T- TIPOS DE DATOS PYHTON

Contenido:

NOCIONES BÁSICAS Python:

- Todo se trata como un objeto. Con lo cual, sea el tipo que sea lo que declaremos llevará unos atributos y métodos.
- No se declara el tipo de variable. Al inicializarla ya le asignará automáticamente el tipo
- Los tabuladores indican todas las sentencias que pertenecen a un bloque.
- El final de línea no lleva ningún carácter (ni coma, ni punto y coma,…)

NÚMEROS

Los operadores numéricos básicos son la suma (+), resta (−), multiplicación (*), división (/), división entera (//), exponenciación (**) y modulo (%). No existe diferencia entre la división (/) y la división entera (//) si ambos operandos son enteros.

Para el tratamiento de bits se incorporan los siguientes operadores: operación and (&), operación or (|), operación xor (^), operación not (~), desplazamiento a la izquierda (<<) y desplazamiento a la derecha (>>).

Para Python todos los elementos son objetos, por lo que los tipos numéricos también lo son. Los números incorporan un conjunto de métodos que dan acceso a funciones matemáticas avanzadas como la raíz cuadrada (`sqrt`), logaritmos (`log10`), etc.

La documentación oficial de Python muestra todos los métodos, el entorno eclipse los muestra en cuanto tecleamos el punto en una variable con un valor entero.

Ejemplo:

```
# -*- coding: utf-8 -*-
if __name__ == "__main__":
    # Esta línea es un comentario, existen más tipos y se estudiarán más adelante
    # Números
    iNum1 = 23  # Entero como int de C
    fNum1 = 3.34  # float o reales usa doble precisión
    lNum1 = 3L  # Entero largo como long de C
    iNum2 = 027  # Octal
    iNum3 = 0x3F  # Hexadecimal
    cNum1 = 2.1 + 3.1j  # Complejos
    cNum2 = -4.3 - 1j

    print "Valores", iNum1, fNum1, cNum1
    print "Suma de los dos primeros", iNum1 + fNum1
    print "Suma de los dos últimos", fNum1 + cNum1
    print "Suma de dos complejos", str(cNum1) + "+" + str(cNum2) + "=" + str(cNum1 + cNum2)
    print "Octales y hexadecimales", iNum2, iNum3

    from decimal import Decimal, getcontext
    dNum1 = Decimal(fNum1)  # La mayor precisión posible
    print "real y decimal", fNum1, dNum1
    getcontext().prec = 6
    print "Precisión 6:", Decimal(1) / Decimal(7)
    getcontext().prec = 28
    print "Precisión 28:", Decimal(1) / Decimal(7)

    # Operadores +,-+*,/,//,**,%
    print "multiplicación: " + str(iNum1) + "*" + str(fNum1) + "=", iNum1 * fNum1
    print "división: " + str(iNum1) + "/Decimal(" + str(fNum1) + ")=", iNum1 / Decimal(fNum1)
    print "división: " + str(iNum1) + "/" + str(fNum1) + "=", iNum1 / fNum1
    print "división entera: " + str(iNum1) + "//" + str(fNum1) + "=", iNum1 // fNum1
    print "división (solo enteros): " + str(iNum1) + "/" + str(lNum1) + "=", iNum1 / lNum1  # Determina división entera
    print "módulo: " + str(iNum1) + "%" + str(lNum1) + "=", iNum1 % lNum1
    print "exponente: " + str(lNum1) + "**" + str(lNum1) + "=", lNum1 ** lNum1

    # Operadores a nivel de bit: &, |, ^, ~,<<,>>
    bNum1 = 0x9  # 1001
    bNum2 = 0xC  # 1100
    print "And (&)", bNum1 & bNum2  # 1000
    print "Or (|)", bNum1 | bNum2  # 1101
    print "XOR (^)", bNum1 ^ bNum2  # 0101
    print "Not(~)", ~bNum1
    print "Desplazamiento >> de 2", bNum1 >> 2  # 0010
    print "Desplazamiento << de 2", bNum1 << 2  # 0010 0100 tener en cuenta que son 32 o 64 bits la representación

    # Funciones básicas de números
    print "----------Funciones básicas----------"
    print dNum1.copy_abs()
    print dNum1.is_infinite()
    print dNum1.log10()
    print dNum1.sqrt()
    print dNum1.max(lNum1)
    print fNum1.hex()  # representación hexadecimal
    print lNum1.bit_length()  # bits necesarios para representar el valor

    # Booleanos
    # Operadores and, or y not
    print "True and True:", True and True
    print "True and False:", True and False
    print "True or True:", True or True
    print "True or False:", True or False
    print "Not True:", not True

    # Operadores de comparación: ==,!=,<,<=,>,>=
    print "'a'=='a'", 'a' == 'a'
    print "'a'=='b'", 'a' == 'b'
```

