Tipos de datos estructurados

Strings y Listas

Mario González
Facultad de Ingeniería y Ciencias Aplicadas
Universidad de las Américas, Quito, Ecuador

Agosto, 2021

Estructuras de datos

- El objetivo de esta sesión es presentar las estructuras de datos de Python y la mejor manera de utilizarlas.
- Dependiendo de lo que se necesita de una estructura de datos,
  - ya sea de consulta rápida,
  - inmutabilidad,
  - indexación, etc.,
- se puede elegir la mejor estructura de datos para la tarea encomendada.
- La mayoría de las veces combinaremos diferentes estructuras de datos para obtener un modelo lógico y fácil entender.
- Las estructuras de datos de Python son muy intuitivas desde el punto de vista de la sintaxis y ofrecen una gran variedad de operaciones.

Strings I

- Los strings son diferentes de un entero (int), o de un valor de coma flotante (float).
- Los strings son un **tipo de datos compuesto**.
- Dependiendo de lo que estamos haciendo, podemos querer tratar datos compuestos como un elemento único, o podríamos querer acceder a sus partes.

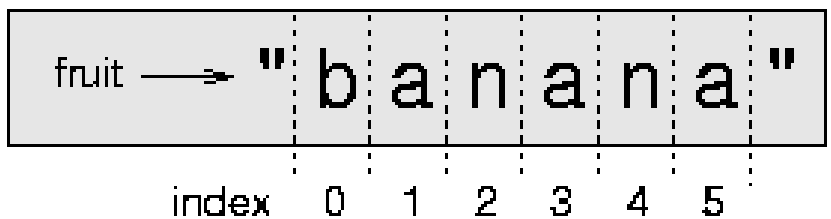
```
fruit = "banana"
letter = fruit[1]
print(letter)
```

Strings II

Longitud

- La función `len` devuelve el número de caracteres en el string.

```
longitud = len(fruit)
ultimo = fruit[length]
```



Strings III

- A menudo necesitamos procesar caracter por caracter del string y recorrerlo de principio a fin:

```
index = 0
while index < len(fruit):
  letter = fruit[index]
  print(letter)
  index = index + 1

for char in fruit:
  print(char)
```

Strings IV

- Recuerden que podemos realizar concatenación de cadenas de caracteres:

```
prefixes = "JKLMNOPQ"
suffix = "ack"

for letter in prefixes:
  print(letter + suffix)
```

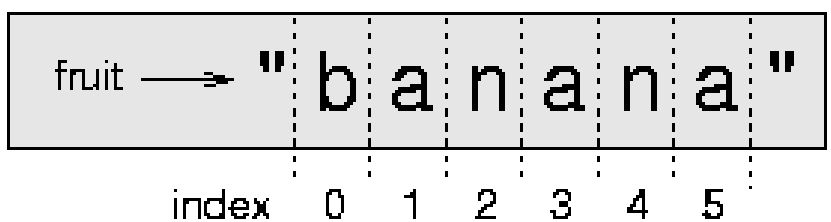
Strings V

Slices

- Un segmento de un string es llamado slice:

```
s = "Peter, Paul, and Mary"
print(s[0:5])
Peter
print(s[7:11])
Paul
print(s[17:21])
Mary
```

- El operador `[n:m]` devuelve el slice del string desde el caracter `n-esimo` al caracter `m-esimo`, incluyendo el primero pero excluyendo el último.



Strings VI

Comparación de strings

- Los operadores de comparación también funcionan con los strings:

```
if word == "banana":
  print("Yes, we have no bananas!")

if word < "banana":
  print("Word," + word + ", is before banana.")
elif word > "banana":
  print("Word," + word + ", is after banana.")
else:
  print("Yes, we have no bananas!")
```

Strings VII

Los strings son inmutables

- Los strings son inmutables, lo que significa que no se puede cambiar un string existente. Lo mejor que puede hacer es crear un nuevo string que es una variación sobre el original:

```
greeting = "Hello, world!"
greeting[0] = 'J'                    # ERROR!
print(greeting)

greeting = "Hello, world!"
newGreeting = 'J' + greeting[1:]
print(newGreeting)
```

Strings VIII

Bucles y conteo

- El siguiente programa cuenta el número de veces que la letra `a` aparece en el string.

```
fruit = "banana"
count = 0
for char in fruit:
  if char == 'a':
    count = count + 1
print(count)
```

- Aquí se demuestra el patrón de computación llamado **contador (counter)**.
  - La variable `count` es inicializada en 0 y es incrementada cada vez que una `a` es encontrada.
  - Incrementar es aumentar en 1, decrementar es reducir en 1, el valor.

