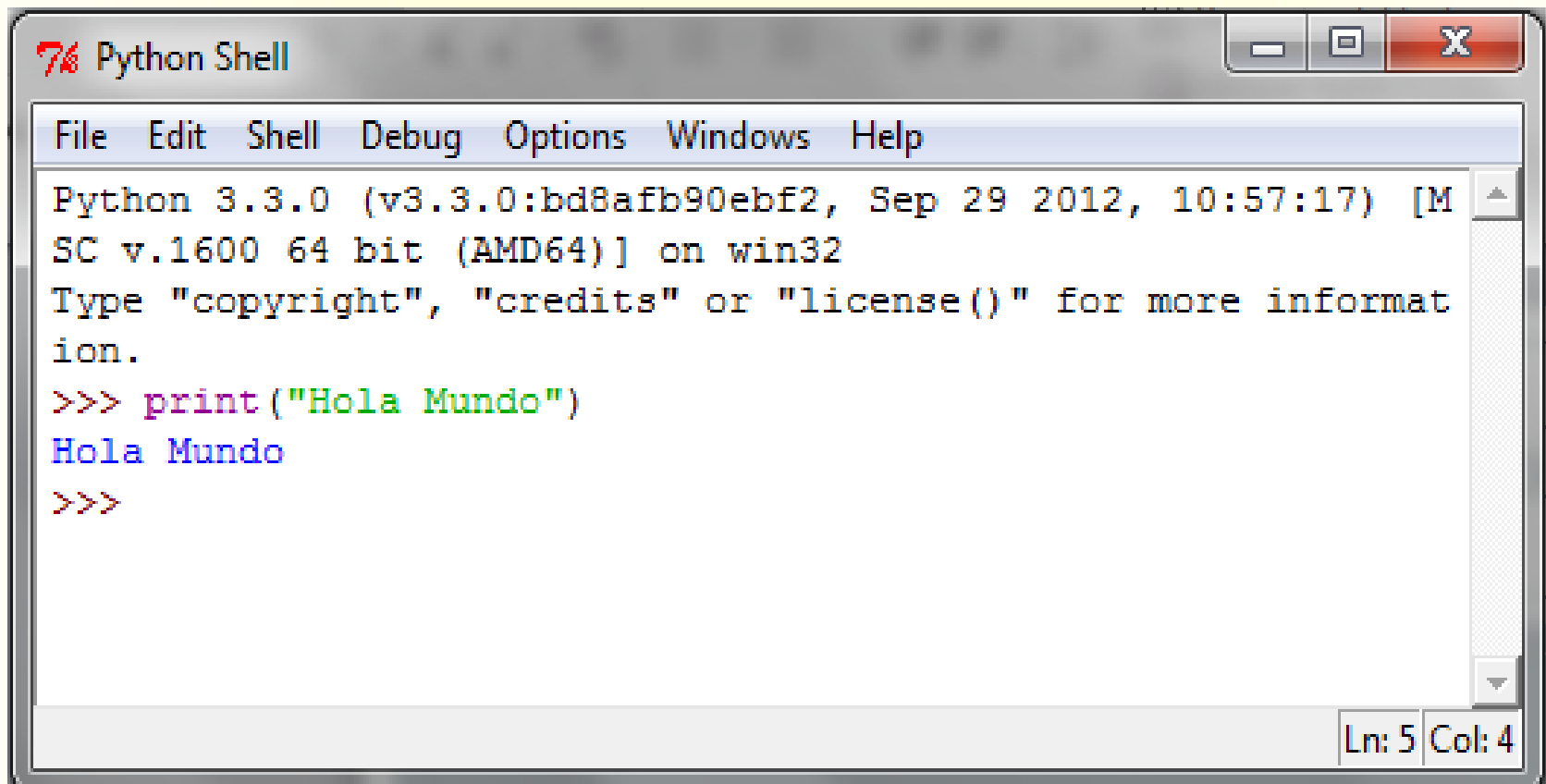
- Seguir las instrucciones de instalación en:

<http://www.python.org/getit/>

- Iniciar el Shell de Python



Aplicación básica “Hola mundo”



The image shows a screenshot of a Python Shell window. The title bar reads "Python Shell" with standard Windows window controls (minimize, maximize, close). The menu bar includes "File", "Edit", "Shell", "Debug", "Options", "Windows", and "Help". The main text area displays the following content:

```
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 10:57:17) [MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.

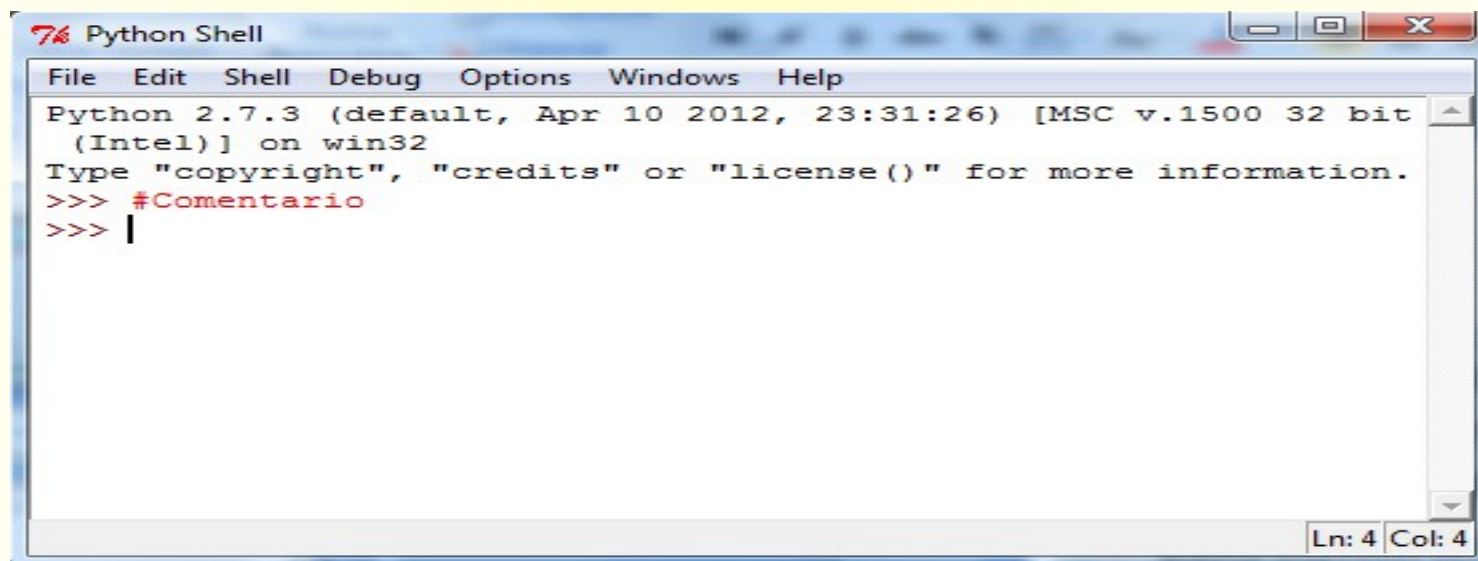
>>> print("Hola Mundo")
Hola Mundo
>>>
```

The status bar at the bottom right indicates "Ln: 5 Col: 4".

