Python

(Por: Antonio Montaño Ramírez
amramirez@sadiel.es)

Índice:

Título: Página:

Python:

-*Python* está considerado como un lenguaje de muy alto nivel, es por ello que se facilita la lectura de su código.

Este lenguaje de programación está pensado para simplificar el trabajo del programador, con sentencias más cortas y menos líneas de código que otros lenguajes de programación.

Un ejemplo de ello es la posibilidad de utilizar una misma variable, en el mismo código, como dos tipos distintos de datos, sin haberlo declarado y sin tener en consideración el tipo de datos con el que fue tratado anteriormente. Esto le da al programador ligereza al leer el código sin tener que preocuparse del tipo de datos con el que tratamos.

Python es un lenguaje de script (por lo que no es compilado), lo que lo hace muy potente en el sentido de la sencillez y la capacidad de escribir programas cortos y efectivos, pero pone en contra que es mucho más lento que cualquier programa en código binario.

-El prompt será el encargado de decir donde empieza una línea de código, y lo denotaremos de aquí en adelante como: ">>>"

A continuación mostraremos las líneas de código básicas con las que trabajaremos en Python.

Código básico:

Comentarios:

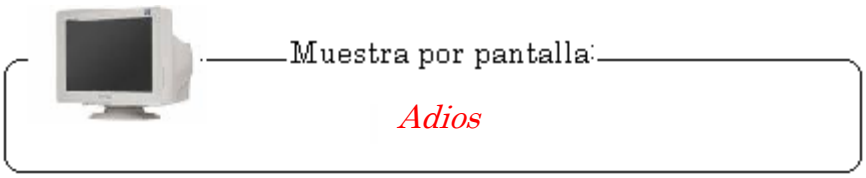
Los comentarios sirven para mostrar un texto en un trozo de código, y que esta sea desechada por Python, y de este modo poder poner , por ejemplo, como notas de lo que realiza ese código.

Tenemos 2 formas distintas de comentar código:

a) # Con este símbolo comentaremos todo lo que se encuentre a la derecha de él, haciéndolo invisible a la ejecución del programa

Ejemplo:

```
>>>a='Hola'
>>>#print a
>>>print 'Adios'
```



Puesto que la sentencia a no la ejecutara Python ya que está comentada

b) “””<línea 1>
...
<línea n>“””

De este modo comentaremos más de una línea de texto, o sentencias del programa

```
>>>a='Hola'
>>>""""  print a
>>>print 'Adios'  """"
```

Esto no mostrará nada por pantalla, ya que las dos últimas líneas están comentadas.

print:

Utilizaremos la palabra reservada “print” para decirle a Python que muestre por pantalla el texto que pongamos a continuación.
Realmente lo que hace print es llamar a un módulo del espacio de nombre sys: sys.stdout.write(‘cadena’), el cual le pasa por parámetro de entrada lo que pongas a continuación de print.

Equivalentes	

import sys #(“Ya lo veremos más #adelante”)
sys.stdout.write(‘Hola’)
	print 'Prueba'

-Mostrar por pantalla:
Sintaxis:

print Mensaje_a_mostrar

Ejemplo:

```
>>> print 'Hola mundo desde Python'
```

raw_input:

-Tomar datos desde el teclado:

Tomar datos desde el teclado es muy parecido a la filosofía usada para el print, ya que en este caso haremos uso del módulo sys.stdin.read().

Sintaxis:

variableDeSalida = raw_input(Mensaje por pantalla)

Ejemplo:

```
>>>sueldo = raw_input('Introduce el sueldo')
```

Python orientado a objetos:

La orientación a objetos representa una gran ventaja respecto al método tradicional. En este sentido la orientación a objetos puede considerarse como una manera de ampliar las posibilidades de los tipos de datos estructuras(llamados struct en el caso de C).

Ahora podemos crear unas clases con todas las funciones y atributos que queramos, y crear objetos de esas clases.

Por ejemplo, podemos crear la clase persona con un atributo edad, dni, nombre... y algunas funciones como pueden ser: dime_La_Edad dime_El_Dni..., y crear varios objetos de personas.

La idea de esto es crear varios ficheros (Clases) y desde el programa principal crear varios objetos para trabajar con ellos. Ahora cualquier tipo de datos será un objeto, y se tratará como tal.

Crear clases:

Sintaxis:

class nombreClase(parámetroDeEntrada1,..., parámetroDeEntradaN):

Ejemplo:

```
>>> class Persona():
```

-Importar:

a) Importar fichero:

Sintaxis:

import nombreClase

Ejemplo:

```
>>>import trabajo
```

Modo de acceder a funciones de ese fichero:
Sintaxis:

nombreFichero.nombreMétodo

Ejemplo:

```
>>>trabajo.sueldo
```

b) Importar funciones de un fichero:

Sintaxis:

From nombreFichero import nombreMétodo1,...,NombreMétodoN

Ejemplo:

```
>>>from persona import MuestraNombre
```

Funciones:

La diferencia que tiene Python con otros lenguajes de programación respecto las funciones es que solo tienen dos tipos de acceso, públicos (se puede ver desde cualquier clase que esté en el mismo proyecto), y privado (que solo se puede acceder desde la misma clase). Para el segundo caso tenemos que usar el siguiente formato:

Sintaxis:

def __nombreMétodo__(parámetroDeEntrada1,..., parámetroDeEntradaN):

Ejemplo:

```
>>>def __init__(self):
```

-Palabra reservada self:

Python utiliza la palabra reservada self para referirse al objeto que estemos tratando, en otros lenguajes se utiliza la palabra reservada this, como es el caso en java.

