Bucles y aproximaciones

Computación 2021

FaMAF

26 mar. 2021

Contenidos

Repetición

Códigos de la clase en:
https://github.com/mlares/compuprof2021

Iteración en Python

Repetición

La candente mañana de febrero en que Beatriz Viterbo murió, dezpuéz de una imperioza agonía que no ze rebajó un zolo inztante ni al zentimentalizmo ni al miedo, noté que laz carteleraz de fierro de la Plaza Conztitución habían renovado no zé qué avizo de cigarrilloz rubioz; el hecho me dolió, puez comprendí que el incezante y vazto univerzo ya ze apartaba de ella y que eze cambio era el primero de una zerie infinita. Cambiará el univerzo pero yo no, penzé con melancólica vanidad; alguna vez, lo zé, mi vana devoción la había exazperado; muerta yo podía conzagrarme a zu memoria, zin ezperanza, pero también zin humillación. Conzideré que el treinta de abril era zu cumpleañoz; vizitar eze día la caza de la calle Garay para zaludar a zu padre y a Carloz Argentino Daneri, zu primo hermano, era un acto cortéz, irreprochable, tal vez ineludible. De nuevo aguardaría en el crepúzculo de la abarrotada zalita, de nuevo eztudiaría laz circunztanciaz de zuz muchoz retratoz

Fragmento de "El Aleph", de J. L. Borges

Repetición

La repetición de tareas iguales o similares, muchas veces, es algo que las personas hacen bastante mal pero las computadoras hacen bastante bien. Por eso uno de los usos más comunes de las computadoras es el de automatizar tareas repetitivas.

La **iteración**, o repetición de tareas parecidas o iguales, es una de las aplicaciones más usadas en programación. Python cuenta con varias herramientas para llevarlo a cabo.

Las estructuras de control que permiten ejecutar muchas iteraciones se llaman bucles o *loops*.

Repetición en python

En python hay varias formas de hacer bucles, lo más importante a tener en cuenta es si conozco de antemano la cantidad de iteraciones.

▷ e.g.: Una cuenta regresiva desde 10 a 0, ya sé que van a ser 10 iteraciones.

```
n = 5
while n > 0:
    print(n)
    n = n - 1
print('Boom!')
```

▷ e.g.: Si sorteo números enteros aleatorios entre 1 y 100 hasta que salga 42

```
n = 0
while n != 42:
    n = random.sample(range(100), 1)[0]
print('salió el 42!')
```

Contadores y acumuladores

Vemos que muchas veces necesitamos "llevar la cuenta" de la cantidad de iteraciones, o ir cambiando una variable para usarla en cada iteración.

```
i = 0
i = i + 1
```

Otras veces es necesario ir sumando, por ejemplo, para calcular $\sum_{i=1}^{i=N} i$

```
s = 0
i = 0
while i<=n:
    s = s + i
print(s)
```

Bucle infinito

Los bucles pueden ser definidos o indefinidos, y se pueden hacer con diferentes estructuras.
El denominado "bucle infinito" usa una condición para "romperse", mediante el comando `break`.

▷ e.g.: (cuidado que esto es un ejemplo, pero no se aconseja usarlo así!)

```
n = 10
while True:
   print(n, end=' ')
   n = n - 1
   if n==0:
      break
print('Done!')
```

También es posible saltar el resto de una iteración con el comando "continue".

Bucle definido

Un función muy interesante es el de los bucles definidos, que en lugar de verificar una condición en cada iteración, recorren un conjunto.

▷ e.g.: Bucle sobre listas:

```
alumnos = ['Juan', 'José', 'Pedro']
for alumno in alumnos:
    print(f'Buenos días {alumno}!')
```

▷ e.g.: Bucles sobre strings:

```
s = 'hola'
for i in s:
    print(i)
```

Bucle definido

▷ e.g.: Bucles sobre diccionarios:

```
s = {'a':0, 'b':1}
for i in s:
    print(i)
```

▷ e.g.: Bucles sobre conjuntos:

```
s = {0, 1, 2, 3}
for i in s:
    print(i)
```

Bucle definido

En general se puede usar **`for`** sobre cualquier objeto que sea iterable:

Definition

Un objeto es iterable si es capaz de devolver sus elementos uno por vez, y puede ser "iterado" con un bloque `for`.
Más precisamente, es un objeto de una clase que posee métodos para devolver elementos indexados, el siguiente y la cantidad.