**¡SI!, así de
sencillo es**

COMENTARIOS

- Antes que nada los comentarios están marcados con el signo # esto es importante para poder entender el código fuente del programa en un futuro.
- Los comentarios NO se ejecutan y son ignorados por el interprete de Python.



The screenshot shows a window titled "Python Shell" with a menu bar (File, Edit, Shell, Debug, Options, Windows, Help). The text inside the window reads: "Python 2.7.3 (default, Apr 10 2012, 23:31:26) [MSC v.1500 32 bit (Intel)] on win32", "Type 'copyright', 'credits' or 'license()' for more information.", followed by a prompt ">>>". The user has entered "#Comentario" on the next line, and the prompt ">>>|" is shown on the following line, indicating the cursor is at the end of the line. The status bar at the bottom right shows "Ln: 4 Col: 4".

```
Python 2.7.3 (default, Apr 10 2012, 23:31:26) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> #Comentario
>>> |
```

VARIABLES

- Declaración:

```
>>> variable = 10
```

```
>>> variable=1+2j
```

```
>>> variable="Metodos numericos"
```

```
>>> variable='Metodos Numericos'
```

VARIABLES

PYTHON ve a las variables como objetos, y cuando le asignamos un nombre a una variable es algo así como asignarle una etiqueta que apuntara a una dirección de memoria donde se encuentra almacenado el objeto.

Cuando cambiamos el valor de dicha variable lo que hacemos es que la etiqueta apunte a otra dirección de memoria de dicho objeto.

Cuando varias variables tienen asignado el mismo valor la etiqueta en realidad apuntara a la misma dirección de memoria.

VARIABLES

```
>>> a='uno'  
>>> b='dos'  
>>> c='tres'
```

```
>>> id(a),id(b),id(c)  
(54143552, 53803752, 54141424)
```

VARIABLES

```
>>> a=5
>>> b=6
>>> c=7
>>> id(a), id(b), id(c)
(505894368, 505894384, 505894400)
>>> a=b=c
>>> id(a), id(b), id(c)
(505894400, 505894400, 505894400)
>>>
```

TIPOS DE DATOS

Tipo	Clase	Notas	Ejemplo
<code>str</code>	Cadena	Inmutable	<code>'Cadena'</code>
<code>unicode</code>	Cadena	Versión <code>Unicode</code> de <code>str</code>	<code>u'Cadena'</code>
<code>list</code>	Secuencia	Mutable, puede contener objetos de diversos tipos	<code>[4.0, 'Cadena', True]</code>
<code>tuple</code>	Secuencia	Inmutable, puede contener objetos de diversos tipos	<code>(4.0, 'Cadena', True)</code>
<code>set</code>	Conjunto	Mutable, sin orden, no contiene duplicados	<code>set([4.0, 'Cadena', True])</code>
<code>frozenset</code>	Conjunto	Inmutable, sin orden, no contiene duplicados	<code>frozenset([4.0, 'Cadena', True])</code>
<code>dict</code>	Mapping	Grupo de pares clave:valor	<code>{'key1': 1.0, 'key2': False}</code>
<code>int</code>	Número entero	Precisión fija, convertido en <i>long</i> en caso de overflow.	<code>42</code>
<code>long</code>	Número entero	Precisión arbitraria	<code>42L</code> ó <code>456966786151987643L</code>
<code>float</code>	Número decimal	Coma flotante de doble precisión	<code>3.1415927</code>
<code>bool</code>	Booleano	Valor booleano verdadero o falso	<code>True</code> o <code>False</code>

Tabla extraída de [1]

TIPOS DE DATOS

- Aunque NO es necesario declarar el tipo de variable, Python SI reconoce automáticamente que tipo de variable es. También podemos operar números complejos y números enteros (o de tipo 'float'), sin ningún problema.
- SIN EMBARGO no podemos hacer cosas como por ejemplo operar cadenas 'string' con números.

OBTENER EL TIPO DE LA VARIABLE

```
>>> variable=1+2j  
>>> type(variable)  
<type 'complex'>
```

CONVERSIONES

```
>>> variable='45'
```

```
>>> variable+1
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#7>", line 1, in <module>
    variable+1
```

```
TypeError: cannot concatenate 'str' and 'int' objects
```

```
>>> int(variable)
```

```
45
```

```
>>> variable+1
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#9>", line 1, in <module>
    variable+1
```

```
TypeError: cannot concatenate 'str' and 'int' objects
```

```
>>> int(variable)+1
```

```
46
```

CONVERSIONES

```
>>> variable='34.5'
```

```
>>> variable-1.5
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#1>", line 1, in <module>
```

```
    variable-1.5
```

```
TypeError: unsupported operand type(s) for -: 'str' and 'float'
```

```
>>> float(variable)-1.5
```

```
33.0
```

CONVERSIONES

```
>>> variable='2+5j'
>>> variable-2j
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    variable-2j
TypeError: unsupported operand type(s) for -: 'str' and 'complex'
>>> complex(variable)-2j
(2+3j)
```


CONVERSIONES

```
>>> variable='7+2.5j'
```

```
>>> variable+0.5j
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#12>", line 1, in <module>
```

```
    variable+0.5j
```

```
TypeError: Can't convert 'complex' object to str implicitly
```

```
>>> eval(variable)+1.5j
```

```
(7+4j)
```

COLECCIONES DE TIPOS DE DATOS

```
#Coleccion de tipos de datos
```

```
#Albergan elementos de cualquier tipo de datos!! Como las CELLS de MATLAB
```

```
#Existen dos tipos de colecciones: Tuplas y Listas
```

```
#Tuplas: No se pueden cambiar los elementos una vez creadas
```

```
#Se crean usando parentesis()
```

```
tupla=(1, 'casa', '3', 4+2j)
```

```
print(tupla)
```

```
#Listas: Se pueden modificar despues de creadas
```

```
#Se crean usando corchetes
```

```
Lista=[1, 2 , 'alpha', 5+2j]
```

```
print(Lista)
```