-Constructor:

Es un tipo especial de método, que se encarga de crear un objeto ó instancia de un tipo en concreto de clase.
En Python , el método constructor tiene el siguiente formato dentro de la clase:

Sintaxis:

def __init__(parámetroDeEntrada1,..., parámetroDeEntradaN):

Ejemplo:

```
>>>def __init__(self):
```

Para llamar a este constructor desde fuera de la clase se utiliza el nombre de la clase que quieres sacar un objeto:

Sintaxis:

nuevoObjeto = nombreClase(parámetroDeEntrada1 ,..., parámetroDeEntradaN):

Ejemplo:

```
>>>persona1=Persona('Antonio','Montaño Ramírez',Programador)
```

Programa ejemplo que muestra todo lo explicado hasta ahora:

Nombre fichero: "persona.py"

```
import trabajo
class Persona():
    def __init__(self, Nombre, Apellidos, Trabajo):
        self.Nombre = Nombre
        self.Apellidos = Apellidos
        self.Trabajo = Trabajo
    def MuestraNombre(self):
        print self.Nombre + ' ' + self.Apellidos
    def MuestraEmpleo(self):
        print 'Titulo: ' + str(self.Trabajo.Titulo)
        print 'Sueldo: ' + str(self.Trabajo.Sueldo)
```

Nombre fichero: "trabajo.py"

```
class Trabajo():
def __init__(self, Titulo, Sueldo):
    self.Titulo = Titulo
    self.Sueldo = Sueldo
```

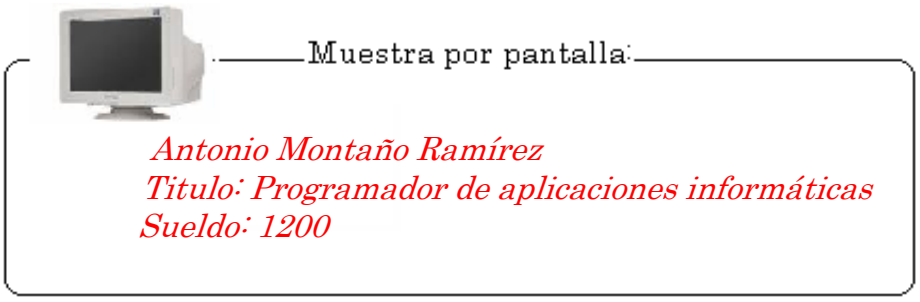
Nombre fichero: "main.py"

```
from trabajo import Trabajo
from persona import Persona
class Main():
    programador=Trabajo('Programador de aplicaciones informáticas',1200)
    persona1=Persona('Antonio','Montaño Ramírez',programador)
    persona1.MuestraNombre()
    persona1.MuestraEmpleo()
```

Al ejecutar el fichero main.py con Python, se crea un objeto de tipo trabajo, y otro de tipo persona, con el objeto trabajo que acabamos de crear como uno de sus parámetros.

Para este caso hemos usado un método llamado str en el cual nos retorna (sí puede) el parámetro que hemos introducido como una cadena de caracteres.

También hemos usado el elemento + para concatenar cadenas de texto.
El resultado final muestra por pantalla:



Podemos ver en el ejemplo anterior que las sentencias de las funciones tienen una mayor indexación o sangrado que su cabecera.

Veamos un ejemplo de la clase persona:

```
1) >>>    def __init__(self, Nombre, Apellidos, Trabajo):
2) >>>        self.Nombre = Nombre
```

Vemos que la segunda línea se encuentra más a la derecha que la primera, eso significa que es parte del método __init__ .

También vemos que ocurre con las clases:

```
1) >>>class Persona():
2) >>>    def __init__(self, Nombre, Apellidos, Trabajo):
```

Así vemos que la segunda línea se encuentra dentro de la clase Persona

Listas:

Las listas en Python son un tipo de datos para almacenar una colección de objetos. Estos objetos no tienen por qué ser del mismo tipo de datos, de este modo puedo tener en una lista con un entero, dos objetos de tipo persona, etc. Esta lista se puede acceder de manera directa, es decir, puedes acceder a un objeto dentro de una lista por su índice, ó lo que es lo mismo, por su posición dentro de la lista.

En el caso particular de java, las listas son unos tipos de datos bastantes más complejos y difíciles de manejar. Un ejemplo de las diferencias es que java solo acepta objetos de tipo Object en sus listas, con lo que cada vez que lo usemos debemos hacer una conversión de tipo, y en Python es completamente invisible este paso.

Sintaxis:

nuevoObjeto = [nombreaVariable1,...,nombreaVariableN]

Ejemplo:

```
>>> listaDeCompra = ["Manzanas","Peras","Leche"]
```

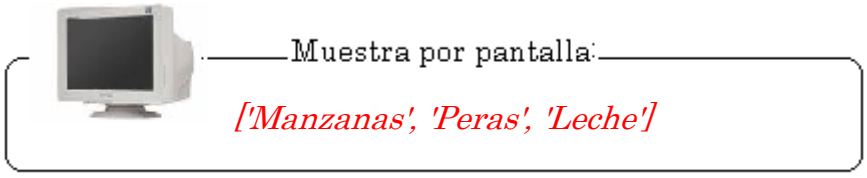
Una de las ventajas que tienen estas listas, es la facilidad de su manejo, como ejemplo sirva la manera de imprimir por pantalla, y las opciones que nos posibilita:

Escribir por pantalla:

Si lo que queremos es mostrarla tal cual, nos sirve poner la sentencia print.

