Departamento de Informática
Universidad Técnica Federico Santa María

Ayudantía

Programación de Computadores

IWI-131

Tema: Resumen Certamen 2

Fecha: 23/10/2017

Autor: Gonzalo Fernández - **@DevTotal**

1. Inicialización y Lectura de Listas

Las listas son un tipo de variable en Python que nos permite almacenar datos de manera ordenada (comenzando desde el **0**), sin importar el tipo de estos (pueden incluso ser otras listas):

Podemos inicializar una lista de 3 maneras:

`lista = []`	Inicializar lista vacía.
`lista = list()`	Inicializar lista vacía.
`lista = [1, 3, 4]`	Inicializar lista con valores.

El acceso a los elementos de una lista se realiza de la siguiente manera:

```
lista[inicio:final:salto]
```

- `inicio`: Posición del elemento desde donde se comenzará a leer la lista.
- `final`: Posición del elemento **posterior** al elemento donde se terminará de leer la lista.
- `salto`: Intervalo de elementos por los que se va a ir leyendo la lista, si es **negativo** entonces la lista será leída de derecha a izquierda.

Se pueden omitir valores de modo de conseguir partes específicas de la lista.

En los siguientes ejemplos se entenderán mejor estos conceptos habiendo definido `lista = [5, 3, 2, 1, 9, 3, 4]`:

```
lista[0]     # 5
lista[1]     # 3
lista[999]   # Error
lista[-1]    # 4, notar que es el último elemento
lista[-3]    # 9, notar la posición de este
elemento.
```

```
# Recordar que lista = [5, 3, 2, 1, 9, 3, 4]

lista[1:1]   # [], notar que entre el inicio (1) y
             el final (1), no hay elementos, pues,
             el final, es el elemento posterior al
             último elemento que se consultará, por
             ende, entre el inicio y el último
             elemento a consultar, no hay nada.

lista[1:1:1] # [], leer nota anterior, notar que si
             salto es 1, no altera la lectura de
             elementos de la lista.

lista[1:2]   # [3], notar que 2 es el elemento
             posterior al elemento del inicio.

lista[-1:1]  # [], notar que esto equivale a
             consultar entre el elemento -1 y el
             elemento 0 de la lista (0 porque 1 es
             el elemento posterior), pero como la
             lectura es de izquierda a derecha,
             entre el final y el inicio no hay nada.

lista[-1:7]  # [4], notar que la lista tiene 7
             elementos, sabemos por el ejemplo
             anterior que en la posición -1 está el
             número 4, además, sabemos que 7 es la
             posición posterior al último elemento
             de la lista, que también es el 4, por
             ende, el resultado es [4].

lista[-3:7]  # [9, 3, 4], notar que -3 equivale a la
             posición del número 9, luego, de allí,
             continúa recorriendo la lista hacia la
             derecha, donde estaría el último
             elemento.

lista[-7:7]  # [5, 3, 2, 1, 9, 3, 4], notar que -7
             equivale a la posición del primer
             elemento, luego, de allí, continúa
             recorriendo la lista hacia la derecha
             hasta la posición antecesora al final,
             esto es, el último elemento!.
```

```
# Recordar que lista = [5, 3, 2, 1, 9, 3, 4]

lista[-6:-1]  # [3, 2, 1, 9, 3], notar que el
              elemento -6 es 3 y el elemento -1 es 4
              (el último), entonces, como estamos
              recorriendo la lista de izquierda a
              derecha, obtenemos el resultado
              mostrado.

lista[0:7:-1] # [], notar que ahora el salto es
              negativo, por ende, estamos leyendo la
              lista de derecha a izquierda.

lista[6:0:-1] # [4, 3, 9, 1, 2, 3], notar que como
              leemos de derecha a izquierda, el
              resultado también está en ese orden
              (usé el 6 porque en el inicio se indica
              el elemento exacto desde el que se
              comenzará a leer, por ende, si quiero
              empezar del último, debo usar el 6 que
              es la última posición válida de la
              lista).

lista[6:0:-2] # [4, 9, 2], notar que como aumentó el
              salto, ahora la lectura es cada 2
              números.

lista[6:0:-9] # [4], notar que usamos un salto muy
              grande y por ende, no hay más elementos
              a excepción del primer elemento en el
              que comenzará la lectura.

lista[::2]    # [5, 2, 9, 4], como vimos
              anteriormente, podemos omitir valores
              al momento de leer nuestra lista.

lista[:2]     # [5, 3], omitimos el salto, por ende
              será 1, además, por defecto el inicio
              será 1, lo único que hemos declarado es
              el fin que es la posición del elemento
              posterior a la posición donde se
              terminará de leer la lista.
```

```
lista[0:]     # [5, 3, 2, 1, 9, 3, 4], omitimos el
              fin pero por defecto es el elemento
              posterior al último elemento de la
              lista, por ende, el resultado es toda
              la lista.
```