COLECCIONES DE TIPOS DE DATOS

```
#Se pueden anidar las tuplas con las listas y viceversa
```

```
Lista=[1,2,'alpha', 5+2j, tupla]  
print(Lista)
```

```
#Obtener e imprimir la cantidad de elementos dentro de la Lista  
# Si hay Tuplas anidadas solo cuentan por un elemento  
print(len(Lista))
```

```
#Agregar mas elementos a la lista  
Lista.append('Adicional')  
print(Lista)
```

```
#Remover elemento de la lista
```

```
Lista.remove('Adicional')  
print(Lista)
```

COLECCIONES DE TIPOS DE DATOS

```
#Remover elemento de la lista
```

```
Lista.remove('Adicional')  
print(Lista)
```

```
#Imprimir el elemento [0] de la lista
```

```
print(Lista[0])
```

```
#Reemplazar elemento [3] por otro
```

```
Lista[3]=3+7j  
print(Lista)
```

COLECCIONES DE TIPOS DE DATOS

```
#Agregar un elemento en una posicion determinada
Lista.insert(4, 'Agregado')
print(Lista)
```

```
#Quitar un elemento de una posicion determinada
Lista.pop(4)
print(Lista)
```

```
#Obtenemos la posicion de un elemento
print(Lista.index('alpha'))
```

DICCIONARIOS

```
#Los Diccionarios son una forma mas
#avanzada para manipular colecciones de datos

Diccionario={ 'Nombre': 'Diego', 'Codigo':12345}

#Mostramos el diccionario en pantalla
print(Diccionario)
#Mostramos las llaves
print(Diccionario.keys())
#Mostramos elementos
print(Diccionario.values())
#Acceder a un elemento
print(Diccionario[ 'Nombre' ])
```

OBTENER UN CARÁCTER DE UNA CADENA

```
>>> variable='Metodos Numericos'
```

```
>>> print(variable[0])
```

```
M
```

```
>>> print(variable[-1])
```

```
s
```

```
>>> print(variable[-45])
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#39>", line 1, in <module>
```

```
    print(variable[-45])
```

```
IndexError: string index out of range
```

OPERACIONES ARITMETICAS

```
#Operadores matematicos
```

```
#Suma
```

```
print (1+1)
```

```
#Resta
```

```
print (1-1)
```

```
#Multiplicacion
```

```
print (2*2)
```

```
#Division con coma flotante
```

```
print (4/2)
```


OPERACIONES ARITMETICAS

```
#Division SIN coma flotante
```

```
print(20//2)
```

```
#Residuo de division
```

```
print(15%4)
```

```
#Aumentar un valor a una Lista
```

```
lista=[10 , 20]
```

```
lista+= [5]
```

```
print(lista)
```

OPERACIONES ARITMETICAS

```
#Agregar a listas 1
```

```
lista+='cadena'
```

```
print(lista)
```

```
#agregar a listas 2
```

```
lista+=['cadena']
```

```
print(lista)
```

OPERACIONES LOGICAS

```
#Creacion de listas
```

```
alpha=[1,2,3]
```

```
betha=[1,2,3]
```

```
#Preguntamos si 'alpha' es igual 'betha' Deberia decir FALSE
```

```
#Porque a pesar de que tienen los mismos datos son dos objetos diferentes
```

```
#es decir apuntan a dos direcciones de memoria diferente
```

```
pregunta=alpha is betha
```

```
print(pregunta)
```

```
#Ahora vamos a redireccionar a la misma direccion de memoria
```

```
#Esta vez deberia decir TRUE
```

```
alpha=betha
```

```
pregunta=alpha is betha
```

```
print(pregunta)
```

```
#Ahora preguntamos si 'alpha' NO apunta a NINGUN objeto
```

```
#Nos debe decir FALSE porque en efecto 'alpha' SI esta apuntando
```

```
#a un objeto
```

```
pregunta=alpha is None
```

```
print(pregunta)
```

OPERACIONES LOGICAS

```
#Ahora preguntamos lo contrario y deberia decir TRUE
```

```
pregunta=alpha is not None  
print(pregunta)
```

```
#Ahora asignamos a 'alpha a nada'
```

```
alpha=None  
print(alpha)  
pregunta=alpha is None  
print(pregunta)
```

```
#Notese que aunque 'None' quiere decir que no hay ningun valor  
# 'Alpha' SI tiene 'id'
```

```
print(id(alpha))
```

OPERACIONES LOGICAS

```
#Pregunta AND
```

```
Q=10
```

```
W=20
```

```
print (Q==W)
```

```
E=10
```

```
print (Q==E)
```

OPERACIONES LOGICAS

```
#Pregunta AND negada (diferencia)
```

```
print (Q!=W)
```

```
#Comparacion de magnitud (mayor que. menor que)
```

```
print (Q>W)
```

```
print (Q<E)
```

```
print (Q>1)
```

```
print (Q>=10)
```

```
print (1<Q<2)
```

```
print (0>Q>15)
```

```
print (0<W<100)
```

OPERADOR DE PERTENENCIA 'in'

```
#Declaracion tupla
```

```
tupla=(1, 'alpha', 5, 2+4j)
```

```
#Pregunta si un elemento ESTA dentro de la tupla
```

```
print(2+4j in tupla)
```

OPERADOR DE PERTENENCIA 'in'

#Pregunta si un elemento NO ESTA dentro de la tupla

```
print('alpha' not in tupla)
```

#Pregunta si una palabra esta dentro de una cadena de caracteres

```
print('Diego' in 'Diego usa Python')
```


SENTENCIA 'for'

```
#La sentencia 'For' en Python es diferente a la de C++
```

```
#Dato
```

```
tupla=(0,2+1j,2+3j,'2+3j','alpha')
```

```
#Para un dato x que este en tupla imprime dicho x
```

```
for x in tupla:
```

```
    print(x)
```

```
print('_____')
```

SENTENCIA 'for'

#Cadena

```
cadena='kasldkj;12wgewegw;ao99oaosdoi{} ,adadas@@@#,><>!@#!@#23we'
```

```
vocales=[]
```

```
consonantes=[]
```

```
numeros=[]
```

```
caracteres=[]
```

SENTENCIA 'for'

```
#Este 'For' Buscara las vocales, consonantes, numeros y caracteres en 'cadena'  
#Creara un 'x' que ira recorriendo 'cadena'
```

```
for x in cadena:
```

```
    #pregunta si x es una vocal preguntando si esta en 'aeiou'  
    #Los siguientes 'if' funcionan igual
```

```
    if x in 'aeiou':
```

```
        vocales.append(x)
```

```
    elif x in '0123456789':
```

```
        numeros.append(x)
```

```
    elif x in 'qwertyuiopasdfghjklzxcvbnm':
```

```
        consonantes.append(x)
```

```
    else:
```

```
        caracteres.append(x)
```