Los iterables se aceptan como argumentos de funciones *built-in*:

`sum, sorted, any, all, max, min.`

También los bucles definidos permiten llevar la cuenta del número de iteraciones con **`enumerate`**, o recorrer dos iterables (con el mismo `len`) simultáneamente, usando la función **`zip`**.

Listas por comprensión

Las listas son muy útiles pero definir listas largas por enumeración puede ser muy trabajoso (▷ e.g.: Hacer una lista con todos los enteros hasta 10000).

La solución que brinda python para esto es definir **listas por comprensión**, que consisten en usar expresiones **`for`** o **`if-else`** en lugar de escribir explícitamente todos los elementos.

▷ e.g.:

```
lista1 = [i for i in range(10000)]
lista2 = [i for i in range(10000) if i%2==0]
```

- Ej.: Crear listas que contengan los elementos de los siguientes conjuntos:
  - $S = \{x^2 : x \in \{0...9\}\}$
  - $V = \{1, 2, 4, 8, ..., 2^{12}\}$
  - $M = \{x | x \in S \wedge x \; es \; par\}$

Métodos de las listas

Las listas tienen un conjunto de métodos que son muy útiles y que también se pueden usar para su construcción:
`append`, `clear`, `copy`, `count`, `extend`, `index`, `insert`, `pop`, `remove`, `reverse`, `sort`

▷ e.g.:

```
lista = list()
for i in range(10):
    lista.append(i)

lista.reverse()
```

Usos de los bucles

Los bucles se usan para muchas estructuras repetitivas, tales como:

- contar
- acumular
- buscar
- **métodos iterativos**
- **aproximaciones**

Vemos que son muy útiles para calcular polinomios o sumas de muchos términos.

Pero... al hacer esas cosas no debemos olvidar los problemas de la representación de punto flotante de los números!

Aproximaciones

Evaluar un polinomio

Supongamos que tenemos un polinomio

$$P(x) = 2x^4 + 3x^3 - 3x^2 + 5x - 1$$

y lo queremos evaluar en un valor de x.

¿Cuál es la mejor forma de hacerlo?

Aquí "mejor" quiere decir más eficiente, usando la menor cantidad de operaciones posibles, y evitando situaciones de "overflow" o "underflow".

¿Cuántas operaciones lleva esta ecuación?

$$P(x) = 2 \cdot x \cdot x \cdot x \cdot x \; + \; 3 \cdot x \cdot x \cdot x \; - \; 3 \cdot x \cdot x \; + \; 5 \cdot x \; - \; 1$$

Evaluar un polinomio

$$P(x) = 2 \cdot x \cdot x \cdot x \cdot x \; + \; 3 \cdot x \cdot x \cdot x \; - \; 3 \cdot x \cdot x \; + \; 5 \cdot x \; - \; 1$$

Vemos que la operación de multiplicar x por sí mismo se repite muchas veces. Una estrategia para evitar esto podría ser la siguiente:

```
m2 = x * x
m3 = m2 * x
m4 = m3 * x
```

Ahora podemos hacer:

```
P(x)= 2 * m4 + 3 * m3 - 3 * m2 + 5 * x - 1
```

Evaluar un polinomio

Podemos todavía eliminar algunas operaciones con un truco matemático:

$$\begin{aligned} P(x) &= 2x^4 + 3x^3 - 3x^2 + 5x - 1 \\ &= -1 + 5x - 3x^2 + 3x^3 + 2x^4 \\ &= -1 + x(5 - 3x + 3x^2 + 2x^3) \\ &= -1 + x(5 + x(-3 + 3x + 2x^2)) \\ &= -1 + x(5 + x(-3 + x(3 + 2x))) \\ &= -1 + x(5 + x(-3 + x(3 + 2x))) \end{aligned}$$

¿Cuántas operaciones lleva ahora?

Este método se conoce como el método de Horner.