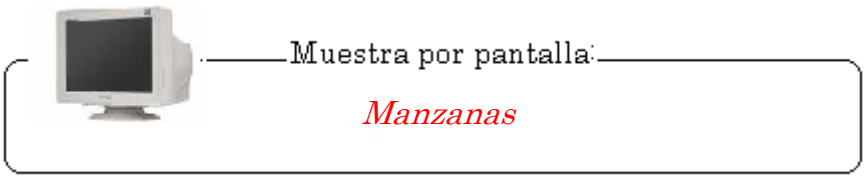
Ejemplo:

```
>>>print listaDeCompra
```



Pero Python nos ofrece más posibilidades para el manejo de listas, como es la anteriormente mostrada de acceso directo:

```
>>>print listaDeCompra[0]
```

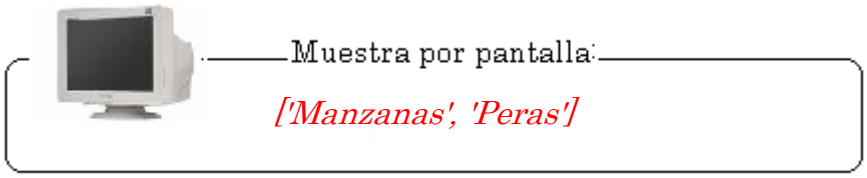


También nos da la posibilidad de acceder a varios elementos a la vez, y lo haremos escribiendo el intervalo que deseamos. Para ello utilizaremos los dos puntos.

Ejemplo:

```
>>>print listaDeCompra[0:2]
```

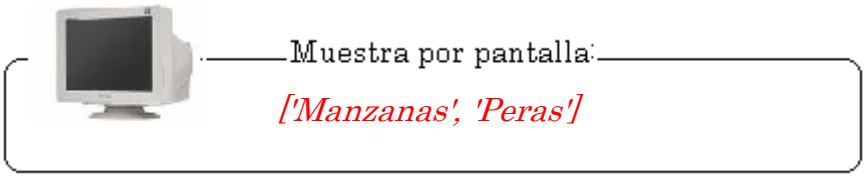
En este caso nos mostrará un intervalo:



Otra de las características especiales que tienen estas listas es poder acceder a números negativos, eso se explica por que Python entiende como -1 al último elemento de la tabla, -2 el penúltimo, y así sucesivamente.

Ejemplo:

```
>>>print listaDeCompra[0:-1]
```



Insertar:

Tenemos tres opciones a la hora de añadir nuevos elementos a la lista:

1) Insert: Añade en la posición ***x*** el parámetro que queramos:

   Ejemplo:

```
>>>listaDeCompra.insert(1, 'Huevos')
```

2) Append: Añade el parámetro de entrada, al final de la lista:

   Ejemplo:

```
>>> listaDeCompra.append('Huevos')
```

3) Extend: Añade todos los objetos de otra lista.

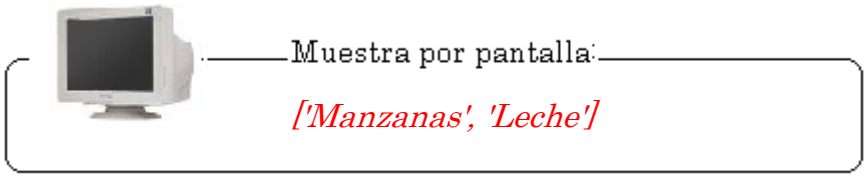
Ejemplo:

```
>>> listaDeCompra.extend(['Galletas','Salchichas'])
```

Borrar:

La sentencia para borrar en una lista se llama remove, que simplemente le entra como parámetro de entrada en elemento que queremos borrar:

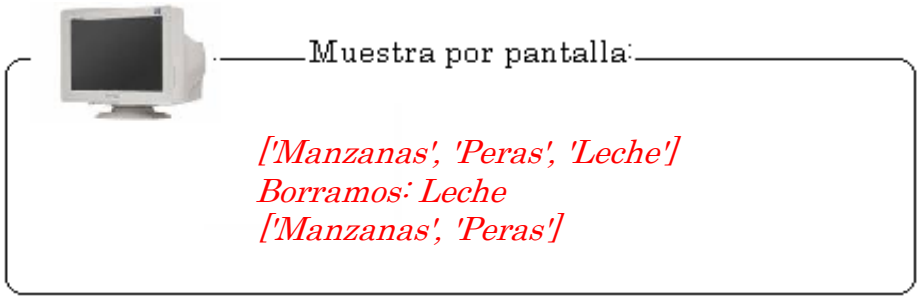
```
>>>listaDeCompra = ["Manzanas","Peras","Leche"]
>>>listaDeCompra.remove("Peras")
>>>print listaDeCompra
```



La otra manera de borrar un objeto de la lista es con la sentencia pop, en la que difiere con la anterior en que esta borra el último objeto de la lista y lo devuelve.

Ejemplo:

```
>>>listaDeCompra = ["Manzanas","Peras","Leche"]
>>>print listaDeCompra
>>>print 'Borramos: ' + str(listaDeCompra.pop())
>>>print listaDeCompra
```

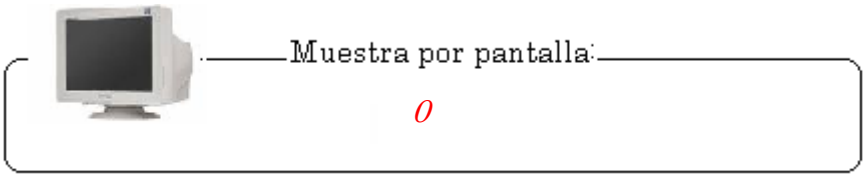


Búsqueda:

La busqueda la realizaremos con el método index, que le entra como parámetro el objeto a buscar y devuelve la posición en la que se encuentra ó lanza una excepción ValueError si no está.