SENTENCIA 'for'

```
#Imprime los valores
```

```
print('Las Vocales son:', vocales)
```

```
print('Las Consonantes son:', consonantes)
```

```
print('Los Numeros son:', numeros)
```

```
print('Los Caracteres son:', caracteres)
```

SENTENCIA 'while'

```
#Datos
```

```
a=0
```

```
#Mientras a sea menor de 10 imprima el valor de 'a'
```

```
while a<10:
```

```
    print(a)
```

```
    a+=1 #Le suma 1 a 'a'
```

SENTENCIA 'if'

```
#Pedimos que el usuario introduzca tres datos
```

```
Dato1=input("Introduzca el dato 1: ")
```

```
Dato2=input("Introduzca el dato 2: ")
```

```
Dato3=input("Introduzca el dato 3: ")
```

```
#Sentencia if: Pregunta si el Dato1 es menor a Dato2 y Dato3: Compuerta AND
```

```
if Dato1<Dato2 and Dato3:
```

```
    print('Verdadero AND')
```

```
#OJO CON LA SANGRIA porque puede dar error
```

```
#NO HACERLE sangria al 'if' o al 'else'
```

```
else:
```

```
    print('Falso en AND')
```

```
#Sentencia if: Pregunta si el Dato1 es menor a Dato2 y Dato3: Compuerta OR
```

```
if Dato1<Dato2 or Dato3:
```

```
    print('Verdadero OR')
```

```
else:
```

```
    print('Falso en OR')
```

FUNCIONES

```
#Declaracion de la funcion vacia
```

```
def suma():
```

```
    a=input('Introduzca un dato: ')
```

```
    b=input('Introduzca otro dato: ')
```

```
    #Toca convertir los datos antes de sumarlos
```

```
    print(int(a)+int(b))
```

```
#Declaracion Funcion con argumentos
```

```
def resta(a,b):
```

```
    print(int(a)-int(b))
```

```
#Programa y llamado de funciones
```

```
suma()
```

```
u=input('Introduzca un dato: ')
```

```
v=input('Introduzca otro dato: ')
```

```
resta(u,v)
```

FUNCIONES

Antes de Realizar el próximo ejercicio cree dos archivos uno que se llame “FuncionesExternas.py” y otro que se llame “LlamadoDeFunciones.py”, que estén en LA MISMA CARPETA.

FUNCIONES

En “FuncionesExternas.py” escriba el siguiente código y guarde el archivo:

```
#Declaracion de funciones

#Funcion sin parametros
def funcion1 ():
    #Contenido de la funcion
    print('Has llamado a la funcion 1')

#Funcion con parametros y retorno
def funcion2 (a,b):
    #Contenido de la funcion
    c=int(a)+int(b)
    return c
```

FUNCIONES

En “LlamadoDeFunciones.py” escriba el siguiente código, guarde el archivo y luego ejecute (F5):

```
#Llamamos las funciones declaradas en otro archivo
from FuncionesExternas import *

#Invocamos las funciones
funcion1()
#Invocamos la otra funcion
Q=input(' Introduzca un dato: ')
W=input(' Introduzca otro dato: ')
print (funcion2 (Q,W) )
```

FUNCIONES

Otra forma de llamar funciones:

```
#Llamamos las funciones declaradas en
#otro archivo (Forma alternativa)
import FuncionesExternas as F

#Invocamos las funciones
F.funcion1()
#Invocamos la otra funcion
Q=input(' Introduzca un dato: ')
W=input(' Introduzca otro dato: ')
print(F.funcion2(Q,W))
```

FUNCIONES

Si el archivo que queremos importar NO esta en la misma carpeta, ni en Python usamos 'sys.path.append("../Direccion donde esta el archivo que necesitamos")' para poder usar 'scripts' en otras ubicaciones:

```
#Si no esta el archivo que necesitamos  
#en la misma carpeta
```

```
import sys  
sys.path.append("C:/Documents and Settings/Diegonimus/Mis documentos")
```

FUNCIONES DE ORDEN SUPERIOR

Otra característica interesante de Python es la capacidad de poder pasar como parámetros funciones como si fueran variables, esto nos evita por ejemplo utilizar en algunos casos el uso de molestas sentencias “if”, simplificando y mejorando la presentación de nuestro código

En el ejemplo a continuación (es recomendable crear un nuevo archivo “.py”) trataremos de hacer una calculadora básica (con operaciones suma, resta, multiplicación y división), y partiremos de las siguientes funciones:

FUNCIONES DE ORDEN SUPERIOR

```
#Funciones predeterminadas
def suma(x,y):
    return eval(x)+eval(y)
def resta(x,y):
    return eval(x)-eval(y)
def multiplicacion(x,y):
    return eval(x)*eval(y)
def division(x,y):
    return eval(x)/eval(y)
```

FUNCIONES DE ORDEN SUPERIOR

Creamos una función “calculadora”, donde si observamos con atención el parámetro “f” es una función y se trata como una variable, recordemos que en Python las variables son objetos luego “f”, también es un objeto.

```
#Funcion en donde el parametro 'f' es una funcion
def calculadora(f,x,y):
    return(f(x,y))
```

FUNCIONES DE ORDEN SUPERIOR

Finalmente procedemos a llamar SOLAMENTE la función “calculadora” y pasarle los parámetros “(f,x,y)”. Seguido de esto ejecutamos nuestro archivo

```
#El usuario introduce datos  
a=input("Introduzca un dato: ")  
b=input("Introduzca otra dato: ")
```


FUNCIONES DE ORDEN SUPERIOR

```
#El usuario DEBE escoger la operacion que desea realizar de entre las opciones disponibles:  
#suma, resta, multiplicacion o divisiona  
Operacion=input("Que operacion desea realizar?(suma, resta, multiplicacion o division): ")  
funcion=eval(Operacion)
```

FUNCIONES DE ORDEN SUPERIOR

```
#Mostramos en pantalla  
print("El resultado es: ",calculadora(funcion,a,b))
```

FUNCIONES LAMBDA

Las funciones Lambda, son funciones ANONIMAS que se componen de una sola linea de código. Estas funciones son limitadas ya que NO podemos escribir dentro de estas algunos comandos de código como sentencias “for”, “while”, etc...

```
#Declaracion de la funcion lambda
g=lambda x: x**3

#Llamado de la funcion lambda
print(g(3))

#Declaracion de una lista con funciones lambda
Lista=[lambda x:x**2, lambda x:x**0.5]

#Llamamos a la lista con funciones lambda
for a in Lista:
    print(a(2))
```