Método de Horner

Teorema

Sea un polinomio de grado n dado por

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

y sean $b_n = a_n$, *y* $b_k = a_k + b_{k+1} x_0$ *para* $k = 0, 1, \ldots, n-1$

Entonces, $P(x_0) = b_0$ *y además* $P(x) = (x - x_0)Q(x) + b_0$
con

$$Q(x) = b_n x^{n-1} + b_{n-1} x^{n-2} + \ldots + b_2 x + b_1$$

Otra ventaja es que se puede calcular la derivada del polinomio fácilmente:

$$P'(x) = Q(x) + (x - x_0 Q'(x)), \qquad P'(x_0) = Q(x_0)$$

Estrategia de Horner

El método para $n = 4$ sería:

$$p(x) = c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0$$

$$p(x) = \Big(\big((c_4x + c_3)x + c_2\big)x + c_1\Big)x + c_0.$$

Un pseudocódigo para la estrategia de Horner sería:

1. Tomar $p = c_n$ y $k = n - 1$,
2. hacer $p \leftarrow px + c_k$,
3. si $k = 0$ parar. Sino hacer $k \leftarrow k - 1$ e ir al paso 1.

Funciones para evaluar polinomios

Definamos dos funciones en Python para evaluar un polinomio de grado n en un punto x:

```
def evalpol(c,x):
  """ Función para evaluar un polinomio.
  Para un polinomio p(x)= c_n x**n+c_{n-1} x**{n-1}+ ...+c_0
  c=[c_0, c_1, ..., c_n], vector de dimensión n+1 """
  p = 0
  n = len(c)
  for i in range(n):
    p = p + c[i] * x**i
  return p

def horner(c,x):
  # Función que implementa el método de Horner
  n = len(c) - 1
  p = c[n]
  for i in reversed(range(n)):
    p = p * x + c[i]
  return p
```

Polinomios en la PC

Evaluar polinomios es una tarea muy importante porque permiten evaluar otras funciones por medio de aproximaciones. Si bien la suma es una de las operaciones más simples en aritmética, en una computadora es bastante difícil hacerlo BIEN.
En particular, los polinomios de Taylor se usan mucho en análisis numérico.

Repaso: series de Taylor

Theorem

Sea una función $f \in C^n[a,b]$, con derivada $f^{(n+1)}$ finita en el intervalo $[a,b]$, y sea $x_0 \in [a,b]$.

Entonces, para cada $x \in [a,b]$ existe un valor $\xi(x)$ entre x_0 y x tal que

$$f(x) = P_n(x) + R_n(x)$$

donde

$$P_n(x) = \sum_{k=0}^{n} \frac{f^{(k)}(x_0)}{k!}(x - x_0)^k$$

y

$$R_n(x) = \frac{f^{(n+1)}(\xi(x))}{(n+1)!}(x - x_0)^{n+1}$$

Polinomios de Taylor

El polinomio $P_n(x)$ es el polinomio de Taylor, y su límite para $n \to \infty$ es la serie de Taylor.
El término $R_n(x)$ es el error de truncamiento.
Tenemos que pensar ahora en dos problemas:

- El error que surge de utilizar un número finito de términos.
- El error que surge de representar los números con una aritmética finita.

El error en las restas de números similares

Para tener problemas al hacer operaciones de sumas y restas no hace falta trabajar con muchos términos, puede pasar con pocos términos cuando hacemos restas de números muy parecidos.
En general hay que evitar restar cantidades que den un valor próximo a cero.

▷ e.g.: Sabemos que podemos calcular las raíces del polinomio $ax^2 + bx + c$ mediante las fórmulas

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, \quad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}.$$

Notar que si b^2 tiene una magnitud mucho mayor a la de $4ac$ entonces tendremos mucho error al calcular x_1 si b es positivo (x_2 si b es negativo).

```
a=1; b=-62.1; c=1
x1=(-b+m.sqrt(b**2-4*a*c))/(2*a)
x2=(-b-m.sqrt(b**2-4*a*c))/(2*a)
print('x1=',x1,', x2=',x2)
print('p(x1)=',a*x1**2+b*x1+c)
print('p(x2)=',a*x2**2+b*x2+c)
```