Ejemplo:

```
>>>listaDeCompra = ["Manzanas","Peras","Leche"]
>>>print listaDeCompra.index("Manzanas")
```



Sentencia if:

La sentencia condicional if es la encargada de decidir si una condición se cumple o no, y lo que debe hacer en cada uno de los casos en los que nos encontremos.
Esta sentencia, en Python, se utiliza con el mismo formato de sangría que las funciones, en caso de que la sentencia a tratar tenga más sangrado que la sentencia if, significa que es parte de ella.

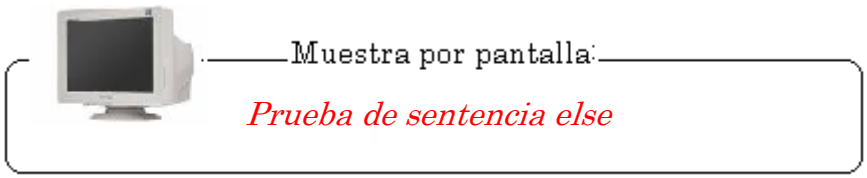
Ejemplo:

```
>>>if 'a' =='a':
>>>    print 'Prueba de sentencia if'
```

En este caso vemos que la sentencia siempre será correcta, con lo que pasa a la línea inferior, que está dentro del if, puesto que así lo demuestra el espacio antes de print. Para el caso que queramos controlar cuando se evalúa a falso usaremos la sentencia condicional else.

Ejemplo:

```
>>> if 'a' =='b':
>>>    print 'Prueba de sentencia if'
>>>else:
>>>    print 'Prueba de sentencia else'
```



Python también nos da la posibilidad de hacer una sentencia condicional múltiple, eso significa que en el caso de no evaluar la condición a cierto, podemos volver a realizar una sentencia if.
Ejemplo:

```
>>>if 'a' =='b':
>>>    print 'Prueba de sentencia if'
>>>else:
>>>    if 'b'=='b':
>>>        print 'Prueba de sentencia else'
```

Vemos que esta es una manera muy engorrosa de hacerlo, así que Python nos ofrece la posibilidad de hacer una sentencias más elegante, llamada elif.

Ejemplo:

```
>>>if 'a' =='b':
>>>    print 'Prueba de sentencia if'
>>>elif 'b'=='b':
>>>        print 'Prueba de sentencia else'
```

Sentencias iterativas:

Las sentencias iterativas nos haran recorrer un número de veces el mismo trozo de código. La cantidad de veces que las recorramos dependerá de la condición que le pongamos a la iteración, y el modelo de setencia iterativa que pongamos. En este caso veremos 2.

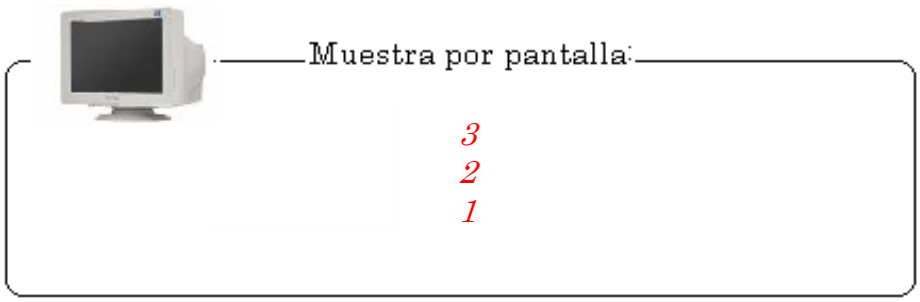
while:

-La sentencia while se utiliza para iterar sobre un trozo de código mientras sea cierta la condición impuesta.

Ejemplo:

```
>>>cont=3
>>>while(cont>0):
>>>    print cont
>>>    cont=cont-1
```

Que dará como resultado :



for-in:

-La sentencia for-in tiene la ventaja sobre while en el hecho de que no necesitamos un variable contador para llevar las cuentas de las veces que iteramos.

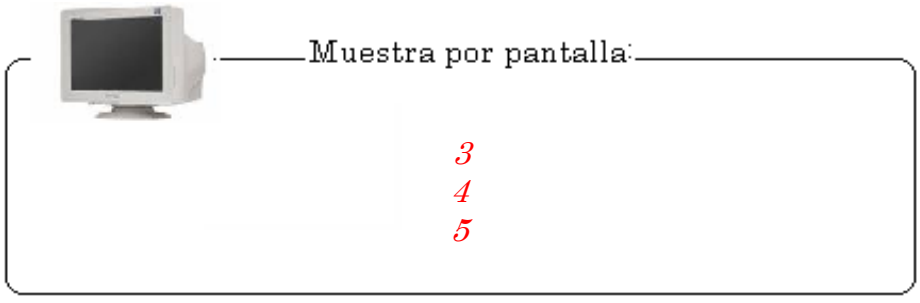
Esta sentencia suele usarse conjuntamente con un módulo llamado range(int,int).