Strings IX

El módulo `string`

- El **módulo** `string` contiene funciones útiles para manipular strings. Como ya hemos visto, debemos importar el **módulo** antes de poder usarlo:

```
import string
fruit = "banana"
index = fruit.find("a")
print(index)
1
fruit.find("na")
2
fruit.find("na", 3)
4
"bob".find("b", 1, 2)
-1
```

Strings X

- A menudo es útil examinar un caracter y comprobar si es mayúscula o minúscula, o si es un dígito. El **módulo** `string` provee varias constantes para este propósito:

```
print(string.ascii_lowercase)
print(string.ascii_uppercase)
print(string.digits)
print(string.whitespace)
```

- Podemos usar estas constantes y la función `find` para clasificar caracteres:

```
word = "ePhoRt"

def isLower(ch):
  return string.ascii_lowercase.find(ch) != -1
```

Strings XI

```
def isLower(ch):
  return ch in string.ascii_lowercase

for ch in word:
  print(isLower(ch))

for ch in word:
  if ch in string.ascii_lowercase:
    print("Minuscula")
  elif ch in string.ascii_uppercase:
    print("Mayuscula")
```

Listas I

- Las listas son conjuntos ordenados de valores, donde cada valor se identifica con un índice.
- Las valores que componen la lista son llamados **elementos**.
- Las listas son similares a los strings, los cuales son conjuntos ordenados de caracteres, con la excepción de que las listas pueden contener cualquier tipo de datos.
- Listas, strings y otras estructuras similares, son llamadas **secuencias**.

```
n=[10, 20, 30, 40]
w=["spam", "bungee", "swallow"]
v=["hello", 2.0, 5, [10, 20]]
```

- Una lista dentro de otra lista se dice que está **anidada**.

Listas II

- Las listas que contienen enteros consecutivos son comunes, Python provee una forma simple de generarlas:

```
range(1,5)
[1, 2, 3, 4]
range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
range(1, 10, 2)
[1, 3, 5, 7, 9]
```

Listas III

- Podemos asignar listas a variables y pasarlas como argumentos a funciones:

```
vocabulary = ["mejora", "castiga", "defenestra"]
numbers = [17, 123]
empty = []
print(vocabulary, numbers, empty)
["mejora", "castiga", "defenestra"] [17, 123] []
```

Listas IV

Accediendo a los elementos de una lista

- La sintaxis para acceder a los elementos de una lista es la misma que para acceder a los elementos de un string: corchetes `[]`. Recuerde que lo índices empiezan en 0.
- Si intentamos acceder o escribir un elemento que no existe, obtenemos un error en tiempo de ejecución.
- Si un índice tiene un valor negativo, cuenta hacia atrás a partir del final de la lista.

```
n=[10, 20, 30, 40]
n[2]
30
n[-3]
20
n[1:]
[20, 30, 40]
n[-5]
```

Listas V

- Es común usar una variable (incremental) del bucle como índice:

```
horsemen = ["war", "famine", "pestilence", "death"]

i = 0
while i < 4:
  print(horsemen[i])
  i = i + 1
```

- La variable `i` toma valores de 0 a 4, y termina la ejecución cuando toma este último valor. El cuerpo del bucle es ejecutado solamente cuando `i` es 0, 1, 2 y 3.
- Este patrón de computación es llamado **recorrer la lista**.

Listas VI

- La longitud de la lista puede obtenerse usando la función `len`.

```
horsemen = ["war", "famine", "pestilence", "death"]

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

Listas VII

Pertenencia a una lista

El operador boolenao `in` comprueba si un elemento pertenece a una secuencia.

