HOJA DE REFERENCIA - PYTHON - SOLO LO BÁSICO

GENERAL

Distingue entre mayúsculas y minúsculas. El índice comienza desde 0. Espacios o tabulaciones para bloques de código.

AYUDA

Página de inicio de ayuda	`help()`
Ayuda de función	`help(str.replace)`
Ayuda de modulo	`help(re)`

MÓDULO (O LIBRERÍA)

Es simplemente un archivo con extensión .py

Lista contenido de modulo	`dir(modulo1)`
Cargar un modulo	`import modulo1 # *`
Llamar función de modulo	`modulo1.func1()`

***** crea un nuevo espacio de nombres y ejecuta todas las declaraciones en el archivo .py asociado dentro de ese espacio de nombres. Si desea cargar el contenido del módulo en el espacio de nombres actual, use '**from module1 import ***'

TIPOS DE ESCALARES

Verificar el tipo de datos: `type(variable)`

SEIS TIPOS DE DATOS DE USO COMÚN

1. **int/long*** - Large int se convierte en long
2. **float*** - 64 bits, no hay tipo 'doble'
3. **bool*** - True o False
4. **str*** - valor ASCII en 2.x y Unicode en 3
   - Entre comillas simples / dobles / triples
   - Se trata como otras secuencias
   - Carácter especial con **** o prefacio con **r**

     ```
     str1 = r'this\f?ff'
     ```
   - Formateo de varias maneras

     ```
     plantilla = '%.2f %s haha $%d'
     str1 = plantilla % (4.88, 'hola', 2)
     ```

*** str(), bool(), int() y float()** también son funciones de conversión de tipo explícito.

5. **NoneType(None)** - valor 'nulo' (SOLO existe una instancia del objeto None)
   - **None** no es una palabra clave reservada, sino una instancia única de '**NoneType**'
   - **None** es un valor predeterminado para argumentos de funciones opcionales:

     ```
     def func1(a, b, c = None)
     ```
   - Uso común de None:

     ```
     if variable is None:
     ```
6. **datetime** - mod. integrado "fecha y hora" brinda los tipos 'datetime', 'date', 'time'.
   - '**datetime**' combina 'date' y 'time'

Crea obj	`dt1=datetime.strptime('20200722', '%Y%m%d')`	
Obtiene el obj	`dt1.date()`	`dt1.time()`
datetime a cadena	`dt1.strftime('%m/%d/%Y %H:%M')`	
Cambiar val	`dt2 = dt1.replace(minute=0,second=30)`	
Obtener dif.	`diff=dt1-dt2 # 'datetime.timedelta'`	

Nota: La mayoría de obj. son mutables, excepto cadenas y tuplas.

ESTRUCTURAS DE DATOS

NOTA: Todas las llamadas a funciones que no son Get, ej: list1.sort() son operaciones in situ (sin crear un nuevo objeto) a menos que se indique lo contrario.

TUPLAS

Secuencia unidimensional, de longitud fija e **inmutable** de objetos de **CUALQUIER** tipo.

Crear	`tup1 = 4,5,6`	`tup2 = (6,7,8)`
Crear anidada	`tup1 = (4, 5, 6), (7, 8)`	
Convertir secuencia/iterator a tupla	`tuple([1,0, 2])`	
Concatenar	`tup1 + tup2`	
Desempaquetar	`a, b, c = tup`	
Intercambiar variables	`b, a = a, b`	

LISTAS

Secuencia unidimensional, de longitud variable, **mutable** (modificable) de objs. de **CUALQUIER** tipo.

Crear	`list1=[1,'a',3]`	`list2 = list(tup1)`
Concatenar	`list1 + list2`	`list1.extend(list2)`
Agregar al final de lista		`list1.append('b')`
Insertar en posición	`list1.insert(posIdx, 'b') # **`	
Inverso insertar	`valueAtIdx=list1. pop(posIdx)`	
Eliminar primer valor de la lista		`list1.remove('a')`
Verificar existencia	`3 in list1 => True # ***`	
Ordenar	`list1.sort()`	
Ordenar con la función de usuario	`list1.sort(key = len)` `# ordenar por len`	

***** La concatenación usando '+' es costosa ya que se debe crear una nueva lista y copiar los objetos. Es preferible extend().
****** Insertar también es costoso en comparación con append.
******* Comprobar que una lista contiene un valor es mucho más lento que los diccionarios y conjuntos, ya que se realiza un escaneo lineal donde otros (tablas hash) en tiempo constante.

Módulo ' bisect incorporado ‡

- Búsqueda e inserción binarias en lista ordenada.
- 'bisect.bisect' busca la ubicación, donde 'bisect.insort' realmente se inserta en esa ubicación.