Ejemplo:

```
>>>for i in range(3,6)
>>>    print i
```

Hagamos una traza del código:

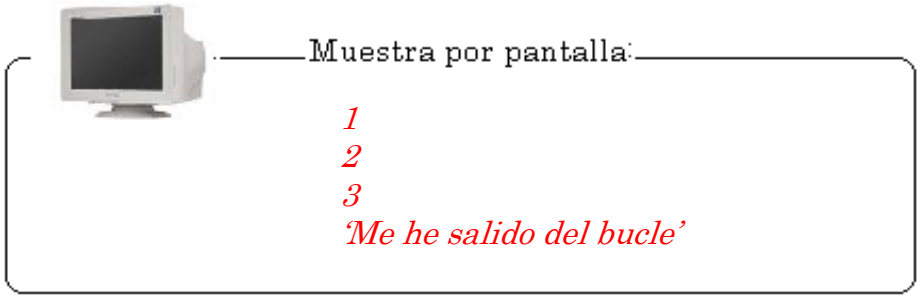
Empezamos en el for, y usamos como valor para i el valor 3, que es el principio del rango, y muestra por pantalla 3, en la segunda iteración usamos el segundo valor del rango y lo insertamos en i, el 4, así seguiremos desde el primero hasta 1 menos que el último del rango, en este caso decimos que el último del rango es 6, así que cuando lleguemos a 5 saldremos del for.



Tanto para while, como para for-in, se puede utilizar la palabra reservada break para salir de la iteración de manera obligada:

Ejemplo:

```
>>>for i in range(1,6):
>>>     print i
>>>     if (i==3):
>>>          break
>>>print 'Me he salido del bucle'
```



Excepciones:

Las excepciones es el método usado por Python para controlar los errores de compilación de nuestros programas y tener control sobre ellos, para que de este modo podamos evitar que termine la ejecución de un programa de forma fortuita.

Para tener este control, Python nos proporciona 3 sentencias :

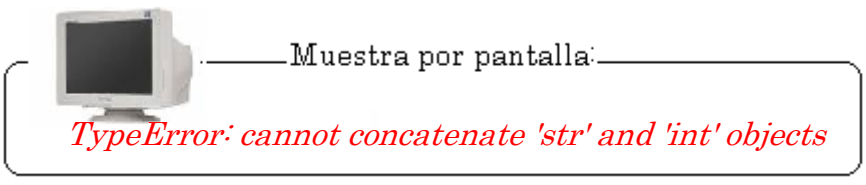
try:

-Esta palabra reservada indica al compilador que las siguientes sentencias pueden lanzar excepciones.

Ejemplo:

```
>>>try:
>>>     a = 'Hola'
>>>     b = 10
>>>     a = a + b
```

En este ejemplo, al ser a una cadena de caracteres y b un entero, el compilador dice que no se pueden concatenar, entonces nos muestra por pantalla un error:



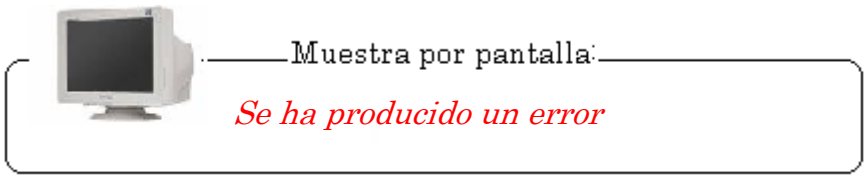
El cual controlaremos dentro de exception.

except:

-except es la palabra reservada que nos ofrece Python para saber que a partir de ahí, será el tratamiento de errores para las excepciones que encontremos dentro del try.

Ejemplo:

```
>>>try:
>>>    a = 'Hola'
>>>    b = 10
>>>    a = a + b
>>>except:
>>>    print 'Error'
```



La posibilidad de hacer un try-except dota a nuestros programas de mucha capacidad de maniobra, ya que si controlamos las excepciones pordemos hacer que el programa siga adelante.

La idea de esto es controlar más aún las excepciones, de modo que las controlemos de manera individual, para ello, Python nos crea un objeto que hereda de la clase exception y nos lo envía por parámetro a except, de este modo podemos saber cual fue la excepción exacta que acaba de lanzarse.

El modo más habitual de controlarlo es poniendo tantos except como excepciones queramos controlar.

Ejemplo:

```
>>>try:
>>>    a = 'Hola'
>>>    b = 10
>>>    a = a + b
>>>except TypeError:
>>>    print 'Error'
```

En este caso el resultado de nuestro programa no cambia con el anterior, puesto que antes controlamos todas las excepciones y ahora solo la que puede producirse.

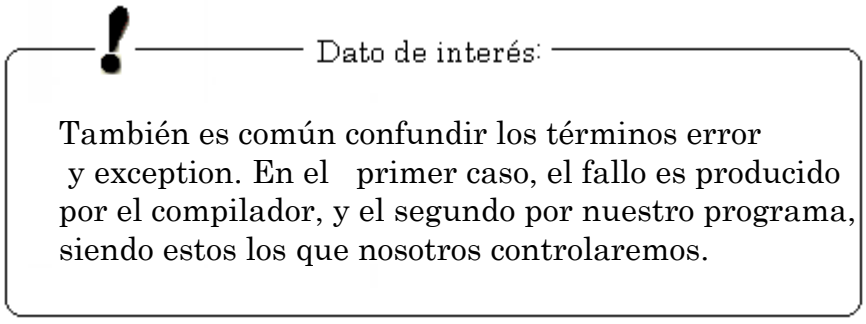
Para los casos en los que controlemos varias excepciones al mismo tiempo, puede que varias de ellas sean compatible, esto quiere decir que al mismo tiempo puede entrar en 2 except distinto, como es el caso de:

Ejemplo:

```
>>>try:
>>>    a = 'Hola'
>>>    b = 10
>>>    a = a + b
>>>except TypeError:
>>>    print 'Tipo erróneo de concatenación '
>>>except:
>>>    print 'Error estándar'
```

En este caso decimos que la primera excepción es más restrictiva que la segunda, con lo que debe ir primero, y no entrará en la excepción general. En caso de querer cambiar el orden, el compilador nos dirá que la sentencia except debe de ser la última de ella sin dejarnos ejecutar el programa.