BOOLEAN

Se utiliza para describir expresiones condicionales, pudiendo almacenar exclusivamente los valores True o False.

Para este tipo se definen operandos especiales de comparación: comparación y (`and`), comparación o (`or`), negación de la expresión (`not`), junta con los tradicionales de igualdad (==), desigualdad (`!=`), mayor(>), mayor o igual (>=), menor (<) y menor o igual (<=).

CADENA

Podemos representar una cadena mediante comillas simples o mediante comillas dobles, pero nunca mezclando notaciones. Dentro de ambas notaciones es posible introducir valores especiales mediante la barra invertida (\). Así `\n` representara el carácter de nueva línea, `\t` el de tabulación, `\r` el de retroceso de carro, etc.

Si no deseamos que la barra invertida se interprete junto con el siguiente carácter antepondremos r a la definición de la cadena fuera de las comillas (`r"\na\t"`).

Si nuestra cadena está formada por valores UTF-8 usaremos `u` como carácter predecesor.

Un mecanismo nuevo de definición de cadenas son las tres comillas simples al inicio y fin de la definición. De esta manera podremos partir una cadena en varias líneas interpretando los retornos de carro que introduzcamos correctamente.

```
# -*- coding: utf-8 -*-
if __name__ == "__main__":
    sCad1 = "Mi cadena 'ñ'\tes esta"  # la ñ aparece bien por la primera línea del código fuente
    sCad2 = 'Mi Cadena "ñ"\tes esta'
    sCad3 = r"Mi Cadena \t'ñ'es esta"  # raw
    sCad4 = u"Mi cadena 'ñ'\tes esta"  # unicode
    sCad5 = """
        Mi Cadena \t'ñ'es esta"""
    sCad6 = "Valor:"
    sCad7 = "esta frase es de pruebas"
    sCad8 = "-";
    sCad9 = ("a", "b", "c");  # Arrays de cadenas, ver tuplas más adelante
    sCad10 = "El valor de {} + {} es {}"  # Las llaves denotan posición den los parámetros a sustituir 0,1,2
    # Las llaves se escapan duplicándolas ver http://docs.python.org/2/library/string.html#formatstrings

    print "Entre comillas dobles:", sCad1
    print "Entre comillas simples:", sCad2
    print "Cadena raw:", sCad3

    # Funciones básicas
    print "----------Funciones básicas----------"
    print sCad7.capitalize()
    print sCad7.center(50)
    print sCad7.ljust(50)
    print sCad7.rjust(50)
    print sCad7.count("es")
    print sCad7.find("se")
    print sCad7.upper()
    print sCad7.strip()
    print sCad7.split(" ")
    print sCad7.splitlines()
    print len(sCad7)
    print sCad8.join(sCad9);
    print sCad10.format(1, 2, 1 + 2)
```

En el ejemplo se utilizan dos operadores con cadenas. El operador más (+) sirve para concatenar y el operador asterisco (*) crea una copia un número de veces de la cadena.

Las cadenas son también objetos, por lo que se incorporan algunas funciones útiles:

- `capitalize()` devolverá una cadena con la primera letra en mayúsculas,

- `center(num)` centrara la cadena en el número de caracteres que le indiquemos usando blancos,
- `ljust()` y `rjust()` justifican a la izquierda y derecha respectivamente,
- `count(subcadena)` devuelve el número de ocurrencia de la cadena pasada como parametro,
- `find(subcadena)` nos devuelve la primera posición en la que aparece la cadena del parámetro,
- `upper()` convierte a mayúsculas,
- `strip()` elimina los blancos,
- `split(caracter)` divide la cadena en partes utilizando coma separador el carácter pasado,
- `splitlines()` divide según líneas,
- `len(cadena)` longitud del objeto, esta funci6n se puede usar con cualquier objeto para encontrar su tamaño,
- `join(cadena)` hace una unión de las dos
- `format(valores)` formatea la cadena de entrada según la expresión contenida con las valores que se pasan.