Resumiendo, sea e_n un elemento en la posición `n` de la `lista`:

- `lista[a]` devuelve el elemento `a` de la lista si `a > 0`, de lo contrario, será el elemento `M + a` con `M` igual al total de elementos de la lista (para la lista que usamos de ejemplo, `M` sería **6**).
- `lista[a:b]` devuelve una lista con los elementos $[e_a, \ldots, e_{b-1}]$ de la lista.
- `lista[a:b:c]` asumiendo un `k` entero que aumenta de 1 en 1, devuelve los elementos $[e_a, e_{(a+c*k)}, \ldots e_{(a+c*k)<(b-1)}]$ de la lista, entendiendo que `k*c` es la forma en que va saltándose números.

2. Operaciones sobre Listas

- Asignación: `lista[posición] = valor`
- Suma de listas: `lista_final = lista1 + lista2`
- Comparar si 2 listas son iguales: `lista1 == lista2`
- Eliminar un elemento de una lista: `del lista[a:b:c]`
- Ordenar una lista: `lista.sort()`
- Invertir una lista: `lista.reverse()`
- Insertar un elemento al final de una lista: `lista.append(valor)`
- Insertar un elemento en una posición dada: `lista.insert(valor, posición)`
- Obtener la posición de un elemento en la lista: `lista.index(valor)`
- Sumar todos los números de una lista: `sum(lista)`
- Comprobar si un elemento está en una lista: `valor in lista`
- Clonación de listas: `lista_nueva = list(lista_a_clonar)`
- Obtener el largo de una lista: `len(lista)`

3. Iterar sobre Listas

Existen 2 maneras de iterar listas, ya sea con el uso del `for`, ó, el `while:`

- Iterar lista con **`for`**:

```
for i in lista:
    print i
```

La variable **`i`** tendrá cada valor de la lista a medida que se itera sobre esta, el resultado del código anterior sería mostrar en pantalla cada valor contenido en la lista.

Ventajas de esta forma:

1. Es fácil de implementar.
2. Nos permite obtener instantáneamente el valor de la lista en una variable.
3. Podemos trabajar el valor de **`i`** sin modificar el valor en la lista.

Desventajas de esta forma:

1. No podemos detener la iteración a menos que usemos **`return`**.
2. No podemos saber la posición del elemento **`i`** en la lista a menos que usemos `lista.index(i).`
3. No podemos avanzar o retroceder en la lista a voluntad.

- Iterar lista con **`while`**:

```
i = 0
while i < len(lista):
    print lista[i]
    i += 1
```

La variable **`i`** tendrá cada posición de cada elemento de la lista, a medida que va aumentando hasta llegar al largo de la lista, notar que una lista de 5 elementos tiene un largo de 5 pero i llegaría a ser 4 ya que la lectura de datos en una lista parte del 0, entonces, i sería 0, 1, 2, 3, 4.

Ventajas de esta forma:

1. Tenemos control sobre la posición donde estamos iterando (aumentando o disminuyendo **`i`**).
2. Podemos detener el while haciendo que la condición de este ya no se cumpla (ejemplo básico: igualar **`i`** a infinito).

