Introducción a las ciencias de la computación
y programación en Python

Manejo de strings, *guess and check*, métodos de aproximación

Rodrigo Chang

Banco de Guatemala

Rodrigo Chang <rrcp@banguat.gob.gt>

Abstract

"Everybody should learn to program a computer, because it teaches you how to think"
— Steve Jobs

Veremos un poco más acerca de la manipulación de cadenas de texto.
Hablaremos de los algoritmos *guess and check*.
Conoceremos algunas aplicaciones de este tipo de algoritmos.

Cadenas de texto

- Tipo de datos no escalar en Python.
- Algunos operadores curiosos definidos: `>`, `<`, `==`.
- Podemos utilizar la función `len()` para obtener el largo de la cadena.

```
s = "abcde"
len(s) # devuelve 3
```

Cadenas de texto

- Los objetos de tipo `str` poseen estructura. Esto nos permite **indexar** los caracteres que la conforman.

```
s[0]  # -> devuelve "a"
s[1]  # -> devuelve "b"
s[2]  # -> devuelve "c"
s[3]  # -> fuera de límites, error
s[-1] # -> devuelve "c"
s[-2] # -> devuelve "b"
s[-3] # -> devuelve "a"
```

Slicing

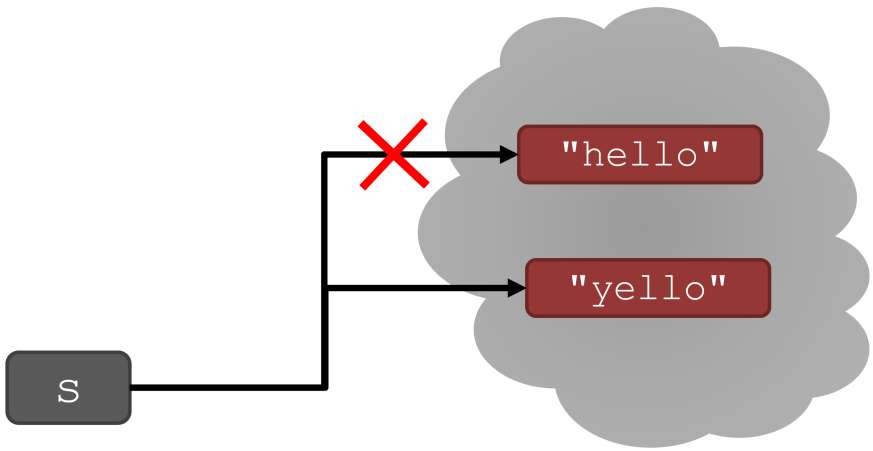
- Las cadenas se pueden "partir" utilizando `[start:stop:step]`.
- Si `[start:stop]`, `step = 1` por defecto.
- Se pueden omitir números y dejar `:`

```
s = "abcdefgh"
s[3:6]   # -> devuelve "def", s[3:6:1]
s[3:6:2] # -> devuelve "df"
s[::]    # -> devuelve "abcdefgh", s[0:len(s):1]
s[::-1]  # -> devuelve "hgfedbca", s[-1:-(len(s)+1):-1]
s[4:1:-2]# -> devuelve "ec"
```

Inmutabilidad

- Las cadenas de texto son “inmutables” = no pueden ser modificadas.

```
s = "hello"
s[0] = 'y'                # error
s = 'y' + s[1:len(s)] # redefinir
```



Recapitulación de `for`

- Previamente, aprendimos que `for` trabaja en conjunto con la función `range`.

```
for var in range(4):
    <expresiones>
    ...

for var in range(4, 10):
    <expresiones>
    ...
```

- Los ciclos `for` pueden iterar sobre cualquier conjunto de valores, ¡no solamente números!

Cadenas de texto y ciclos

- Los bloques de abajo son **equivalentes**.

```
s = "abcdefgh"

for index in range(len(s)):
    if s[index] == 'i' or s[index] == 'u':
        print("There is an i or u")

for char in s:
    if char == 'i' or char == 'u':
        print("There is an i or u")
```

El último es más *Pythónico*.

Ejemplo: porristas robot

```
an_letters = "aefhilmnorsxAEFHILMNORSX"
word = input("I will cheer for you! Enter a word: ")
times = int(input("Enthusiasm level (1-10): "))

i = 0
while i < len(word):
    char = word[i]
    if char in an_letters:
        print("Give me an " + char + "! " + char)
    else:
        print("Give me a  " + char + "! " + char)
    i += 1

print("What does that spell?")
for i in range(times):
    print(word, "!!!")
```

Ejemplo: porristas robot

```
an_letters = "aefhilmnorsxAEFHILMNORSX"
word = input("I will cheer for you! Enter a word: ")
times = int(input("Enthusiasm level (1-10): "))

for char in word:
   if char in an_letters:
       print("Give me an " + char + "! " + char)
   else:
       print("Give me a  " + char + "! " + char)


print("What does that spell?")
for i in range(times):
   print(word, "!!!")
```

Guess and check

- A este tipo de algoritmos se les conoce también como de **enumeración exhaustiva**.
    1. Dado un problema.
    2. Podemos proponer/adivinar una solución.
    3. Podemos verificar que la solución es correcta.
    4. Repetimos 1,2 y 3 hasta encontrar la solución.

Cubos perfectos

- Veamos un ejemplo de **enumeración exhaustiva**.

```
#cube = 27
##cube = 8120601

for guess in range(cube+1):
    if guess**3 == cube:
        print("Cube root of", cube, "is", guess)
```

- ¿Cómo podemos salir si encontramos la solución?
- ¿Qué pasa si `cube` no tiene cubo perfecto o es negativo?

