Clase 3. Manipulación de Cadenas y Ejemplos de Algoritmos.

MIT 6.001. Introduction to Computer Science and Programming in Python.

Resumen

En esta clase profundizamos en distintas técnicas para manipular cadenas de caracteres. Luego, aplicaremos lo aprendido hasta ahora en el curso para revisar dos algoritmos de búsqueda.

1. Trabajando con datos de tipo cadena.

La clase anterior comenzamos estudiando sobre los datos de tipo cadena o `str` (de *string*). Ahí vimos que:

- Son objetos atómicos que consisten de una secuencia de caracteres (letras, espacios, etc) que son declarados mediante comillas (únicas o dobles).
- Pueden ser unidos o **concatenados** usando el operador `+` o la función `print()`.
- Es posible usarlos para crear programas interactivos a partir de la función `input()`.
- La única operación aritmética que podemos realizar entre éstos y un objeto de tipo numérico (`int` o `float`) es la multiplicación (e.g, `3*"a"` es igual a `'aaa'`).

También estudiamos que es posible usar operadores de comparación (`>`, `>=`, `<`, `<=`, `==` y `!=`) **solo entre cadenas** y no entre otros tipos de datos.

Ahora sigamos viendo otras maneras de trabajar con los datos de tipo `str`.

1.1 Cantidad de caracteres de una cadena.

Comúnmente, al trabajar con cadenas nos interesará conocer la **cantidad de caracteres** que la componen. En Python lo podemos obtener mediante la función `len()`.

Como vemos en el siguiente ejemplo, la función `len()` cuenta **todo tipo de caracter** de una cadena. Esto incluye letras, espacios, números y caracteres especiales (e.g, ` o $).

```
>>> x = "abc 45`$"
>>> len(x)
8
```

El valor entregado por `len()` es de tipo `int`.

1.2 Obteniendo un caracter de una cadena.

Al crear una cadena de caracteres, a cada uno de estos se le asigna un número entero llamado **índice** (*index*) que indica su **posición** adentro del objeto.

En Python, el **primer caracter** de una cadena es indicado con el **número cero**. Esto implica que el último podemos definirlo como el valor de `len()` menos `1`.

La indexación de una cadena nos da la habilidad de **recuperar** uno o más caracteres. En Python lo realizamos poniendo paréntesis `[]` adelante de ella y, al interior de éstos, escribimos el número de su posición.

```
>>> # Primer elemento de la variable `x`.
>>> x[0]
'a'
>>> # Tercer elemento de la variable `x`.
>>> x[2]
'c'
>>> # Último elemento de la variable `x`.
>>> x[7]
'$'
>>> x[len(x) - 1]
'$'
```

Si usamos un **valor mayor al índice del último caracter** de una cadena, obtenemos un error.

```
>>> x[8]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Como vemos, nos señala que el valor del índice ingresado está fuera del rango que compone a la cadena.

En Python también es posible recuperar caracteres de una cadena usando **índices negativos**. Cada valor indica la posición **en sentido contrario**, donde `-1` corresponde al último y el primero podemos expresarlo como `-len(x)`, donde `x` es una variable de tipo cadena.

```
>>> # Primer elemento de `x`.
>>> x[-8]
'a'
>>> x[-len(x)]
'a'
>>> # Tercer elemento de `x`.
>>> x[-6]
'c'
>>> # Último elemento de `x`.
>>> x[-1]
'$'
```

1.3 Recuperando un subconjunto de caracteres de una cadena.

Usando paréntesis `[ ]` también es posible obtener un **subconjunto de caracteres** de una cadena, también conocido como *slice*. Para ello, debemos seguir la siguiente estructura:

```
x[inicio:final:pasos]
```

donde:

- `x`: Una variable cualquiera de tipo cadena.
- `inicio`: Índice del primer caracter del subconjunto. Por defecto es igual a `0`.
- `final`: Índice `- 1` del último caracter del subconjunto. Por defecto es igual a `len(x)`.
- `pasos`: Cantidad de caracteres que se deben avanzar desde `inicio` para llegar a `final`. Por defecto, es igual a `1`.

Definamos a `x` como:

```
x = "abcd5678?^"
```

Recuperemos el subconjunto `'cd567'` de `x`.

```
>>> x[2:7]
'cd567'
```

Ahora obtengamos la subcadena `'c57'` de `x`. Veamos que es similar al anterior, pero debemos avanzar de dos en dos caracteres posteriores a `inicio = x[2]` para llegar a `final = x[7-1]`. Por lo tanto, vamos a definir que `pasos` sea igual a `2`.