FUNCIONES LAMBDA

También podemos hacer algunas operaciones lógicas SIMPLES por ejemplo la siguiente expresión:

```
#Funcion cualquiera
def f(x,y):
    if x<y:
        return x
    else:
        return y
```

Puede ser reemplazada por:

```
#Funcion lambda que reemplaza la funcion anterior
m=lambda x,y: x if x<y else y
```

FUNCIONES LAMBDA

```
#Probamos y ejecutamos, las salidas en ambos casos  
#DEBE SER la misma  
print(f(1,2))  
print(m(1,2))
```

COMPRESION DE LISTAS

Las listas podemos llenarlas de forma automática usando la sentencia “for” o con algún otro bucle como se vio anteriormente, sin embargo es posible simplificar aun más la sintaxis mediante la comprensión de listas, aunque tenemos restricciones similares a las de las funciones lambda, podemos reemplazar una función como la siguiente:

```
#List Comprehension
#Es una forma facil de llenar listas

#Podemos reemplazar la siguiente funcion
L1=[]
for x in range(5):
    L1.append(x**2)

print(L1)
```

COMPRESION DE LISTAS

Por lo siguiente:

```
#List Comprehension
#Es una forma facil de llenar listas

#Ejemplo 1: Llenar una lista con los
#cuadrados de los numeros 0 al 5
L1=[x**2 for x in range(5)]
print(L1)
```

COMPRESION DE LISTAS

Algunos ejemplos adicionales:

```
#Ejemplo 2: Crear una lista "L3" solo  
#con los elementos ENTEROS (int) de "L2"
```

```
L2=[3, 'casa', 4+2j, 5, 'fruta', 3.16, 4, 'Martes']  
L3=[x for x in L2 if type(x) is int]  
  
print(L2)  
print(L3)
```

```
#Ejemplo 3: Crear una lista "L4" solo con los elementos que sean numeros  
#de "L2" sin importar si es int, float o complex
```

```
L4=[x for x in L2 if type(x) is not str]  
print(L4)
```


COMPRESION DE LISTAS

```
#Ejemplo 4: Crear una lista "L4" solo con los elementos que sean numeros  
#de "L2" float o arreglos 'str'
```

```
#Ejercicio en clase
```

GENERADORES

Veíamos en el ejemplo anterior que podíamos crear listas de forma automática con una sintaxis muy breve, ahora veremos como generar los valores de uno en uno, sin necesidad de crear una lista, a esto los llamamos generadores.

Los cuales son funciones que usan la palabra reservada “yield” y generar cada valor en secuencia cada vez que se llame la función “next()”.

GENERADORES

La primera opción de sintaxis es similar a la comprensión de listas. OJO lo que arroja esto son valores y NO una lista.

```
#Generadores
```

```
#Ejemplo 1: Crear los cuadrados de los
```

```
#numeros del 0 al 3
```

```
G1= {x**2 for x in range(4)}
```

GENERADORES

Efectivamente si verificamos en la consola o “shell”:

```
>>> G1
<generator object <genexpr> at 0x00F5AC10>
>>> type(G1)
<class 'generator'>
```

GENERADORES

```
....  
>>> next(G1)  
0  
>>> next(G1)  
1  
>>> next(G1)  
4  
>>> next(G1)  
9  
>>> next(G1)  
Traceback (most recent call last):  
  File "<pyshell#10>", line 1, in <module>  
    next(G1)  
StopIteration
```

GENERADORES

La segunda opción es emplear una función con la palabra reservada “yield”:

```
#Ejemplo 2: Generados mediante función
#que hace lo mismo que el ejemplo 1
def f(n):
    c=0
    while c<n:
        yield c**2
        c+=1

#Llamamos la función y almacenamos
#el generador en una variable
a=f(4)
```

GENERADORES

Y verificamos en la consola o “shell”:

```
>>> a
<generator object f at 0x00F68C10>
>>> type(a)
<class 'generator'>
```

GENERADORES

```
>>> next(a)
```

```
0
```

```
>>> next(a)
```

```
1
```

```
>>> next(a)
```

```
4
```

```
>>> next(a)
```

```
9
```

```
>>> next(a)
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#8>", line 1, in <module>
```

```
    next(a)
```

```
StopIteration
```


DECORADORES

Los decoradores son funciones que reciben funciones como argumentos y devuelven funciones.

Debido a que debemos hacer una función que reciba funciones de cualquier tipo junto que argumentos de cualquier tipo debemos disponer de los siguientes parametros en el decorador:

- “*args”: Es un parámetro que recibe una cantidad “n” de atributos.
- “**kwargs”: Es un diccionario donde se recibirán los valores de los argumentos y sus respectivas llaves.

Ahora procederemos con el ejemplo si tenemos las siguientes funciones:

DECORADORES

```
#Funciones a decorar  
def suma(x, y) :  
    print (x+y)  
  
def resta(x, y) :  
    print (x-y)
```

DECORADORES

```
#Declaramos Funcion decoradora:
def decorador(funcion):
    #Declaramos una funcion decorada
    #dentro de la cual ira la funcion que se
    #va a decorar
    def funcionDecorada(*args,**kwargs):
        #Nota: __name__ se debe escribir con doble "_"
        #para que no salga error
        print("Fue decorada la funcion: ",funcion.__name__)
        funcion(*args, **kwargs)

    #Devolvemos la funcion decorada
    #SIN ARGUMENTOS
    return funcionDecorada
```

DECORADORES

```
#Finalmente llamamos a las funciones
```

```
#Decoramos las funciones
```

```
decorador(suma) (1,2)
```

```
decorador(resta) (2,1)
```

DECORADORES

```
#Tambien podemos decorar las  
#funciones permanentemente  
S=decorador (suma)  
S (1, 2)  
  
R=decorador (resta)  
R (2, 1)
```

DECORADORES

```
#Funciones a decorar
#Otra opcion es declarando las funciones a
#decorar de la siguiente manera
@decorador
def suma(x,y):
    print(x+y)

@decorador
def resta(x,y):
    print(x-y)
```

```
#Y llamandolas SOLAMENTE asi
suma(1,2)
resta(2,1)
```

DECORADORES

Ahora probemos un ejemplo complejo con varios decoradores, primero tenemos uno llamado “administrador que se encargará de verificar si la clave que introduzca el usuario es correcta ANTES de ejecutar la “función a decorar”:

```
#Decorador llamado 'administrador'
def administrador(funcion):
    def verificacion(*args, **kwargs):
        clave=input('Introduzca la clave de acceso: ')
        if clave=='12345':
            funcion(*args, **kwargs)
        else:
            print('Permiso denegado!!!!!!')
    return verificacion
```