Es un fallo muy común intentar controlar así todas las excepciones, el problema recae cuando tengamos un programa amplio con muchas posibilidades de lanzar excepciones, con lo que nos quedará un código casi ilegible a simple vista.



finally:

-Por último, veremos la sentencia finally, que es la encargada de seguir el programa después de controlar las excepciones.

Ejemplo:

```
>>>try:
>>>    a = 'Hola'
>>>    b = 10
>>>    a = a + b
>>>except TypeError:
>>>    print 'Error'
>>>finally:
 >>>    print 'Esto se mostrará para cualquier excepción'
```

La traza de este programa sería algo así:

Guardamos en ***a*** la cadena 'Hola', después guardamos en ***b*** el entero 10. Intentamos concatenarlas, pero el compilador nos lanza una excepción de tipo TypeError, con lo que vamos a buscar si la tenemos controlada.

En este caso podemos entrar en except TypeError: y mostrar por pantalla 'Error', para ahora seguir la ejecución del programa buscando si tiene o no la sentencia finally. En caso de que no tuviera acabamos el programa, en caso contrario seguimos la ejecución del programa por esta sentencia.

Como nuestro programa consta de un finally nuestro compilador nos lleva hasta allí y sigue la ejecución del programa con el *"print 'Esto se mostrará para cualquier excepción'"* dando así por terminado el porgrama.

Ficheros de texto:

Los ficheros serán el soporte permanente que usaremos para guardar información sobre nuestros programas, de este modo los programas dejarán de tener el uso de vida visto hasta ahora: se ejecuta el programa, se procesan datos introducidos por teclado y se muestran por pantalla.

Los ficheros nos dan soporte para poder guardar datos y luego procesarlos más adelante.

Varias formas de abrir ficheros:

Escritura: Se usa para añadir datos al fichero, borrando todo lo que tuviera anteriormente.

Lectura: Te da permiso para leer el fichero.

Añadir: Sirve para poder abrir un fichero e insertar texto detrás de este.

Cuando hemos abierto el fichero toca el momento de usarlo, y al finalizar debemos cerrar el fichero para que no tenga concurrencias y puedan ocurrir problemas.

Abrir fichero:

Sintaxis:

NombreFichero = open ('nombreFichero','modoDeApertura')

Ejemplo:

```
>>> fichero1 = open('Prueba1.txt','w')
```

Cierre de un fichero:

Sintaxis:

nombreFichero.close()

Ejemplo:

```
>>> fichero1.close()
```

Escritura en un fichero:

Escritura de un fichero:

Sintaxis:

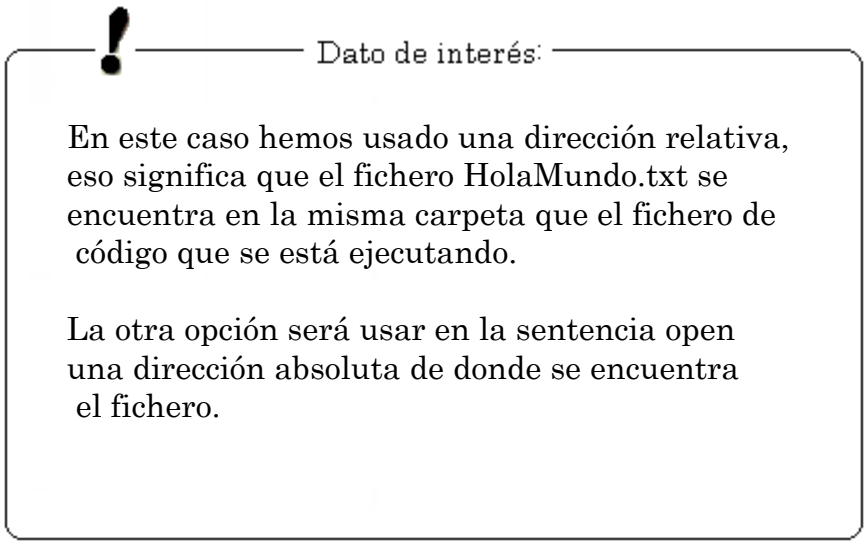
nombreFichero.write('cadenaDeCaracteres')

Ejemplo:

```
>>> fichero1.write('Hola mundo')
```

Veamos como quedaría este ejemplo:

```
>>>fichero1.open('HolaMundo.txt', 'w')
>>>fichero1.write('Hola Mundo desde Python')
>>>fichero1.close()
```



Ejemplo:

```
>>>fichero1.open('c:\HolaMundo.txt','w')
```

Para este caso hemos usado el modo 'w', que es de escritura, también podemos usar el modo lectura y escritura ('a') y el modo añadir ('a')

Lectura de un fichero:

Lectura de un fichero:

Sintaxis:

VariableDeSalida = nombreFichero.read()

Ejemplo:

```
>>>cadena = fichero1.read()
```

Veamos como quedaría este ejemplo:

```
>>>fichero1.open('HolaMundo.txt', 'r')
>>>cadena=fichero1.read()
>>>print cadena
>>>fichero1.close()
```

Modo de recorrer un fichero:

Para recorrer un fichero de texto usaremos la sentencia ya vista for-in, la cual nos facilita mucho la operación.
Por cada iteración nos dará una línea del fichero hasta que lleguemos la final.