Las dos primeras líneas del ejemplo indican:

- que el archivo se guarde en formato UTF-8, permitiendo caracteres acentuados y demás signos
- el bloque `if` hará que el resto del código (sangrado cuatro espacios formando un bloque) se ejecute exclusivamente cuando lo lancemos de forma directa, no cuando sea importado desde otro fichero Python (import)

EJ1

DICCIONARIOS O TABLAS HASH

Tipo de datos parecido a los arrays en los cuales se accede a los datos a través de una clave (y no una posición)

La definici6n de un diccionario se realiza mediante llaves (`{}`), separando cada registro por una coma (`,`) y la clave de su valor par dos puntos (`:`)

Una vez creada tendremos acceso a cada uno de los elementos mediante la notación de corchetes (`dVal["clave"]`) en donde el valor a utilizar dentro será el nombre o valor de la clave usado en la definición, nunca una posición. Se utilizaran claves de los tipos básicos numéricos o cadenas teniendo en cuenta que la clave es sensible a las mayúsculas, ningún otro es aceptado siendo posible mezclar los tipos de las claves en un mismo diccionario.

La clave no se podrá repetir ni variar una vez creada, pero su contenido se puede modificar en cualquier momento usando los corchetes (`dVal["clave"]=valor`).

Ejemplo:

```
#-*-coding:     utf-8 -*-
'''
Created on 22/11/2015

@author: Amaia
'''

if __name__ == '__main__':
    #DEFINICIÓN
'
    dVal1  = {"key1":"valor1","key2":"valor2"}
    print  "Diccionario:",dVal1
    print "Diccionario  valor con clave key2 d['key2']:",dVal1['key2']
    # Se puede  usar cualquier  tipo de valor,  y en las claves  tipos  básicos
    dVal2  = {1:"Valor cadena","keyABC":23}
    print  "Diccionario:",dVal2
    print  "Diccionario valor con clave1 d['1']:",dVal2[1]
    print  "Diccionario  valor con clave keyABC d('keyABC']:",dVal2['keyABC']

    # Es dinámico  pero sin claves duplicadas
    print  "Diccionario  AÑADIR"
    print "Original",dVal2
   #Añado uno nuevo con clave sin duplicar
    dVal2[2]=6+7j
    print "Nuevo añadido",dVal2
    print "\r"

    print  "Diccionario  MODIFICAR:"
    print  "Original",dVal2
    #Utilizo una de las claves ya existentes para reasignar
    dVal2[2]= "ABC"
    print  "Modificado",dVal2
    print "\r"

    print  "Diccionario   BORRAR:"
    #comando del seguido del diccionario y la clave
    print  "Original",dVal2
    del dVal2[1]
    print  "Diccionario   borrado clave 1:",  dVal2
```

```
    print "\r"
    print "\r"
    print "\r"
    print "Diccionario vaciar- clear:"
    print  "Original",dVal2
    dVal2.clear()
    print  "Tras vaciado",dVal2

    print  "Diccionario   borrado con del:",  dVal2
    del dVal2


# Funciones básicas
    print "--------------Funciones  basicas----------------"
    dVal2={1:"Valor cadena","keyABC":23}
    print  len(dVal2)
    print dVal2.items()          # Lista  de todo
    print dVal2.keys()           # Lista de las claves
    print dVal2.values()         #Lista de los valores
    print dVal2.has_key("1")     # ¿Existe  la clave 1?.Es una cadena  de texto
debe dar falso
```

LISTAS

Estructuras tipo tabla que son dinámicas (pueden cambiar de número de elementos). La definición de una lista se realiza encerrando entre corchetes los valores que necesitamos almacenar, siendo estos de cualquier tipo posible (`lVal1=["valor1",2, 2+4j]`), incluso otras tablas, diccionarios, etc. El acceso y modificación del contenido de una de las posiciones se realizara por un índice numérico, empezando dichos valores en cero.

Pero los índices en las listas de Python no tienen por qué ser solo positivos, se pueden utilizar valores negativos indicando que se empiece a contar desde el final. La posición `lVal1 [-1]` sería la última, -2 la antepenúltima y así sucesivamente.