DECORADORES

```
#Declaramos Funcion decoradora:
def decorador(funcion):
    #Declaramos una funcion decorada
    #dentro de la cual ira la funcion que se
    #va a decorar
    def funcionDecorada(*args,**kwargs):

        #Nota: __name__ se debe escribir con doble "_"
        #para que no salga error
        print("Fue decorada la funcion: ",funcion.__name__)
        funcion(*args, **kwargs)

    #Devolvemos la funcion decorada
    #SIN ARGUMENTOS
    return funcionDecorada
```

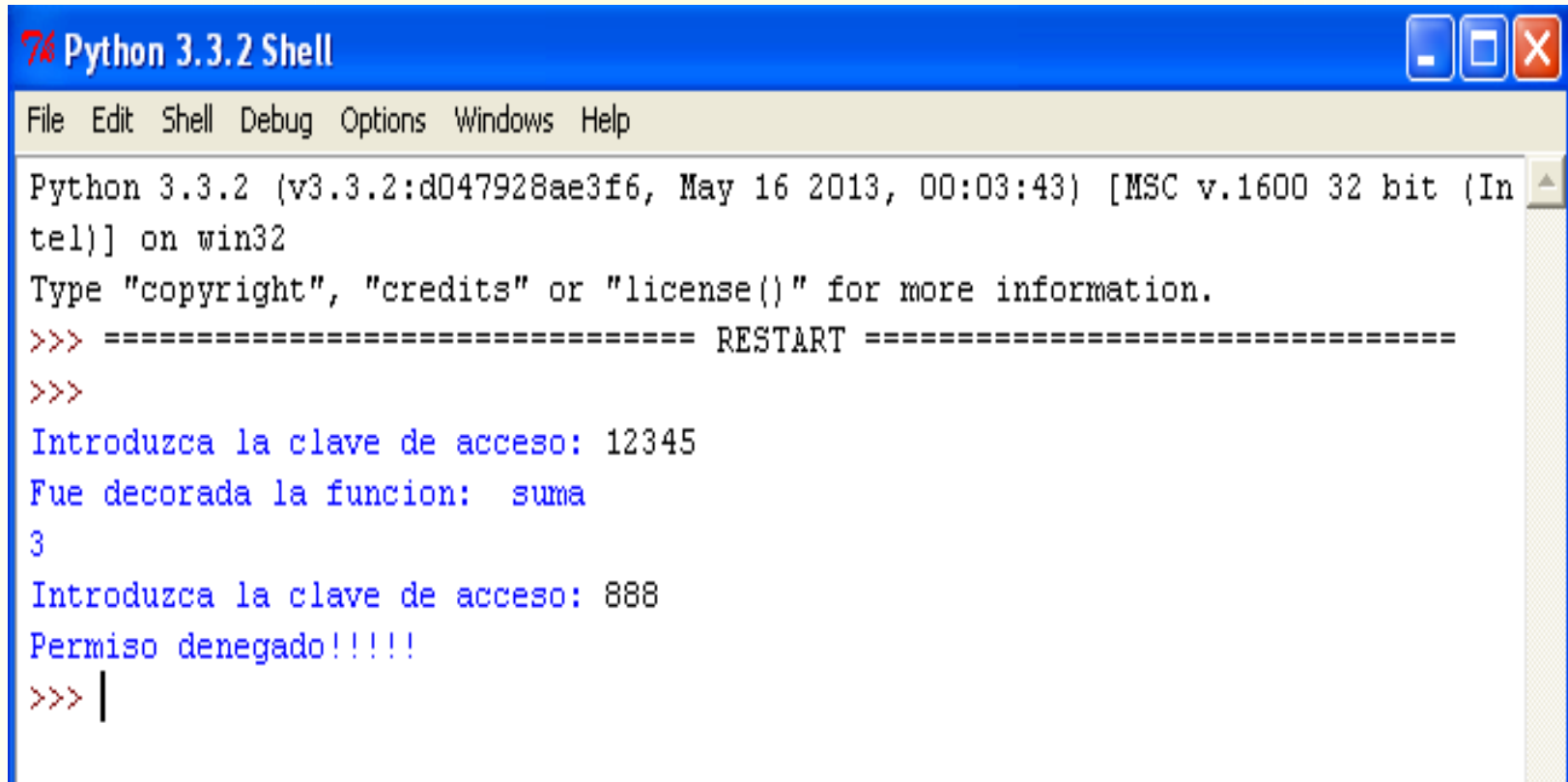

DECORADORES

```
#Funciones a decorar: Primero se ejecutara  
#"@decorador" y luego "@administrador"  
#La ejecucion es de abajo hasta arriba  
@administrador  
@decorador  
def suma(x, y):  
    print(x+y)  
  
@administrador  
@decorador  
def resta(x, y):  
    print(x-y)
```

DECORADORES

```
#Llamamos las funciones que  
#se decoraran automaticamente  
suma(1,2)  
resta(2,1)
```

DECORADORES



The screenshot shows a Python 3.3.2 Shell window with a blue title bar and standard Windows window controls. The menu bar includes File, Edit, Shell, Debug, Options, Windows, and Help. The main text area displays the following content:

```
Python 3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:03:43) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> ===== RESTART =====
>>>
Introduzca la clave de acceso: 12345
Fue decorada la funcion: suma
3
Introduzca la clave de acceso: 888
Permiso denegado!!!!
>>> |
```

MANEJO DE ARCHIVOS

```
#Se crea un objeto archivo con la funcion 'open'  
#la funcion open tiene como parametros en este caso  
#el nombre del archivo que se desea abrir  
#y el modo: 'wt' para modo escritura en este caso
```

```
#Si el archivo no existe lo crea  
#Si ya existe se sobre escribe el archivo  
ArchivoSalida=open('Salida.txt','wt')  
#Indica el tipo de dato  
ArchivoSalida.write(str(type(ArchivoSalida)))  
  
ArchivoSalida.write("\nHola Archivo")  
ArchivoSalida.write("\nAlgo de Texto")  
Texto="\nAun mas texto"  
ArchivoSalida.write(Texto)  
ArchivoSalida.close()
```

```
#Ahora se va a abrir el archivo para leerlo
```

```
ArchivoEntrada=open('Salida.txt','rt')  
print(ArchivoEntrada.read())  
ArchivoEntrada.close()
```