‡ **ADVERTENCIA:** las funciones del módulo bisect no verifican si la lista está ordenada, ya que sería muy costoso computacionalmente. Por lo tanto, usarlos en una lista no ordenada tendrá éxito sin errores, pero da lugar a resultados incorrectos.

CORTE POR TIPOS DE SECUENCIA †

† Tipos de secuencia: 'str', 'array', 'tuple', 'list', etc.

Notación	`list1[inicio:detener]` `list1[inicio:detener:paso] # §`
§ Toma cada 2 elementos	`list1[::2]`
§ Invertir una cadena	`str1[::-1]`

NOTA: 'inicio' y 'detener' son opcionales; Incluye el índice 'inicio', pero 'detener' NO.

DICCIONARIOS (HASH MAP)

Crear	`dict1 ={'clave1' :'valor1',2 :[3, 2]}`
Crear de secuencia	`dict(zip(keyList, valueList))`
Obtener, Fijar o Insertar elemento	`dict1['clave1'] # *` `dict1['clave1'] = 'newValue'`
Obtener con valor def	`dict1.get('clave1',defVal) # **`
Verificar existencia	`'clave1' in dict1`
Eliminar elemento	`del dict1['clave1']`
Obtener lista de claves	`dict1.keys() # ***`
Obtener lista de valores	`dict1.values() # ***`
Actualizar valores	`dict1.update(dict2)` `# valores dict2 a dict1`

***** Excepción 'KeyError' si la clave no existe.
****** 'get()' por defecto (no 'defaultValue') devolverá 'None' si la clave no existe.
******* Devuelve las listas de claves y valores en el mismo orden. El orden no es un orden en particular, es muy probable que no esté ordenado.

Tipos de clave de diccionario válidos

- Las claves deben ser inmutables como los tipos escalares (int, float, string) o tuples (todos los objetos en la tupla también deben ser inmutables)
- El término técnico aquí es 'hashability', compruebe si un objeto es hashable con el hash ('cadena'), hash ([1, 2]); esto fallaría.

CONJUNTOS (SET)

- Colección **desordenada** de elementos **ÚNICOS**.
- Son como los diccionarios pero solo con claves.

Crear	`set([3, 6, 3])`	`{3, 6, 3}`
Verificar si es subconjunto	`set1.issubset(set2)`	
Verificar si es superconjunto	`set1.issuperset(set2)`	
Verificar si es mismo contenido		`set1 == set2`
Unión (or)		`set1 | set2`
Intersección (and)		`set1 & set2`
Diferencia		`set1 - set2`
Diferencia simétrica (xor)		`set1 ^ set2`

FUNCIONES

Argumentos de la función se pasan por referencia.

- Forma básica

```
def func1(posArg1, keywordArg1 = 1, ...):
```

NOTA: Los argumentos de palabras clave DEBEN seguir argumentos posicionales; NO es "evaluación perezosa", las expresiones se evalúan de inmediato.

- Mecanismo de llamada de función:
  1. Las funciones son locales para el alcance del nivel del módulo.
  2. Internamente, los argumentos se empaquetan en una tupla y dict, la función recibe una tupla 'args' y dict 'kwargs' y se desempaqueta.
- Uso común de 'Las funciones son objetos':

```
def func1(ops=[str.strip,user_def_func, ..], ..):
    for function in ops:
        value = function(value)
```

VALORES DE RETORNO

- **None** se devuelve si se llega al final de la función sin una declaración de retorno.
- Valores múltiples regresan por UN objeto tupla.

```
return (valor1, valor2)
valor1, valor2 = func1(..)
```

ANONYMOUS (LAMBDA)

- Función que consiste en una sola declaración.

```
lambda x : x * 2 # def func1(x):return x * 2
```

- Aplicación de funciones lambda: 'curring', también conocido como derivar nuevas funciones de las existentes mediante la aplicación de args. parciales.

```
ma60 = lambda x : pd.rolling_mean(x, 60)
```

FUNCIONES UTILES PARA ESTRUCTURAS DE DATOS

1. **Enumerate** - devuelve una secuencia (i, valor) de tuplas donde i es índice del elemento actual.

   ```
   for i, value in enumerate(coleccion):
   ```
2. **Sorted** - devuelve una nueva lista ordenada de cualquier secuencia

   ```
   sorted([2, 1, 3]) => [1, 2, 3]
   sorted(set('abc bcd')) => [' ','a','b','c','d']
   ```
3. **Zip** – empareja elementos de listas, tuplas u otras secuencias creando lista de tuplas:

   ```
   zip(seq1, seq2) =>
   [('seq1_1', 'seq2_1'), (..), ..]
   ```
   - Puede tomar un número arbitrario de secuencias. El número de elementos que produce es determinado por la sec. 'más corta'.
   - Iteración simultánea sobre múltiples secuencias:

     ```
     for i, (a, b) in enumerate(zip(seq1, seq2)):
     ```
   - Convertir lista de filas en una lista de columnas:

     ```
     seq1, seq2 = zip(*zipOutput)
     ```
4. **Reversed** – itera sobre los elementos de una secuencia en orden inverso.