Además, el conseguir varios elementos consecutivos de la lista como una nueva es un proceso muy sencillo, indicando los índices inicial y final a conseguir (el final no se incluye) separándolos por dos puntos (`:`). Este mecanismo también permite utilizar números negativos para indicar la posici6n inicial y final, así como no establecer cualquier índice, si deseamos coger desde el primero o hasta el último elemento.

```
Lista [pos_inicial:pos_final]

Lista [pos_inicial:]

Lista[]
```

Para las listas se definen solo dos operadores, el operador concatenación (+) y el operación de repetición (*) similar a los diccionarios.

Las listas son dinámicas, por lo que puede variar el número de elementos que la constituye durante la ejecución del programa. Para implementar este mecanismo se han dotado al tipo de una serie de métodos que lo facilitan. Para aumentar los componentes utilizaremos uno de los tres métodos existentes para tal fin:

- `append(valor)`: añade el valor pasado al final de la tabla incrementando el ultimo índice
- `insert(posición,valor)`: la segunda función permite insertar en una posición especifica un valor, desplazando el resto.
- `extends(iterable)`: concatena la lista actual a la que se le pasa creando una nueva con todos los valores, este método implementa la misma funcionalidad que el operador concatenación (+) pero de una forma mucho más eficiente.

Al igual que aumentar el número, tenemos varios métodos para disminuir los componentes:

- `pop(indice)`: extraerá el último de la lista si no se proporciona un índice o el elemento del índice indicado similar a: `del lVal[indice]`
- `remove(valor)`: quitará el primero que concuerde con el valor produciéndose un error en caso de que no exista coincidencia.

Una lista si tiene orden, el que se encuentra al recorrer la lista desde la posición inicial hasta la final de forma ascendente. Como el orden es importante tenemos un par de mecanismos para realizar búsquedas sobre los contenidos:

- método `index(valor)`: devolverá la posición en la lista del valor pasado.
- Uso de la palabra clave in para determinar si un valor se encuentra en alguna posición (`"c" in lVal1`).

Una función muy interesante para las listas es la posibilidad de aplicar una operación (un mapeo) a cada elemento de la lista a la vez que determinamos si dicho componente formará parte de la nueva lista .

La estructura tiene la siguiente sintaxis:

[operación ***for*** variable ***in*** lista condición_opcional]

Ejemplo:

```
[varEle*2 for varEle in listaValores if varEle%2!= 0]
```

para cada elemento valEle en la listaValores que sea impar (`varEle%2!=0`) devolverlo multiplicado por dos (`varEle*2`).

Ejemplo:

```
#-*- coding: utf-8 -*-
'''
Created on 22/11/2015

@author: Amaia
'''

if __name__ == '__main__':
    lVal1=["valor1",2,2+4j]
    print "Lista:",lVal1
    print "Lista  valor con índice 2  d(2]:",lVal1[2]

    # Es dinámnica, se le pueden añadir valores
    print("AÑADIR")
    print "Lista original:",lVal1
    lVal1.append(6+7j)
    print "Lista con nuevo valor añadido:",lVal1
    print ("\r\r")
    print("MODIFICAR")
    print "Lista original:",lVal1
    lVal1[3]="Mi  cadena"
    print "Lista  modificada:",lVal1
    print ("\r\r")
    print("BORRAR UN ELEMENTO")
    print "Lista original:",lVal1
    del lVal1[0]  # Base cero
    print "Lista  borrada elemento 0:",  lVal1
    print ("\r\r")
    # Se puede conseguir  partes
    print "Desde  el final índice  negativo",lVal1 [-2] # -1  es el último
    print "Una parte",lVal1[1:]   #desde  el segundo  hasta  el final, empieza
en cero
    print "Una   parte",lVal1[1:2]  # inicio  incluido  índice fin excluido
```

```
    print ("\r\r")
    # Operadores +,+=,*
    print ("SUMA DE LISTAS +")
    print "Dos listas:",lVal1 + lVal1
    print ("REPETICION DE LISTAS con *")
    print "Una lista tres veces:",lVal1 * 3
    print ("OPERADOR +=")
    lVal2=[True]
    lVal2+=lVal1
    print "RESULTADO operador +=",lVal2
    print ("\r\r")
    # Funciones básicas
    print "--------------Funciones básicas----------------"
    print lVal1
    print "Longitud listas",len(lVal1)
    print "Índice del valor 2+4j ",lVal1.index(2+4j) #índice del valor 2+4j
    print "Está el valor 3 en la lista? ",3 in lVal1   # 3 está en la lista?
    print "Borramos el valor 2 con remove"     # borramos por valor, no por
índice
    lVal1.remove(2)
    print lVal1
    print "Eliminar el último con pop"   # Extrae el último, no tiene
parámetro
    lVal1.pop()
    print lVal1
    print "Insertar en la posición final" # Insertamos en la posici6n final
    lVal1.insert(len(lVal1),"45")
    print lVal1
    print "Insertar al final varios valores: 1,2,3 con extend" # Insertamos
todos los del objeto iterable (en este caso la lista [1,2,3])
    lVal1.extend([1,2,3])
    print lVal1
    print ("\r\r")
    # mapeo de listas, ejecutar una acción sobre todos los elementos
    print "--------------mapeo----------------"
    lVal2  = [1,  2, 3, 4, 5]
    print "Original:",lVal2
    print "Todo por dos:",[ele * 2 for ele in lVal2]
    print  "Los impares   por dos",[ele*2 for ele in lVal2 if ele%2!=0]
```