Desventajas de esta forma:

1. Cada vez que trabajemos un valor de la lista, debemos llamarlo con `lista[i]` lo cual, si bien no es grave, puede hacer que el código se vuelva ilegible.
2. Trabajar un valor implicaría modificar el valor en la lista, a menos que efectivamente sea ese nuestro objetivo, ó, guardemos el valor extraído de `lista[i]` en otra variable para así trabajar esa variable **.**
3. Es más difícil de usar, pues, según lo que hagamos dentro del while, podemos salirnos de la lista y causar un error, aunque claro, esto dependerá netamente del código ejecutado dentro del while, lo ideal es pensar bien lo que se está realizando.

Luego de esta comparación suele entrar la duda, ¿Es mejor usar el for sobre el while ?, ó, ¿Hay situaciones en que es mejor usar while que for? La respuesta a estas preguntas sólo se las dará la práctica, se pueden realizar exactamente las mismas cosas con ambos, ahora, el uso de uno por sobre el otro depende netamente de lo que le acomode más al programador, a modo personal, me gusta usar for cuando no necesito saber en qué posición de la lista estoy, en cambio, cuando necesito control sobre la posición de la lista donde estoy trabajando, hago uso del while, pero al final, se puede hacer exactamente lo mismo en ambos, ya sea en el for contando la posición en la que nos encontramos de la lista con una variable auxiliar, como en el while, asignando instantáneamente a una variable auxiliar el valor de lista[i].

4. Inicialización y Lectura de Tuplas

Las tuplas son un tipo de variable en Python que nos permite almacenar datos de manera ordenada (comenzando desde el **0**), sin importar el tipo de estos, casi iguales a las listas, pero con la particularidad de que estas son **inmutables**, esto es, que no pueden ser modificadas:

Podemos inicializar una tupla de 3 maneras:

`tupla = ()`	Inicializar tupla vacía.
`tupla = tuple()`	Inicializar tupla vacía.
`tupla = (1, 4, 3)`	Inicializar tupla con valores.

El acceso a los elementos de una tupla se realiza **del mismo modo** que con las listas:

```
tupla[inicio:final:salto]
```

- `inicio`: Posición del elemento desde donde se comenzará a leer la tupla
- `final`: Posición del elemento **posterior** al elemento donde se terminará de leer la tupla.
- `salto`: Intervalo de elementos por los que se va a ir leyendo la tupla, si es **negativo** entonces la tupla será leída de derecha a izquierda.

Se pueden omitir valores de modo de conseguir partes específicas de la tupla, no agrego muchos ejemplos porque es exactamente lo mismo que los ejemplos anteriores con listas, sólo que en vez de devolver otra lista, en estos casos se devolverá otra **tupla**.

```
tupla = (1, 3, 4, 2)      # (1, 3, 4, 2)

tupla[2]                  # Devuelve 4

tupla[0:3]                # Devuelve (1, 3, 4)
```

5. Operaciones sobre Tuplas

- Asignación: **No existe,** las tuplas no pueden modificarse
- Suma de tuplas: `tupla_final = tupla1 + tupla2 # notar que estamos inicializando una tupla nueva, no modificándola.`
- Comparar si 2 tuplas son iguales: `tupla1 == tupla2`
- Eliminar un elemento de una tupla: **No existe,** las tuplas no pueden modificarse.
- Ordenar una tupla: **No existe,** las tuplas no pueden modificarse
- Invertir una tupla: **No existe,** las tuplas no pueden modificarse
- Obtener la posición de un elemento en la tupla: `tupla.index(valor)`
- Sumar todos los números de una tupla: `sum(tupla)`
- Comprobar si un elemento está en una tupla: `valor in tupla`
- Clonación de tuplas (desconozco qué utilidad tendrá esto): `tupla_nueva = tuple(tupla_a_clonar)`
- Obtener el largo de una tupla: `len(tupla)`
- Comparar 2 tuplas: `tupla1 < tupla2 # Notar que esta comparación se realiza elemento a elemento, de izquierda a derecha, uno de los usos más habituales a esta operación es la comparación de fechas, ejemplo:`

  ```
  tupla1 = (2017, 12, 28)
  tupla2 = (2017, 11, 27)
  tupla1 > tupla2
  True # La fecha de tupla1 es mayor a la de tupla2
  ```

- Transformar una tupla en una lista: `list(tupla) # No olvidar.`

6. Iterar Tuplas

Las tuplas se iteran del mismo modo que las listas, ambos métodos son compatibles con las tuplas, pero ojo, siempre hay que tener en cuenta que una tupla no se puede modificar.