```
hm = ['war', 'famine', 'pestilence', 'death']
'pestilence' in hm
True
'debauchery' in hm
False
'debauchery' not in hm
True
```

Listas VIII

Listas y bucles `for`

- El bucle for funciona con las listas de la siguiente manera:

```
for VARIABLE in LIST:
  BODY
```

- Esto es equivalente a:

```
i = 0
while i < len(LIST):
  VARIABLE = LIST[i]
  BODY
  i = i + 1
```

Listas IX

- Cualquier expresión de lista puede ser usada en un bucle `for`:

```
for number in range(20):
  if number % 2 == 0:
    print(number)

for fruit in ["banana", "apple", "quince"]:
  print("I like to eat " + fruit + "s!")
```

Listas X

Operaciones sobre listas

- El operador + concatena listas:

```
a = [1, 2, 3]
b = [4, 5, 6]
c = a + b
print(c)
[1, 2, 3, 4, 5, 6]
```

- De forma similar el operador * repita una lista un número dado de veces:

```
[0] * 4
[0, 0, 0, 0]
[1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Listas XI

Slices

- Las operaciones para obtener slices en strings también funcionan en las listas:

```
list = ['a', 'b', 'c', 'd', 'e', 'f']
list[1:3]
['b', 'c']
list[:4]
['a', 'b', 'c', 'd']
list[3:]
['d', 'e', 'f']
```

Listas XII

Las listas son mutables

- A diferencia de los strings, las listas son mutables, lo que significa que podemos alterar sus elementos usando el operador `[]` en la parte izquierda de una asignación:

```
fruit = ["banana", "apple", "quince"]
fruit[0] = "pear"
fruit[-1] = "orange"
print(fruit)
['pear', 'apple', 'orange']
```

- Utilizando operaciones de slice podemos actualizar el valor de varios elementos a la vez:

```
list = ['a', 'b', 'c', 'd', 'e', 'f']
list[1:3] = ['x', 'y']
print(list)
['a', 'x', 'y', 'd', 'e', 'f']
```

Listas XIII

- También podemos remover elementos de una lista asignando una lista vacía:

```
list = ['a', 'b', 'c', 'd', 'e', 'f']
list[1:3] = []
print(list)
['a', 'd', 'e', 'f']
```

- Podemos añadir elementos a una lista usando un slice vacío en el lugar deseado:

```
list = ['a', 'd', 'f']
list[1:1] = ['b', 'c']
print(list)
['a', 'b', 'c', 'd', 'f']
list[4:4] = ['e']
print(list)
['a', 'b', 'c', 'd', 'e', 'f']
```

Listas XIV

Eliminando elementos de una lista

```
a = ['one', 'two', 'three']
del a[1]
a
['one', 'three']

list = ['a', 'b', 'c', 'd', 'e', 'f']
del list[1:5]
print(list)
['a', 'f']
```

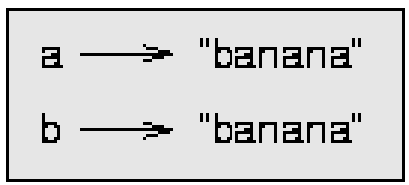
Listas XV

Objetos y valores

- Si ejecutamos las siguientes sentencias:

```
a = "banana"
b = "banana"
```

- sabemos que a y b refieren al string ”banana”, pero hay dos posibilidades:



Listas XVI

- Podemos saber cual es el caso utilizando la función **`id`**, que devuelve el **identificador** único de cada objeto.

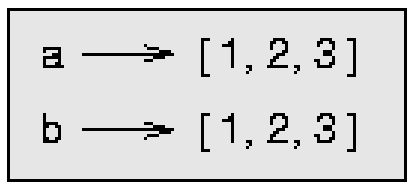
```
id(a)
135044008
id(b)
135044008
```

- Tenemos el mismo identificador dos veces, lo que significa que Python sólo ha creado un string, y a y b se refieren a este.
- Las listas se comportan de manera diferente:

```
a = [1, 2, 3]
b = [1, 2, 3]
id(a)
135045528
id(b)
135041704
```

Listas XVII

El diagrama de estados luce así:



- a y b tienen el mismo valor, pero no se refieren al mismo objeto.