Ejemplo:

```
>>>fichero1.open('HolaMundo.txt', 'r')
>>>for cadena in fichero1:
>>>    print cadena
>>>    fichero1.close()
```

Cuando usamos un fichero para crearlo, si está lo borra, en caso de que no esté lo crea, el problema viene en la lectura, ya que si intento acceder a un fichero, y este no existe se crea una excepción y nos pararía el programa

Una manera de evitar este problema es con un método de la clase "os", exactamente el método path.exist(), que nos dice si existe una ruta que le pasemos por parámetro.

El ejemplo anterior quedará así:

```
>>>from os import path
>>>if path.exists('c:\HOLAMUNDO.txt'):
>>>for cadena in fichero1:
>>>     print cadena
>>>    fichero1.close()
```

Si no comprobáramos la existencia del fichero y no estuviese, nos saltará la excepción: IOError, la cual podemos de capturarla con try-excpetion como ya hemos vistos.

Manejo de base de datos Oracle:

Para el uso de base de datos Oracle utilizaremos un espacio de nombre llamados "cx_Oracle", el cual tendremos que guardar en nuestro ordenador, estas librerías están bajadas de la página:
" *http://www.python.net/crew/atuining/cx_Oracle/* "y son especificas para cada tipo de máquina y tipo de servidor Oracle que tengamos instalado.

Después de instalar el archivo estamos preparado para manejar una base de datos Oracle desde Python.

Datos básicos:

Para manejar la librería que acabamos de instalar, hemos de decirle a nuestro programa donde se encuentra. Esto lo haremos con la sentencia sys.path.insert (después de haber importados sys):

Sintaxis:

sys.path.insert(posición,ruta)

Ejemplo:

```
>>>sys.path.insert(0,'C:\Python24\Lib\site-packages')
```

Ahora podemos insertar cx_Oracle sin problemas.

Conexión:

Para realizar la conexión de nuestro programa con nuestra base de datos tenemos que tener en cuenta todos los parámetros que necesitamos:

Un objetos dsn que tendrá la información de cómo encontrar la base de datos. A este método accederemos con la sentencia cx_Oracle.makedsn:

Sintaxis:

variableDeSalida = cx_Oracle.makedsn('direcciónIpDeLaMáquina', puertoDeEscuchaDeLaBaseDeDatos, 'nombreBaseDeDatos')

Ejemplo:

```
>>>dsn = cx_Oracle.makedsn('127.0.0.1', 1521, 'DB1')
```

Ahora debemos de saber los datos personales para acceder a la base de datos en cuestión. Estos parámetros los necesitamos para crear un objeto de tipo conexión, donde también debemos darle el dsn que acabamos de crear.

Sintaxis:

variableDeSalida = cx_Oracle.connect('nombreUsuario','contraseña',dsn)

Ejemplo:

```
>>>conn = cx_Oracle.connect('AntonioMontaño','miClave',dsn)
```

Ahora ya tenemos creado un objeto de tipo conexión apto para entrar en nuestra base de datos.

Cuando creemos una consulta de tipo "SELECT * ...." el resultado será varias tuplas, con lo que necesitaremos algo para movernos de una tupla a otra. Este objeto se llama cursor, y será el encargado de pasarnos de una tupla a otra.

Sintaxis:

variableDeSalida = nombreConexión.cursor()

Ejemplo:

```
>>> cursor = conn.cursor()
```

El objeto cursor tiene un método llamado execute, que le entra un parámetro de tipo cadena, esta será la consulta que queramos hacerle a la base de datos.

Sintaxis:

nombreCursor.execute(consultaSql)

Ejemplo:

```
>>>cursor.execute('select * from Empleados')
```

Ahora tenemos que (en el caso de que la consulta retorne algo) devolverlas una a una para tratarlas. Esto lo haremos con un método del cursor llamado fetchone.

Sintaxis:

variableDeSalida = nombreCursor.fetchone()

Ejemplo:

```
>>>row=cursor.fetchone()
```

Ahora podemos acceder a los datos devuelto por esa tupla, cada vez que accedemos al método fetchone nos devolverá una nueva, excepto que lleguemos al final
Que nos devolvera "*None*".

Los datos devuelto por fetchone nos da una lista con los datos de las columnas que acabamos de extraer. De este modo es muy sencillo acceder a una columna en concreto:

Ejemplo

```
>>>print row[0]
```

Cuando terminemos de usar la conexión y el cursor deberemos cerrarlos para que no tengamos conflictos. Tanto la conexión como el cursor tienen un método close().

Para los tipo de consultas que no devuelven nada, como es el caso de un insert into, no debemos crear un cursor, pero si que tenemos que hacer que la base de datos guarde los cambios realizasdos, usando el método commit() de la conexión.