7. Inicialización y Lectura de Conjuntos

Los conjuntos son un tipo de variable en Python que nos permite almacenar datos de cualquier tipo **sin orden**, con la restricción de que estos **no** pueden repetirse en un mismo conjunto, de agregar 2 o más elementos iguales, sólo 1 quedará en el conjunto, además, este tipo de variables nos permite aplicar operaciones de conjuntos matemáticos sobre ellas, lo cual será visto con detalle más adelante.

Podemos inicializar un conjunto de 5 maneras:

`conj = set()`	Inicializar conjunto vacío.
`conj = {1, 2, 3}`	Inicializar conjunto con valores.
`conj = set([1, 2])`	Inicializar conjunto con valores (lista).
`conj = set((1, 2))`	Inicializar conjunto con valores (tupla).
`conj = set("Hola")`	Inicializar conjunto con valores (string).

Como se dijo anteriormente, los conjuntos no tienen orden, por ende, no podemos acceder a sus valores mediante operadores como los de las tuplas o listas ([]), más bien, la única forma de acceder a los valores de un conjunto es transformándolos primero a una lista o tupla, ó, iterando el conjunto. El principal uso de los conjuntos es aprovechar el hecho de que si insertamos valores repetidos en ellos, estos no se almacenarán más de 1 vez (esto sería muy útil para contar cuantas letras diferentes tiene una palabra), además, el poder aplicar operaciones de conjuntos matemáticos facilita muchas cosas al momento de querer resolver problemas de intersección de datos u otros.

8. Operaciones sobre Conjuntos

- Agregar un elemento: `conj.add(valor)`
- Eliminar un elemento: `conj.remove(valor)`
- Unión de conjuntos: `a | b # Devuelve un conjunto con todos los elementos de **a** y **b**`
- Intersección de conjuntos: `a & b # Devuelve un conjunto con los elementos que están en **a** y en **b** al mismo tiempo.`
- Resta de conjuntos: `a – b # Devuelve un conjunto con todos los elementos de **a** que no están en **b**.`
- Exclusión de conjuntos: `a ^ b # Devuelve un conjunto con todos los elementos que están en **a** y **b** pero no en ambos al mismo tiempo.`
- Comprobación de subconjuntos: `a < b # Devuelve True en caso de que **a** sea un subconjunto de b`
- Comprobar si un elemento está en un conjunto: `valor in conj`
- Obtener el largo de un conjunto: `len(conj)`

9. Iterar Conjuntos

La única forma de iterar un conjunto es haciendo uso del `for`, de otro modo, es necesario transformar el conjunto a una lista e iterar la lista usando `for` o `while` según encuentren más cómodo.

- Iterar conjunto con **`for`**:

```
for i in conjunto:
    print i
```

10. Inicialización y Lectura de Diccionarios

Los diccionarios son una estructura de datos **no ordenada** de Python que nos permite asignar una **llave** a los datos almacenados en este, de modo que un dato sólo puede ser accedido mediante una única llave.

Podemos inicializar un diccionario de 3 maneras:

`dict = {}`	Inicializar diccionario vacío.
`dict = dict()`	Inicializar diccionario vacío.
`dict = { "A" : 5 }`	Inicializar diccionario con valores.

El acceso a los elementos de un diccionario se realiza de la siguiente manera:

```
diccionario[llave]
```

- `llave`: Llave usada para almacenar un elemento, esta puede ser un string, un número, ó, un flotante.

No hay mucho que detallar, básicamente es necesario conocer la llave de un dato en un diccionario para acceder a este.