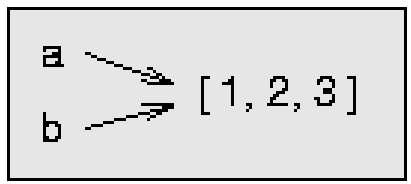
Listas XVIII

Aliasing

- Dado que las variables se refieren a objetos, si asignamos una variable a otra, ambas se refieren al mismo objeto:

```
a = [1, 2, 3]
b = a
```

- En este caso el diagrama de estados luce así:



- Dado que la misma lista tiene dos nombres, a y b, decimos que la lista está **aliased**. Cambios hechos en un **alias** afecta al otro.

```
b[0] = 5
print(a)
[5, 2, 3]
```

Listas XIX

Clonando listas

- A veces queremos modificar una lista y mantener una copia de la lista original, necesitamos hacer una copia de la lista, no solo de su referencia. Este proceso es llamado **clonación** para evitar la ambigüedad de la palabra “copia”.
- La forma más fácil de clonar una lista es usando el operador slice.

```
a = [1, 2, 3]
b = a[:]   # b = a.copy()
print(b)
[1, 2, 3]
```

Listas XX

- En este caso somos libres de hacer cambios a **b** sin preocuparnos por **a**:

```
b[0] = 5
print(a)
[1, 2, 3]
```

Listas XXI

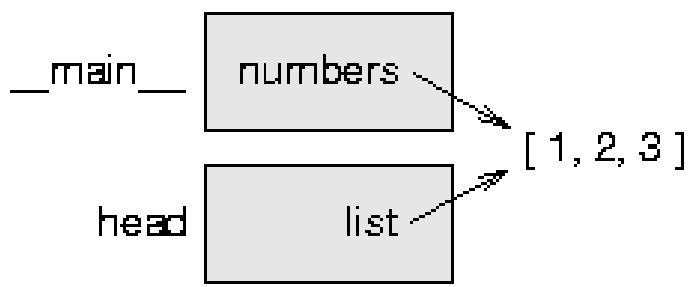
Listas como parámetros

- Cuando se pasa una lista como argumento, se está pasando una referencia a la lista, no una copia de la lista.

```
def head(list):
  return list[0]

numbers = [1, 2, 3]
head(numbers)
1
```

- El parámetro `list` y la variable `numbers` son *aliases* del mismo objeto. El diagrama de estados luce así:



Listas XXII

- Si una función modifica una lista que es parámetro, el cambio se refleja en el objeto y por tanto en sus referencias:

```
def deleteHead(list):
  del list[0]

numbers = [1, 2, 3]
deleteHead(numbers)
print(numbers)
[2, 3]
```

Listas XXIII

- Si una función devuelve una lista, devuelve una referencia a dicha lista. Por ejemplo:

```
def tail(list):
  return list[1:]

numbers = [1, 2, 3]
rest = tail(numbers)
print(rest)
[2, 3]
```

- Dado que el valor devuelto fue creado con el operador slice, es una nueva lista. La creación de `rest` y cualquier cambio subsecuente no afectan a `numbers`.

Listas XXIV

Listas anidadas

- Una lista anidada es una lista que aparece como elemento de otra lista.
- Para extraer un elemento de la lista anidada, podemos hacerlo de dos maneras:

```
list = ["hello", 2.0, 5, [10, 20]]
elt = list[3]
elt[0]
10
list[3][1]
20
```

Listas XXV

Matrices

- Podemos usar listas anidadas para representar matrices.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

- La matriz anterior puede ser representada como:

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

- Para extraer una fila de la matriz:

```
matrix[1]
[4, 5, 6]
```

- Para extraer un elemento de la matriz:

```
matrix[1][1]
5
```

Listas XXVI

Listas y Strings

- Dos de las funciones más útiles en el **módulo string** implican listas de strings.
- La función `split` separa un string en una lista de palabras.
- Por defecto, cualquier número de espacios en blanco se considera el límite de una palabra.

```
song = "The rain in Spain..."
song.split()
['The', 'rain', 'in', 'Spain...']
```

Listas XXVII

- El argumento opcional **`delimiter`** puede utilizarse para especificar qué caracter utilizar para definir los límites. Ejemplo:

```
song.split('ai')
['The r', 'n in Sp', 'n...']
```

- Note que el **delimitador** no aparece en la lista.