```
>>> x[2:7:2]
'c57'
```

Para obtener la cadena completa de `x`, simplemente escribimos los `:` adentro de `[]`, pero sin señalar los índices. En dicho caso tomará los valores definidos por defecto.

```
>>> x[::]
'abcd5678?^'
```

Podemos revertir el orden de los caracteres de `x` de la siguiente manera:

```
>>> x[::-1]
'^?8765dcba'
```

Donde `pasos` igual a `-1` indica que avanzamos por cada (`1`) caracter en sentido contrario, desde `final = x[len(x)]` hasta `inicio = x[0]`. El código anterior es lo mismo que los dos siguientes:

```
>>> x[-1:-len(x)-1:-1]
'^?8765dcba'
>>> x[len(x)-1:-len(x)-1:-1]
'^?8765dcba'
```

1.4 Las cadenas no se pueden modificar.

Algo a tener en cuenta al trabajar con cadenas, es que no son modificables.

Por ejemplo, digamos que en nuestra cadena `x = "abcd5678?^"` queremos cambiar el caracter `"a"` por `"u"`. Podríamos intentar asignar a este último usando el índice del primero: `x[0] = "u"`, pero obtendremos el siguiente error:

```
>>> x[0] = "u"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Como vemos, en Python no se puede usar el operador de asignación (`=`) con objetos de tipo `str` de la manera en que lo hicimos. Por lo tanto, para lograr nuestro objetivo tendremos que

definirlo en otra variable o en la misma, como vemos a continuación.

```
>>> x = "u" + x[1:len(x)]
>>> x
'ubcd5678?^'
>>> x[0]
'u'
```

1.5 Búcles con objetos de tipo cadena.

Como cada cadena está compuesta de una secuencia de caracteres ordenados por un índice, podemos usarla para hacer iteraciones mediante búcles.

En general, para hacer iteraciones usando objetos de tipo cadena podemos tomar dos estrategias:

1. Iterar usando los índices de los caracteres.
2. Iterar usando cada caracter.

La primera estrategia consiste en que la variable de iteración (en este caso será `i`) tome los valores de una **secuencia numérica** generada por los **índices de la cadena**.

```
# Primera estrategia.
x = "abcd5678?^"

for i in range(len(x)):
  print(x[i])
```

En la segunda estrategia, la variable de iteración **toma el valor de cada caracter** de la cadena.

```
# Segunda estrategia.
x = "abcd5678?^"

for i in x:
  print(i)
```

En ambos códigos obtenemos el mismo resultado, pero se diferencian en el valor que toma la variable de iteración `i`.

2. Algoritmos de búsqueda.

A partir del conocimiento que manejamos hasta ahora, revisaremos dos algoritmos de búsqueda: el de enumeración exhaustiva y el de bisección. Los aplicaremos para calcular el valor de la raíz cúbica de un número. También añadiremos el caso de obtener su valor aproximado.

2.1 Enumeración exhaustiva.

El objetivo del algoritmo de **enumeración exhaustiva** es resolver un problema dando una solución posible y verificando si es la correcta. En caso de no serlo, usamos otro valor y volvemos comprobar. La idea es repetir este proceso hasta agotar todas las opciones.

Llevemos el concepto de la enumeración exhaustiva a código en Python para calcular la raíz cúbica perfecta de un número `x`.

```
x = int(input("Ingresa un número entero: "))
soluciones = range(0, abs(x) + 1)

for solucion in soluciones:
  if solucion**3 == abs(x):
    break

if solucion**3 != abs(x):
  print("El número", x, "no tiene una raíz cúbica perfecta.")
else:
  # La raíz cúbica de un número negativo es un número negativo.
  if x < 0:
    solucion = -solucion

  print("La raíz cúbica perfecta de", x, "es:", solucion)
```

En el código de arriba, la enumeración exhaustiva se realiza en las líneas

```
soluciones = range(0, abs(x) + 1)

for solucion in soluciones:
  if solucion**3 == abs(x):
    break
```

Las posibles `soluciones` se definen como una lista ordenada de enteros desde `0` hasta `x` y

se verifica en cada `solucion` si su cubo es igual a `x`. De serlo, significa que es la raíz cúbica perfecta de `x` y se detiene la búsqueda, por lo que la variable `solucion` queda asignada al último ítem de `soluciones` antes del `break`.

2.2 Solución aproximada.

La gran limitante del programa que vimos en la sección 2.1, es que solo permite calcular raices cúbicas perfectas. Ampliar su alcance implicaría buscar números irracionales. Como es imposible obtener valores exactos de estos últimos, optaremos por **aproximarnos** a ellos.