MANEJO DE ARCHIVOS

```
#Metodos
```

```
Archivo=open('Salida.txt','rt')
```

```
#Salto por caracteres
```

```
Archivo.seek(20)
```

```
print(Archivo.read())
```

```
Archivo.seek(0)
```

```
#Leer un numero limitado de caracteres
```

```
print(Archivo.read(40))
```

```
print(Archivo.read())
```

```
#Devuelve el offset o la posicion del ultimo caracter
```

```
print(Archivo.tell())
```

```
Archivo.close()
```

MANEJO DE ARCHIVOS

```
Texto="\nHola Archivo"
Archivo=open("Archivo.txt", "wt")
Archivo.write(Texto)
Texto="\nPrueba"
Archivo.write(Texto)
Texto="\nLinea 1"
Archivo.write(Texto)
Texto="\nLinea 2"
Archivo.write(Texto)
Archivo.close()
```

#Otra forma de abrir el archivo

```
with open("Archivo.txt", "rt") as Archivo:
    print(Archivo.read())

Archivo.close()
```

MANEJO DE ARCHIVOS

```
import pickle

#El modulo pickle sirve para crear
#representaciones serializadas portables de
#los objetos

Lista=['Casa','Carro','Manzana',2,2+3j]
#Cambiar el 'wt' por 'wb'
Archivo=open('ArchivoPickle.txt','wb')

#Escribir en el archivo
pickle.dump(Lista,Archivo)
Archivo.close()

#Leer Archivo y cargar su contenido

Archivo=open('ArchivoPickle.txt','rb')
CargarLista=pickle.load(Archivo)
print(CargarLista)
Archivo.close()
```

ESCRIBIR TABLAS EN PYTHON

```
#Tablas en python 3.3
```

```
#Llenamos la siguiente tabla con un for
```

```
#Imprimimos el cuadrado y el cubo de x .format(x,x*x,x*x*x)
```

```
#Notese que no llegaremos al limite superior
```

```
for x in range(1,15):
```

```
    #Formato: Lo que hacemos es crear la tabla donde: cada elemento
```

```
    #entre {} es una columna la cual se le asigna en orden a cada
```

```
    #argumento dentro de la funcion .format()
```

```
    #{0:2d} nos garantiza que la tabla se vea bien dejando espacios
```

```
    #para que se vea bien la tabla
```

```
    print ('{0:2d} {1:3d} {2:4d}'.format(x,x*x,x*x*x))
```


EXCEPCIONES Y ERRORES

La gestión de errores es algo muy importante en proyectos de alta complejidad, para garantizar el correcto funcionamiento del programa evitando, en varias ocasiones puede que Python por si mismo no pueda ayudarnos a identificar el error apropiadamente por lo cual debemos servirnos de las excepciones para poder detectar que partes de nuestro código podría fallar.

EXCEPCIONES Y ERRORES

Python emplea palabras reservadas para la gestión de excepciones como:

- “try”
- “except”
- “continue”
- “break”
- “TypeError”
- “NameError”
- “finally”
- “raise”

EXCEPCIONES Y ERRORES

Podemos dispararlas manualmente bien sea de esta forma:

```
#Excepciones manuales

#Vamos a verificar la division por cero

a=int(input(' Introduzca un Divisor: '))
b=int(input(' Introduzca un Dividendo: '))

#Verificamos que la division no sea entre cero
if b==0:
    #Disparamos el error manualmente
    raise NameError('Division por cero NO valida')
else:
    #Si no hay error hacemos la operacion manualmente
    print("Division valida")
    print(a/b)
```

EXCEPCIONES Y ERRORES

O de esta otra forma:

```
#Excepciones manuales
```

```
#Vamos a verificar la division por cero
```

```
a=int(input('Introduzca un Divisor: '))
```

```
b=int(input('Introduzca un Dividendo: '))
```

```
#Verificamos que la division no sea entre cero
```

```
if b==0:
```

```
    #Disparamos el error predeterminado por Python manualmente
```

```
    raise ZeroDivisionError
```

```
else:
```

```
    #Si no hay error hacemos la operacion manualmente
```

```
    print("Division valida")
```

```
    print(a/b)
```

EXCEPCIONES Y ERRORES

Otra opción puede ser:

```
#Excepciones: Uso 'continue' , 'break', 'input'

#Declarar unos valores
total=0
contador=0

#Este while es un bucle infinito
while True:

    variable=input('Introduzca un numero: ')
    #Si se a introducido un valor....
    if variable:
```

EXCEPCIÓNES Y ERRORES

```
#En el bloque 'try', ponemos el codigo
#que es suceprible de error
try:
    #...el entero de 'variable' se almacenara en 'numero'
    numero=int(variable)
    print('Contador: ',contador,'\nTotal: ', total)
```

EXCEPCIONES Y ERRORES

```
#Almacenara un dato erroneo como 'error'
#el bloque 'except' se ejecuta si hay un error
except ValueError as error:
    #En caso de que el usuario introduzca un dato
    #NO valido imprimira el dato incorrecto como 'error'
    print(error)
    print('\nEl dato no es valido!!!!!!!!!!!!')
    #Al decir 'continue' REGRESARA al 'while'
    #es opcional
    continue
```

EXCEPCIÓNES Y ERRORES

```
#El bloque 'finally' se ejecuta halla o no, error
finally:
    print('Se continua con la ejecucion de todas formas')

total +=numero
contador +=1

else:
    #Si no se introduce ningun valor se sale del bucle
    break
```


PROGRAMACIÓN ORIENTADA A OBJETOS

¿QUE ES UN OBJETO?:

Son entidades(cosas, personas, animales, etc...) a las cuales se le atribuyen las siguientes propiedades:

- Estado/Atributo: Son datos o valores que describen el objeto (Longitud, color, material...)
- Comportamiento: Más conocidos como métodos, es lo que podemos hacer con el objeto como por ejemplo abrir una caja, encender un televisor, etc...
- Identidad: Es lo que diferencia a un objeto de OTROS objetos de su misma **CLASE**

PROGRAMACIÓN ORIENTADA A OBJETOS

¿COMO DEFINIMOS UN OBJETO Y QUE ES ENTONCES LA PROGRAMACIÓN ORIENTADA A OBJETOS?:

En la mayoría de lenguajes de programación como Python los objetos los declararemos por medio de las **CLASES** las cuales agruparan las propiedades anteriormente enunciadas (Estado, comportamiento e identidad).

De esta forma definiremos la programación orientada a objetos puede definirse como: el desarrollo de aplicaciones informáticas a partir de estos objetos y sus interacciones. [5]

PROGRAMACIÓN ORIENTADA A OBJETOS

CONCEPTOS FUNDAMENTALES [6]:

- Clase: Contiene los atributos y métodos de todos los objetos, la instanciación es el procedimiento mediante el cual creamos un objeto en particular de una determinada clase.
- Herencia: Propiedad mediante la cual se transfieren o comparten propiedades de objetos de una clase (padre) a otra (hija) .
- Objeto: Instancia de una clase.
- Metodo: Es un comportamiento asociado a un objeto el cual se ejecutara al recibir un mensaje.