11. Operaciones sobre Diccionarios

- Agregar un elemento: `dict[`**`llave_nueva`**`] = valor`
- Eliminar un elemento: `del dict[llave]`
- Obtener el largo de un diccionario (¿Para qué?): `len(dict)`
- Obtener todas las llaves de un diccionario: `dict.keys() # Notar que devuelve todas las llaves en una lista.`
- Obtener todos los valores de un diccionario: `dict.values() #Notar que devuelve todos los valores en una lista.`
- Obtener una lista de tuplas con todas las llaves y valores de un diccionario: `dict.items() #[("llave1" : valor1) ...]`
- Comprobar si existe una llave en un diccionario: `llave in dict`

12. Iterar Diccionarios

Generalmente, para iterar los valores de un diccionario se hace uso del `for`, es posible hacerlo con `while`, pero, en este material sólo se estudiará la iteración con el `for` ya que es más sencillo de programar y entender. Además, es necesario hace uso de las funciones antes estudiadas como `keys()`, `values()` y, `ítems()`:

- Iterar diccionario con **`diccionario.keys()`**:

```
for i in diccionario.keys():
    print diccionario[i]
```

  La variable **`i`** tendrá cada llave del diccionario a medida que itera, de este modo, podemos acceder a los valores pasando la llave en el diccionario.

- Iterar diccionario con **`diccionario.values()`**:

```
for i in diccionario.values():
    print i
```

  La variable **`i`** tendrá cada valor del diccionario a medida que itera, aunque, no tenemos como saber a qué llave pertenece este valor, no es muy recomendable iterar de este modo los diccionarios.

- Iterar diccionario con **`diccionario.items()`**:

```
for llave, valor in diccionario.items:
    print "la llave", llave, "tiene el valor",
    valor
```

  Esta es probablemente la mejor forma de iterar un diccionario, pues, tenemos acceso a la llave y al valor al mismo tiempo, debido a que, en **llave** se almacenará la llave dentro de la tupla en la que estaremos iterando (recordar qué es lo que devuelve **dict.items**), y en **valor**, el valor asociado a esta, así, podemos trabajar muy fácilmente todos los datos del diccionario.

13. Función Range

La función `range()` es una función tan potente que merece toda una sección para ella sola. Esta función nos permite obtener una lista de números ordenada según los parámetros ingresados, casi al igual que en las listas, podemos indicar la forma en la que queremos esta lista de números con 3 parámetros:

```
range(inicio, fin, salto)
```

Donde:

- `inicio`: Número desde el cual comenzará el rango de números a entregar.
- `final`: Número posterior al número donde terminará el rango de números a entregar.
- `salto`: Intervalo de números que se va a ir saltando en el rango de números a entregar.

Se pueden omitir valores ya que por defecto se asume que `inicio` es 0 y `salto` es 1, por ende, si sólo se pasa 1 parámetro, se tomará como el parámetro `final`, y si se pasan 2, se tomarán como `inicio` y `final`.

Ejemplos:

```
range(5)          # [0, 1, 2, 3, 4]
range(4, 9)       # [4, 5, 6, 7, 8]
range(-5, 0)      # [-5, -4, -3, -2, -1]
range(4, 10, 2)   # [4, 6, 8]
range(10, 4, -2)  # [10, 8, 6], notar que el salto es
                  negativo
range(0)          # [], notar que antes del 0, no hay
                  más números ya que el salto por
                  defecto es 1 y esto significa que
                  los números a entregar serán de
                  izquierda a derecha
```

```
range()             # Error
```

Resumiendo:

- `range(a)` devuelve una lista con todos los números del 0 hasta `a-1`.
- `range(a, b)` devuelve una lista con todos los números desde `a` hasta `b-1`.
- `range(a, b, c)` devuelve una lista con todos los números de la forma `a + n*c` (partiendo con `n = 0`) menores a `b`, si `c` es < 0, entonces esta lista irá de derecha a izquierda (partiendo siempre desde `a`)

Recomiendo abrir su intérprete de Python y probar otras posibles combinaciones de estos parámetros para saldar cualquier tipo de duda.