Definamos el término de error

$$|\texttt{solucion}^3 - \texttt{x}|$$

donde `x` es el número que buscamos calcular su raíz cúbica. Por lo tanto, obtendremos una buena aproximación mientras el cubo de la solución sea lo más cercano a `x`.

Como buscamos una aproximación a la raíz cúbica de un número y no su valor exacto, debemos determinar el nivel de precisión de la primera. Para ello, usemos un valor $\epsilon > 0$ donde:

$$|\texttt{solucion}^3 - \texttt{x}| < \epsilon$$

Es decir, hemos definido que "solucion" será una buena aproximación a `x` si la distancia de su cubo con respecto a `x` es menor a un $\epsilon > 0$ elegido arbitrariamente.

A continuación tenemos el código en Python de la configuración para calcular la raíz cúbica aproximada de `x`.

```
x = int(input("Ingrese un número: "))
epsilon = float(input("Nivel de precisión: "))

if epsilon <= 0:
  exit("\nError: Nivel de precisión debe ser mayor a cero.")

incremento = epsilon**2
num_intentos = 0
solucion = 0.0

while abs(solucion**3 - x) >= epsilon and solucion <= x:
  # Es lo mismo que `solucion = solucion + incremento`
  solucion += incremento
```

```
  num_intentos += 1

print("\nCantidad de intentos: " + str(num_intentos))

if abs(solucion**3 - x) >= epsilon:
  print("No se pudo encontrar la raíz cúbica de", x)
else:
  print("La raíz cúbica de", x, "se aproxima a", solucion)
```

En las siguientes líneas del programa de arriba se puede observar que aún estamos usando el método de enumeración exhaustiva. En este caso, cada `solucion` posible es un valor que va aumentando linealmente desde `0.0` en ϵ^2 veces, almacenada en la variable `incremento`.

```
incremento = epsilon**2
solucion = 0.0

while abs(solucion**3 - x) >= epsilon and solucion <= x:
  solucion += incremento
```

Observemos también que el búcle `while` se detendrá no solo cuando $|\texttt{solucion}^3 - \texttt{x}| < \epsilon$, sino que también cuando la solución sea mayor a `x`. Si no añadimos esa última restricción, es posible que entremos en un búcle infinito porque `solucion` seguirá aumentando y nunca se cumplirá su primera condición de termino.

Cuando escribimos un programa que calcula una aproximación, siempre tendremos que lidiar con el dilema entre la rapidez de la ejecución del código y la precisión del valor de salida. En el caso de la raíz cúbica, a medida que $\epsilon \to 0$, obtendremos un valor cada vez más similar al exacto, pero la cantidad de pasos para llegar a éste será mayor.

2.3 Bisección.

El método de bisección, en vez de hacer una búsqueda lineal como ocurre en la enumeración exhaustiva, lo hace dentro de un intervalo de valores que se va acortando a medida que se acerca al valor de salida.

Usemos el método de bisección para calcular una aproximación de la raíz cúbica de `x`. Buscaremos a la `solucion` como el punto medio de los intervalos que se vayan formando.

$$\texttt{solucion} = \frac{\texttt{cota\_sup} + \texttt{cota\_inf}}{2.0}$$

Las variables `cota_sup` y `cota_inf` son las cotas superior e inferior del intervalo. Definiremos que `cota_inf = 0.0` y `cota_sup = max(1.0, x)`.

Como se observa a continuación, el código para calcular la raíz cúbica de `x` usando el método de bisección es similar al usado con el de enumeración exhaustiva.

```
x = int(input("Ingresa un número: "))

if x < 0:
  exit("Número ingresado debe ser mayor a cero.")

epsilon = float(input("Nivel de precisión: "))

if epsilon <= 0:
  exit("\nError: Nivel de precisión debe ser mayor a cero.")

cota_sup = max(1.0, x)
cota_inf = 0.0

solucion = (cota_sup + cota_inf)/2.0
num_intentos = 0

while abs(solucion**3 - x) >= epsilon:
  num_intentos += 1

  if solucion**3 < x:
    cota_inf = solucion
  else:
    cota_sup = solucion

  solucion = (cota_sup + cota_inf)/2.0

print("\nCantidad de intentos:", num_intentos)
print("La raíz cúbica de", x, "se aproxima a", solucion, "\n")
```

Si aumentamos la precisión de la aproximación (i.e, si $\epsilon \to 0$), lo hará también la cantidad de intentos para llegar a su mejor valor de salida, pero serán mucho menos que en el método de enumeración exhaustiva.