Sintaxis:

NombreConexión.commit()

Ejemplo:

```
>>>conn.commit()
```

Veamos un ejemplo de conexión con base de datos:

Base de datos:

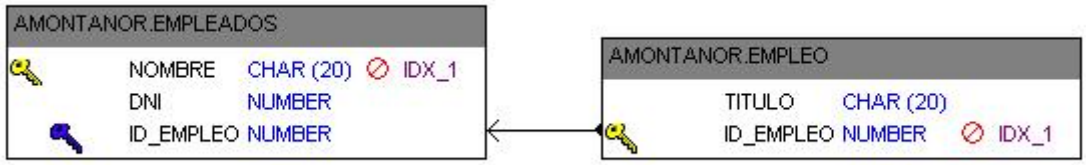
Empleados:

NOMBRE	DNI	ID_EMPLEO
Antonio	28605258	1

Empleo:

TITULO	ID_EMPLEO
Jefe	1
Programador	2

Relaciones:



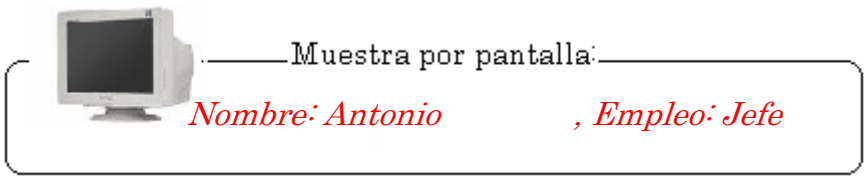
Nombre fichero: "database.py"

```
import sys
sys.path.insert(0,'C:\Python24\Lib\site-packages')
import cx_Oracle
def get_connection():
    dsn = cx_Oracle.makedsn('127.0.0.1', 1521, 'Db1')
    connection = cx_Oracle.connect('AntonioMontaño', 'miClave', dsn)
    return connection
```

Nombre fichero: "consulta.py"

```
import database, sys
conn = database.get_connection()
cursor = conn.cursor()
Nombre = raw_input('Introduce nombre a buscar\n')
sql="select Ems.NOMBRE, Em.TITULO from EMPLEADOS Ems,
EMPLEO Em where Ems.NOMBRE = '"+str(Nombre)+"' and
Ems.ID_EMPLEO = Em.ID_EMPLEO"
cursor.execute(sql)
row = cursor.fetchone()
print 'Nombre: ' + row[0] + '  , Empleo: ' + str(row[1])
cursor.close()
conn.close()
```

Si ejecutamos consulta.py, nos piden un nombre, si ponemos Antonio nos mostrará por pantalla:



Dato de interés:

El problema es que busquemos a alguien que no esté, ya que utilizamos posiciones de row que no existirán. Eso lo podemos comprobar con un simple if.

Vamos a ver el mismo ejemplo, pero en este caso devolveremos más de un elemento. Dejaremos el fichero database.py como estaba.

Base de datos:

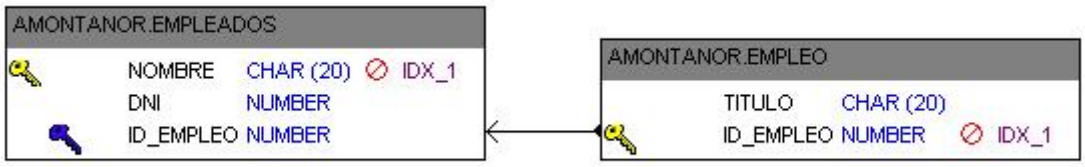
Empleados:

NOMBRE	DNI	ID_EMPLEO
Alejandro	28645456	2
Antonio	28605258	1

Empleo:

TITULO	ID_EMPLEO
Jefe	1
Programador	2

Relaciones:

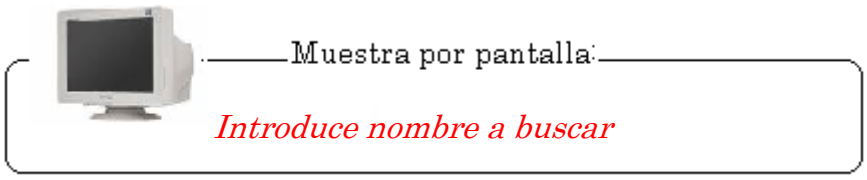


Nombre fichero: "consulta.py"

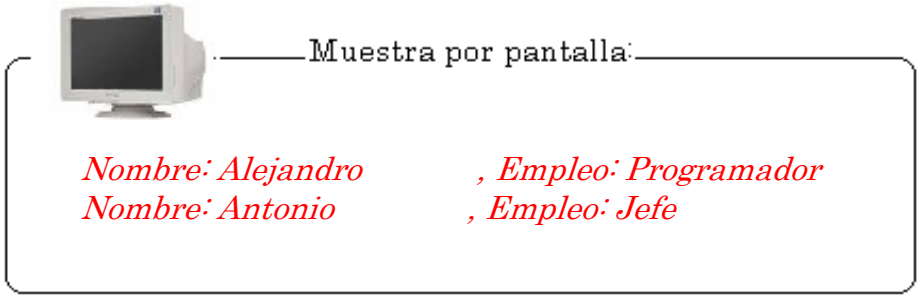
```
import database, sys

conn = database.get_connection()
cursor = conn.cursor()
Nombre = raw_input('Introduce nombre a buscar\n')
sql="select Ems.NOMBRE, Em.TITULO from EMPLEADOS Ems,
EMPLEO Em where Ems.NOMBRE like '%"+str(Nombre)+"%' and
Ems.ID_EMPLEO = Em.ID_EMPLEO"
cursor.execute(sql)
row = cursor.fetchone()

while row != None:
    print 'Nombre: ' + row[0] + '  , Empleo: ' + str(row[1])
    row=cursor.fetchone()

cursor.close()
conn.close()
```

Vemos 2 casos:

1) Lo que escribamos retorna algo, por ejemplo:



Escribimos A, y nos devolverá:



2) En el caso de que no retorne nada, ahora no dará fallo como antes, puesto que en row = cursor.fetchone() nos retornará None y no entraremos en el bucle.