PROGRAMACIÓN ORIENTADA A OBJETOS

PROPIEDADES FUNDAMENTALES [6]:

- Abstracción: Es el aislamiento de una entidad de tal forma que no nos preocupemos en el 'como lo hace', algo así como encerrarlo en una caja negra.
- Encapsulamiento: Es el ocultamiento de los datos de tal forma que estos solo puedan ser accedidos por los métodos del objeto.

PROGRAMACIÓN ORIENTADA A OBJETOS

PROPIEDADES FUNDAMENTALES [6]:

- Modularidad: Permite dividir la aplicación en partes más pequeñas las cuales pueden funcionar de forma independiente.
- Principio de ocultación: NO confundir con abstracción o encapsulamiento, consiste en aislar al objeto del exterior y hacen accesible los atributos solo por medio de una determinada interfaz.

PROGRAMACIÓN ORIENTADA A OBJETOS

PROPIEDADES FUNDAMENTALES [6]:

- Polimorfismo: Es la capacidad de usar objetos de diferente clase por medio de una misma interfaz por ejemplo obtener el perímetro de un triángulo y/o un rectángulo. (ver ejemplo aquí:
[http://es.wikipedia.org/wiki/Polimorfismo_\(programaci%C3%B3n_orientada_a_objetos\)](http://es.wikipedia.org/wiki/Polimorfismo_(programaci%C3%B3n_orientada_a_objetos))
))
- Herencia: Propiedad mediante la cual se transfieren o comparten propiedades de objetos de una clase (padre) a otra (hija) .

PROGRAMACIÓN ORIENTADA A OBJETOS

PROPIEDADES FUNDAMENTALES [6]:

- Recolección de basura: es la técnica por la cual el entorno de objetos se encarga de destruir automáticamente, y por tanto desvincular la memoria asociada, los objetos que hayan quedado sin ninguna referencia a ellos

PROGRAMACIÓN ORIENTADA A OBJETOS

Antes de Realizar el próximo ejercicio cree dos archivos uno que se llame “Classes.py” y otro que se llame “main.py”, que estén en LA MISMA CARPETA.

PROGRAMACIÓN ORIENTADA A OBJETOS

En “Classes.py” escriba el siguiente código y guarde el archivo:

```
#Declaracion de la clase
```

```
#self: permite acceder a las propiedades y métodos de un objeto concreto  
#contiene una referencia al objeto que recibe  
#la señal (el objeto cuyo método es llamado),  
#por lo que podemos usar esa referencia para acceder  
#al espacio de nombres del objeto,  
#es decir, a sus atributos individuales.  
#_init_: Inicializa el objeto
```

```
class Clase:
```

```
    def __init__(self, numero, palabra):  
        self.numero = numero  
        self.palabra=palabra  
        self.a = numero**2  
        self.b = numero**3  
        self.c = 999
```

```
    def imprimir(self):  
        print("Hola yo soy {0}".format(self.palabra))
```

```
    def dime_datos(self):  
        print("Con x=%i obtenemos: a=%i, b=%i, c=%i" % (self.numero, self.a, self.b, self.c))
```

PROGRAMACIÓN ORIENTADA A OBJETOS

En “main.py” escriba el siguiente código y guarde el archivo, luego ejecute el programa (F5):

```
#Llamara al archivo que continene la clase de  
#preferencia debe estar en la misma carpeta
```

```
from Classes import Clase
```

```
#Probando la clase
```

```
a=Clase(8, "Camilo")
```

```
a.imprimir()
```

```
a.dime_datos()
```

PROGRAMACIÓN ORIENTADA A OBJETOS (HERENCIA)

```
#La clase B hereda los atributos y los metodos de la Clase A

class ClaseA:
    def __init__(self, x):
        self.a = x
        self.b = 2 * x
    def muestra(self):
        print ("a=%i, b=%i" % (self.a, self.b))

class ClaseB(ClaseA):
    def __init__(self, x, y):
        ClaseA.__init__(self, x)
        self.c = 3 * (x + y) + self.b
    def muestra(self):
        print ("a=%i, b=%i, c=%i" % (self.a, self.b, self.c))

#a.__dict__ muestra los atributos de la clase

print ("Objeto A")
a = ClaseA(2)
print (a.__dict__)
a.muestra()

print ("ObjetoB")
b = ClaseB(2, 3)
print (b.__dict__)
b.muestra()
```

PROGRAMACIÓN ORIENTADA A OBJETOS (CLASES DECORADORAS)

Al igual que las funciones decoradoras, podemos también crear “clases” decoradoras, por ejemplo para las funciones siguientes del ejemplo de decoradores que vimos anteriormente:

```
#Funciones a decorar
@decorador
def suma(x, y) :
    print (x+y)

@decorador
def resta(x, y) :
```

PROGRAMACIÓN ORIENTADA A OBJETOS (CLASES DECORADORAS)

Cambiamos la función decoradora por una clase decoradora:

```
#Declaramos la funcion decoradora
class decorador(object):
    #Funcion inicializadora del objeto
    def __init__(self,function):
        self.function=function
    #Este metodo hara de funcion decorada de salida
    #Y sera llamado cada vez que se ejecute la funcion
    def __call__(self,*args,**kwargs):
        print("Esta funcion fue decorada: ",self.function.__name__)
        self.function(*args,**kwargs)
```

PROGRAMACIÓN ORIENTADA A OBJETOS (CLASES DECORADORAS)

Finalmente llamamos las funciones ejecutadas:

```
#Llamamos las funciones que  
#se decoraran automaticamente  
suma(1,2)  
resta(2,1)
```

BIBLIOGRAFIA

[1] <http://es.wikipedia.org/wiki/Python>

[2]

<http://reflexionesdesdebaetulo.files.wordpress.com/2010/07/spanish-inquisition.jpg>

[3] <http://www.python.org/>

[4] <http://docs.python.org/3/tutorial/>

[5] <http://www.desarrolloweb.com/articulos/499.php>

[6]

http://es.wikipedia.org/wiki/Programaci%C3%B3n_orientada_a_objetos

SOBRE EL AUTOR Y EL CONTENIDO

A menos que se informe de otra manera esta obra está bajo una licencia de Creative Commons Reconocimiento-NoComercial-CompartirIgual 2.5 Colombia.



Diego Camilo Peña Ramírez
Bogotá, Colombia
Agosto de 2013

Twitter: @nervencid