Cubos perfectos

- Veamos una posible solución

```
cube = 27
#cube = 8120601
for guess in range(abs(cube)+1):
    # passed all potential cube roots
    if guess**3 >= abs(cube):
        # no need to keep searching
        break
if guess**3 != abs(cube):
    print(cube, 'is not a perfect cube')
else:
    if cube < 0:
        guess = -guess
    print('Cube root of ' + str(cube) + ' is ' + str(guess
    ))
```

Soluciones aproximadas

- En ocasiones, basta con una solución lo **suficientemente buena**.
  1. Dado un problema.
  2. Emmpezamos con una solución y aumentamos por un pequeño valor.
  3. Si `|guess**3 - cube| >= epsilon` , seguimos probando.
- Si aumentamos $\epsilon \Rightarrow$ menor precisión.
- Si disminuimos $\epsilon \Rightarrow$ mayor lentitud.

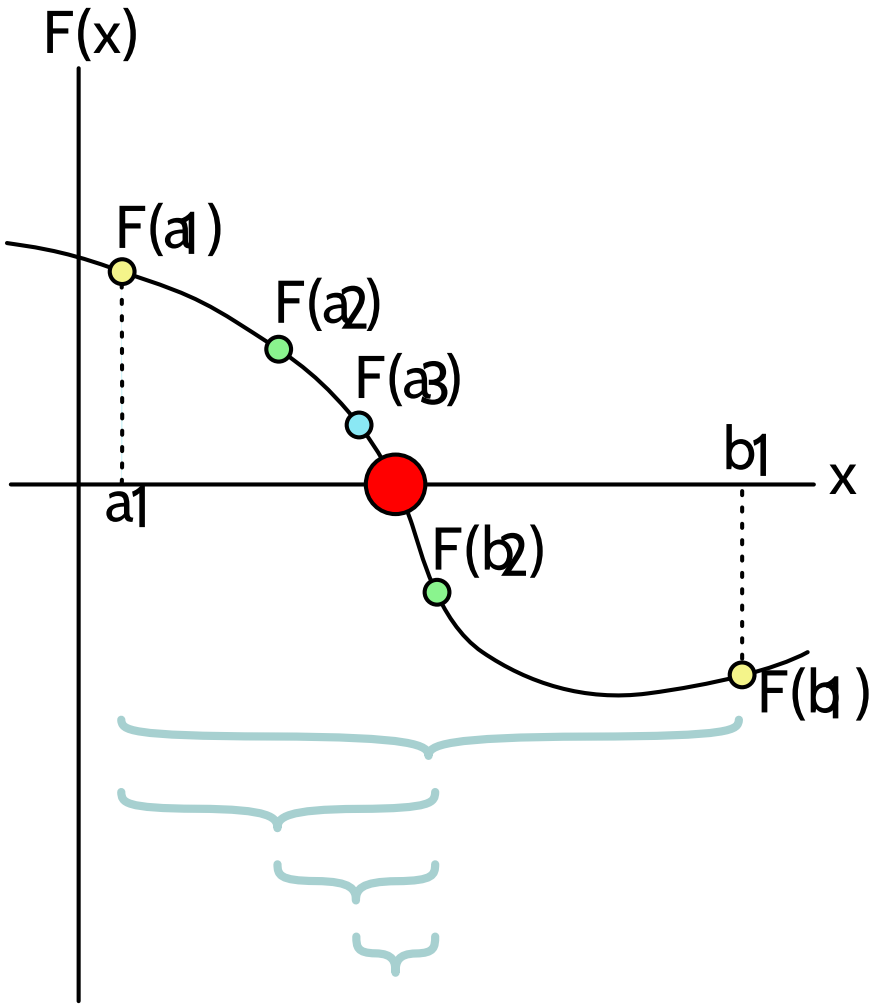
Raíz cúbica aproximada

- Veamos el siguiente ejemplo.

```
cube = 27

epsilon = 0.1
guess = 0.0
increment = 0.01
num_guesses = 0
# look for close enough answer and make sure
# didn't accidentally skip the close enough bound
while abs(guess**3 - cube) >= epsilon and guess <= cube:
    guess += increment
    num_guesses += 1
print('num_guesses =', num_guesses)
if abs(guess**3 - cube) >= epsilon:
    print('Failed on cube root of', cube)
else:
    print(guess, 'is close to the cube root of', cube)
```

Búsqueda por bisección



- En cada iteración, seleccionamos una mitad del intervalo.
- Solución candidata en el intervalo.

Raíces cúbicas

```
cube = 27

epsilon = 0.01
num_guesses, low = 0, 0
high = cube
guess = (high + low)/2.0

while abs(guess**3 - cube) >= epsilon:
    if guess**3 < cube:
        # look only in upper half search space
        low = guess
    else:
        # look only in lower half search space
        high = guess

    guess = (high + low)/2.0
    num_guesses += 1
print('num_guesses: ', num_guesses, 'raiz: ', guess)
```

Acerca de la convergencia

- La solución candidata converge en un orden de $\log_2 N$ pasos.
- La búsqueda de bisección funciona cuando la función varía monótonamente de acuerdo a la entrada.
- El código anterior funciona solamente para cubos positivos mayores a 1. ¿Por qué?

¿Cómo podemos modificar el código para que funcione con cubos negativos y positivos < 1?