TUPLAS

Las tuplas se interpretan como listas inmutables tanto en tamaño como en contenido.

Esta función permite que sean más ligeras que las listas y mucho más eficientes, a cambio de ser invariables.

Para definir una tupla cambiaremos los corchetes de las listas por ***paréntesis*** (`tVal1 = ("valor1", 2, 2 + 4j))`

Accederemos de la misma manera que las listas, usando los corchetes para tal fin (`tVal1 [2]`) evitando usar una asignación ya que no está permitido (`tVal1 [2]=3 error`).

Otra característica que comparte con las listas es el uso de los índices para seleccionar un elemento o un rango (índices negativos, uso de los dos puntos, base cero, etc.)

Las tuplas al ser objetos fijos no tienen los métodos que permiten variar el contenido o el tamaño existentes en las listas, solo está presente el método index(valor) que devolverá la posición del elemento que concuerde con el valor pasado.

Las listas y las tuplas son tipos de datos similares en cuanto funcionamiento, por lo que es factible crear una a partir de la otra. Así, si tenemos una lista con la llamada a la función `tuple(var_lista)` formaremos una tupla desde la lista pasada como parámetro. Si necesitamos una lista a partir de una tupla, llamaremos a la función `list(var_tupla)` con la que conseguiremos una lista, modificable por tanto, desde una tupla ya existente.

Ejemplo:

```
#-*- coding: utf-8 -*-
'''
Created on 22/11/2015

@author: Amaia
'''
if __name__ == '__main__':
    # Definici6n
    tVal1=("valor1",2,2+4j)

    print "Tupla:",tVal1
    print "Tupla valor con índice 2 d(2]:", tVal1[2]

    # No Es dinámico
    # Se puede conseguir partes
    print "Desde el final índice negativo", tVal1[-2] # -1 es el
ultimo
    print "Una parte con inicio marcado:",tVal1[1:]
    print "Una parte con inicio y fin marcados",tVal1 [0:2] # inicio
incluido índice fin excluido

    # Operadores +,+=,*
    print "Dos tuplas:",tVal1 + tVal1
    print "Una tupla tres veces:",tVal1*3

    # Funciones básicas
    print "--------------Funciones básicas---------------"
    print "Original:",tVal1
```

```
    print "Longitud con len:",len(tVal1)
    print "Índice del valor 2+4j:",tVal1.index(2+4j)
    print "Está el valor 3 en la lista?:",3 in tVal1   # Valor 3 en la
tupla?

    # mapeo de tuplas, ejecutar una acción sobre todos los elementos
    print "--------------mapeo----------------"
    tVal2 = (1,2,3,4,5)
    print "Original: ",tVal2
    print "Todo  por dos",  tuple([ele   * 2 for ele  in tVal2])   #
generarnos una tupla desde una lista con tuple. Al revés list print
"Los  impares  por dos",  tuple([ele*2 for ele in tVal2 if ele %2!=0)
```

RESUMEN:

- Diccionarios con {}
- Listas con []
- Tuplas con ()

EJ2