Fórmula de Baskara

La manera de evitar este tipo de error, modificaremos la fórmula racionalizando el numerador:

Si $b > 0$,

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \frac{-b - \sqrt{b^2 - 4ac}}{-b - \sqrt{b^2 - 4ac}} = \frac{-2c}{b + \sqrt{b^2 - 4ac}},$$

y si $b < 0$,

$$x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \frac{-b + \sqrt{b^2 - 4ac}}{-b + \sqrt{b^2 - 4ac}} = \frac{-2c}{b - \sqrt{b^2 - 4ac}}.$$

Tomando esto en cuenta, podemos modificar el programa que escribimos para calcular éstas raíces y hacer una nueva versión que se comporta mejor.

Fórmula de Baskara

```
def raizCuad(a,b,c):
  D=b**2-4*a*c
  if D>0:
    if b>=0:
      x1=-2*c/(b+m.sqrt(D))
      x2=(-b-m.sqrt(D))/(2*a)
    else:
      x1=(-b-m.sqrt(D))/(2*a)
      x2=-2*c/(b-m.sqrt(D))
    return [x1,x2]
  elif D==0:
    x=-b/(2*a)
    return x
  else:
    return "no existen raíces reales"

a=1; b=62.1; c=1
xnuevo=raizCuad(a,b,c)
print(xnuevo)
print('p(x1)=',a*xnuevo[0]**2+b*xnuevo[0]+c)
print('p(x2)=',a*xnuevo[1]**2+b*xnuevo[1]+c)
```

Restando números muy parecidos

En general, se recomienda reescribir las fórmulas donde pueda existir una diferencia cercana a cero.

▷ e.g.: Para $b > 0$,

$$\log(b) - \log(b + \varepsilon) = \log\left(\frac{b}{b + \varepsilon}\right).$$

Calculando el número e

Un problema típico que surge cuando usamos series de Taylor para aproximar funciones es el de sumar a valores cada vez más grandes incrementos cada vez más chicos.
Por ejemplo, para un valor grande de n sabemos que

$$e \approx \sum_{i=0}^{n} \frac{1}{i!} = \sum_{j=0}^{n} \frac{1}{(n-j)!}.$$

Veamos que pasa cuando realizamos estas dos sumas en Python...
Solución: Realizar sumas de menor a mayor para un listado de números positivos.

notación de orden asimptótico

Vimos que aparecen muchos problemas numéricos y algunas estrategias para mitigarlos. Pero, ¿cómo sabemos si estamos cometiendo errores?

Necesitamos una métrica que nos permita tener una idea o cota del error que cometemos.

Deberemos utilizar métodos que generen sucesiones que irán aproximándose a la solución del problema. Éstos se conocen como **métodos de aproximación**.

En general, para los problemas en donde la solución no está dada por una fórmula, la estrategia es utilizar un método que va calculando aproximaciones cada vez mejores.

Aproximaciones

Uno de los aspectos a tener en cuenta en estos métodos es la **estabilidad**.

Diremos que un método es:

- estable si pequeñas variaciones en los datos iniciales producen pequeñas variaciones en los resultados,
- inestable si pequeñas variaciones en los datos iniciales producen variaciones grandes en los resultados, y
- condicionalmente estable si son estables solo para algunos datos iniciales.

En la mayoría de los casos se puede estimar el error cometido en la iteración n, digamos E_n. A medida que vayamos aproximándonos a la solución, E_n debería tender a cero.

Velocidad de convergencia

Como los procedimientos tendrán una cantidad finita de pasos, nos gustaría que esta sucesión converja lo más rápidamente posible. Para estimar la **velocidad de convergencia** usaremos la notación O y o.