   ```
   list(reversed(range(10))) # *
   ```

***** reversed() devuelve el iterador, list() lo convierte en una lista.

CONTROL Y FLUJO

1. Operadores para condiciones en 'if else':

Verifica si dos variables son el mismo objeto	`var1 is var2`
. . . son objetos diferentes	var1 is not var2
Verifica si dos variables tienen el mismo valor	var1 == var2

ADVERTENCIA: Utilice los operadores 'and', 'or', 'not' para condiciones compuestas, no &&, ||, !.

2. Uso común del operador 'para':

Iterando sobre una colección (lista o tupla) o un iterador	`for elemento in iterator:`
. . Si los elementos son secuencias, se pueden 'desempaquetar'	`for a, b, c in iterator:`

3. 'pass' - declaración de no operación. Se utiliza en bloques donde no hay acciones.
4. Expresión ternaria - no bulliciosa 'if else'

   ```
   v=true-expr if condition else false-expr
   ```
5. Sin declaración de switch/case, use if/elif.

PROGRAMACIÓN ORIENTA A OBJETOS

1. '**objeto**' es la raíz de todos los tipos
2. Todo (número, cadena, función, clase, módulo, etc.) es un objeto, cada objeto tiene un 'tipo'. La variable de objeto es un puntero a su ubicación en la memoria.
3. Los objetos son contados por referencia.

   ```
   sys.getrefcount(5) => x
   a = 5, b = a # Crea una 'referencia' al obj derecho de =, => a y b apuntan a 5
   sys.getrefcount(5) => x + 2
   del(a); sys.getrefcount(5) => x + 1
   ```
4. **Class** – forma basica:

   ```
   class MiObjeto(object):
       # 'self' es 'this' de Java/C++
       def __init__(self, name):
           self.name = name
       def miembroFunc1(self, arg1):
           ..
       @staticmethod
       def classFunc2(arg1):
           ..
   obj1 = MiObjecto('name1')
   obj1.miembroFunc1('a')
   MiObject.classFunc2('b')
   ```
5. Útil herramienta interactiva:

   ```
   dir(var1) # Lista todos los métodos del obj.
   ```

OPERACIONES DE CADENA COMUNES

Lista / tupla concatenada con separador	`','.join(['v1','v2','v3'])` `# => 'v1, v2, v3'`
Formateo de cadena	`s1 = 'Mi nombre es {0} {nombre}'` `ns1=s1.format('Martin',nombre='Nelbren')`
Dividir	`sl1=ns1.split(' ') #=> ['Mi', 'nombre',... ]`
Obtener subcadena	`ns1[1:8] # => 'i nombr'`
Relleno de cadena con ceros	`month = '5'` `month.zfill(2) => '05'`

MANEJO DE EXCEPCIONES

1. Forma básica

   ```
   try:
       ..
   except ValueError as e:
       print e
   except (TypeError, AnotherError):
       ..
   except:
       ..
   finally:
       ..
   ```
2. Disparar la excepción manualmente

   ```
   raise AssertionError # Error de aserción
   raise SystemExit # solicitar salida de programa
   raise RuntimeError('Mensaje de error: ..')
   ```

COMPRENSIONES LISTAS, CONJUNTOS Y DICCIONARIOS

Sintáctico para código sea más fácil de leer y escribir

1. **Comprensiones de Lista**
   - Forme de manera concisa una nueva lista filtrando los elementos de una colección y transformando los elementos que pasan el filtro en una expresión concisa.

     ```
     [expr for val in coleccion if condicion]
     ```

     Un atajo de:

     ```
     resultado = []
     for val in coleccion:
         if condicion:
             resultado.append(expr)
     ```

     La condición del filtro puede omitirse, dejando la expresión.
2. **Comprensiones de Diccionarios**

   ```
   {key-expr : val-expr for val in coleccion if condicion}
   ```
3. **Comprensiones de Conjuntos**

   Igual que la lista, excepto con {} en lugar de []
4. **Compresiones de Listas anidadas**

   ```
   [expr for val in coleccion for innerVal in val if condic]
   ```

v0.0.1 – 2020-07-23 – nelbren.com
Presentación y adaptación por **Martin Cuellar**
Original creado por **Arianne Colton** y **Sean Chen**