Listas XXVIII

- La función **`join`** es la inversa de `split`. Toma una lista de string y concatena los elementos con un espacio entre cada par:

```
lista = ['The', 'rain', 'in', 'Spain...']
' '.join(lista)
'The rain in Spain...'
```

- De igual manera que `split`, la función `join` acepta un parámetro opcional como delimitador que es insertado entre los elementos:

```
'_'.join(lista)
'The_rain_in_Spain...'
```

Más sobre listas I

- Las listas tienen algunos métodos más. Aquí hay un listado de los métodos de los objetos lista:
  - `list.append(x)`: agrega un elemento al final de la lista, es equivalente a: `a[len(a):]=[x]`
  - `list.extend(L)`: amplía una lista agregando todos los elementos en la lista `L`, es equivalente a: `a[len(a):]=L`
  - `list.insert(i,x)`: inserta un elemento en una posición dada. El primer argumento es el índice del elemento delante del cual se realizará la inserción. Entonces, `a.insert(0, x)` inserta un elemento al inicio de la lista, y `a.insert(len(a), x)` equivale a `a.append(x)`.
  - `list.remove(x)`: Retira el primer elemento de la lista cuyo valor es x. Devuelve un error si no existe el elemento.

Más sobre listas II

- `list.pop([i])`: Retira un elemento de la lista de la dada, y lo devuelve. Si no se especifica ningún índice, **`a.pop()`** elimina y devuelve el último elemento de la lista. Los corchetes alrededor de la **`i`** indican que el parámetro es opcional, no es que debamos escribir entre corchetes la posición. Verá esta notación con frecuencia en la Biblioteca de referencias de Python.
- `list.index(x)`: Devuelve el índice del primer elemento cuyo valor es x. Devuelve un error si no existe el elemento.
- `list.count(x)`: Devuelve el número de veces que x aparece en la lista.
- `list.sort()`: Ordena los elementos de la lista en el lugar (in place).
- `list.reverse()`: Invierte los elementos de la lista (in place).

Más sobre listas III

- Ejemplos:

```
a = [66.25, 33, 33, 1, 1234.5]
print(a.count(33), a.count(66.25), a.count('x'))
2 1 0
a.insert(2, -1)
a.append(33)
a
[66.25, 33, -1, 33, 1, 1234.5, 33]
a.index(33)
1
a.remove(33)
a
[66.25, -1, 33, 1, 1234.5, 33]
a.reverse()
a
[33, 1234.5, 1, 33, -1, 66.25]
a.sort()
a
[-1, 1, 66.25, 33, 33, 1234.5]
```

Comprensión de listas (list comprehension) I

- Las comprensiones de listas ofrecen una manera concisa de crear listas. Sus usos comunes son para hacer nuevas listas donde cada elemento es el resultado de algunas operaciones aplicadas a cada miembro de otra secuencia o iterable, o para crear una subsecuencia de esos elementos para satisfacer una condición determinada. Por ejemplo, asumamos que queremos crear una lista de cuadrados:

```
cuadrados = []
for x in range(10):
...     cuadrados.append(x**2)
...
cuadrados
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

- Un equivalente es:

```
cuadrados = [x ** 2 for x in range(10)]
```

Comprensión de listas (list comprehension) II

- Una lista de comprensión consiste de corchetes rodeando una expresión seguida de la declaración **for** y luego cero o más declaraciones **for** o **if**. El resultado será una nueva lista que resulta de evaluar la expresión en el contexto de los **for** o **if** que le siguen. Por ejemplo, esta lista de comprensión combina los elementos de dos listas si no son iguales:

```
[(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

- Más ejemplos:

```
vec = [2, 4, 6]
[3*x for x in vec]
[6, 12, 18]

[[x, x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]
```

Comprensión de listas (list comprehension) III

```
#Usando el if podemos filtrar

[3*x for x in vec if x > 3]
[12, 18]
[3*x for x in vec if x < 2]
[]

freshfruit = ['  apple', ' berry ', 'passion fruit ']
[f.strip() for f in freshfruit]
['apple', 'berry', 'passion fruit']
```

Comprensión de listas (list comprehension) IV

Comprensión de listas anidadas

- Si tienen el estómago para ello, las listas por comprensión se pueden anidar. Son una herramienta poderosa, pero -como todas las herramientas poderosas- necesitan ser utilizadas con cuidado.

```
mat = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print([[row[i] for row in mat] for i in [0, 1, 2]])
[[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

- Una versión más *verbosa* es:

```
for i in [0, 1, 2]:
    for row in mat:
        print(row[i], end = ' ')
    print()
```

Recursos

- Operaciones comunes con Strings: Investigar
  - string methods
  - Python regular expressions.