Def./

Dada una sucesión $t_n \to 0^+$ y una función $F : (0, \infty) \mapsto \mathbb{R}$, denotaremos:

- $F(t_n) = O(t_n)$ si existe $c > 0$ tal que $\frac{|F(t_n)|}{t_n} < c$, para n suficientemente grande, y diremos que $F(t_n)$ converge a cero con orden (o con velocidad de convergencia) $O(t_n)$.
- $F(t_n) = o(t_n)$ si para todo $\varepsilon > 0$ vale que $\frac{|F(t_n)|}{t_n} < \varepsilon$, para n suficientemente grande, y diremos que $F(t_n)$ converge a cero con orden $o(t_n)$

Velocidad de convergencia

De manera equivalente,

- $F(t_n) = O(t_n)$ si $\limsup_{n\to\infty} \dfrac{|F(t_n)|}{t_n} < \infty$,
- $F(t_n) = o(t_n)$ si $\lim_{n\to\infty} \dfrac{|F(t_n)|}{t_n} = 0$.

Cuando analizamos la velocidad de convergencia del error E_n decimos que converge:

- linealmente, si $E_{n+1} = O(E_n)$ con $c < 1$,
- superlinealmente, si $E_{n+1} = o(E_n)$,
- cuadráticamente, si $E_{n+1} = O(E_n^2)$.

▷ *e.g.: media aritmética*

Media aritmética-geométrica: dados dos números reales positivos x e y, se generan sucesiones $\{a_n\}$ y $\{b_n\}$ tales que $a_0 = x$, $b_0 = y$,

$$a_{n+1} = \frac{a_n + b_n}{2}, \quad b_{n+1} = \sqrt{a_n b_n}.$$

Se sabe que $\{a_n\}$ es decreciente, $\{b_n\}$ es creciente, $b_n \leq \text{agm}(x, y) \leq a_n$ y ambas convergen a

$$\text{agm}(x, y) = \frac{\pi}{2} \left(\int_0^{\pi/2} \frac{d\theta}{\sqrt{x^2 \cos^2(\theta) + y^2 \text{sen}^2(\theta)}} \right)^{-1}.$$

▷ *e.g.:*

Suponga que se tienen las sucesiones

$$x_n = \frac{n+1}{n^2}, \quad y_n = \frac{n+3}{n^3},$$

para $n \geq 1$ con $x_0 = y_0 = 5$.

```
N=10
x=[5]; y=[5]
for n in range(1,N):
  x.append((n+1)/n**2)
  y.append((n+3)/n**3)
print(x)
print(y)
plt.figure()
plt.plot(x)
plt.plot(y)
plt.title("{xn}, {yn}");
```

▷ *e.g.: Velocidad de convergencia*

Si analizamos la velocidad de convergencia notaremos que

$$\frac{x_n}{1/n} = 1 + \frac{1}{n} \to 1, \quad \frac{x_n}{1/n^2} = n + 1 \to \infty,$$

por lo tanto $x_n = O(\frac{1}{n})$, y como

$$\frac{y_n}{1/n} = \frac{n+3}{n^2} \to 0, \quad \frac{y_n}{1/n^2} = 1 + \frac{3}{n} \to 1,$$

entonces $y_n = o(\frac{1}{n})$ y $y_n = O(\frac{1}{n^2})$.

▷ *e.g.: Fibonacci*

En algunos casos pasa que para algunos valores de p la sucesión converge y para otros diverge.
Considere la sucesión de Fibonacci:

$$F_0 = 1, \quad F_1 = 1, \quad F_{n+2} = F_n + F_{n+1}.$$

Para analizar la velocidad de convergencia definimos

$$x_n = \frac{F_{n+1}}{F_n}.$$

Si existiera $x = \lim_{n\to\infty} x_n$, como

$$x_n = \frac{F_{n-1} + F_n}{F_n} = \frac{F_{n-1}}{F_n} + 1 = \frac{1}{x_{n-1}} + 1,$$

tomando límite obtendríamos que $x = 1/x + 1$ y por lo tanto cumple la ecuación $x^2 - x - 1 = 0$. O sea,

$$x = \frac{1 + \sqrt{5}}{2}.$$

Qué aprendimos

- ¡el orden en el que hacemos las operaciones importa! (en computación, aunque no analíticamente)
- ¡cuidado con las sumas de muchos términos y las restas de números parecidos!
- cuando hay que calcular algo que no se puede hacer de manera exacta, reemplazamos la función por algo más simple que la aproxima, y hacemos el cálculo exacto en la aproximación simple.