MAT 305: Mathematical Computing

Repeating a task with loops

John Perry

University of Southern Mississippi

Fall 2011

Outline

Outline

Loops?

- **loop:** a sequence of statements that is repeated

big time bug: *infinite loops*

Why loops?

- like functions: avoid retyping code
  - many patterns repeated
  - same behavior, different data
- unlike functions: easily vary repetitions of code
  - easier than typing a function name 100 times
  - can repeat without knowing number of times when programming

Types of loops

- definite
  - number of repetitions known at beginning of loop
- indefinite
  - number of repetitions not known (even unknowable) at beginning of loop

Types of loops

- definite
  - number of repetitions known at beginning of loop
- indefinite
  - number of repetitions not known (even unknowable) at beginning of loop

Python uses different constructions for each

$\therefore$ *Sage uses different constructions for each*

Outline

The `for` command

`for` *each* `in` *L*`:`

  statement1

  statement2

  ...

where

- *each* is an identifier
- *L* is an "iterable collection" (tuples, lists, sets)
- if you modify *each*,
  - corresponding element of *L* does *not* change
  - on next loop, *each* takes next element of *L* anyway

What does it do?

`for` *each* `in` L`:`

  statement1

  statement2

  ...

- suppose L has n elements
- *statement1*, *statement2*, etc. are repeated (looped) n times
- on ith loop, *each* has the value of ith element of L

Trivial example

```
sage: for each in [1, 2, 3, 4]:
          print each
1
2
3
4
```

Less trivial example

```
sage:  for each in [x**2, cos(x), e**x*cos(x)]:
         print diff(each)
2*x
-sin(x)
-e^x*sin(x) + e^x*cos(x)
```

What happened?

```
L == [x**2, cos(x), e**x*cos(x)]
```

What happened?

```
L == [x**2, cos(x), e**x*cos(x)]
```

loop 1: `each = x**2`

`print diff(each)` ⇝ `2x`

What happened?

`L == [x**2, cos(x), e**x*cos(x)]`

loop 1: `each = x**2`

`print diff(each)` $\rightsquigarrow$ `2x`

loop 2: `each = cos(x)`

`print diff(each)` $\rightsquigarrow$ `-sin(x)`

What happened?

`L == [x**2, cos(x), e**x*cos(x)]`

loop 1: `each = x**2`

`print diff(each)` $\rightsquigarrow$ `2x`

loop 2: `each = cos(x)`

`print diff(each)` $\rightsquigarrow$ `-sin(x)`

loop 3: `each = e**x*cos(x)`

`print diff(each)` $\rightsquigarrow$ `-e^x*sin(x) + e^x*cos(x)`

Changing *each* ?

```
sage: L = [1,2,3,4]
sage: for each in L:
        each = each + 1
        print each
2
3
4
5
```

Changing *each* ?

```
sage: L = [1,2,3,4]
sage: for each in L:
        each = each + 1
        print each
2
3
4
5
```

Notice: loop ran 4 times (L has 4 elements) even though *each* had value 5

Changing L ?

Don't modify L **unless you know what you're doing.**

Changing L ?

Don't modify L **unless you know what you're doing.** Usually, you don't.

```
sage:  L = [1,2,3,4]
sage:  for each in L:
         L.append(each+1)
```

Changing L ?

Don't modify L **unless you know what you're doing.**
Usually, you don't.

```
sage:  L = [1,2,3,4]
sage:  for each in L:
           L.append(each+1)
```

…infinite loop!

More detailed example

Given $f(x)$, $a, b \in \mathbb{R}$, and $n \in \mathbb{N}$, estimate $\int_a^b f(x)\, dx$ using n left Riemann sums.

More detailed example

Given $f(x)$, $a, b \in \mathbb{R}$, and $n \in \mathbb{N}$, estimate $\int_a^b f(x)\,dx$ using n left Riemann sums.

- Excellent candidate for definite loop if n known from outset.
  - Riemann sum: *repeated* addition: loop!
  - If n *is not* known, can still work, but a function with a loop is better. (Details later.)
- Start with pseudocode…

Pseudocode for definite loop

for *counter* $\in L$
    loop statement 1
    loop statement 2
    ...
out-of-loop statement 1

Pseudocode for definite loop

for *counter* $\in L$
    loop statement 1
    loop statement 2
    ...
out-of-loop statement 1

Notice:

- indentation ends at end of loop
- $\in$, not `in` (mathematics, not Python)
- no colon

Building pseudocode

Ask yourself:

- What list do I use to repeat the action(s)?
- What do I have to do in each loop?
  - How do I break the task into pieces?
  - *Divide et impera!* **Divide and conquer!**

Review

How do we estimate limits using left Riemann sums?

Review

How do we estimate limits using left Riemann sums?

$$\int_a^b f(x)\,dx \approx \sum_{i=1}^{n} f(x_i)\,\Delta x$$

where

- $\Delta x = \frac{b-a}{n}$
- $x_1 = a,\ x_2 = a + \Delta x,\ x_3 = a + 2\Delta x,\ \ldots\ x_n = a + (n-1)\Delta x$
  - short: $x_i = a + (i-1)\Delta x$

Review

How do we estimate limits using left Riemann sums?

$$\int_a^b f(x)\,dx \approx \sum_{i=1}^{n} f(x_i)\,\Delta x$$

where

- $\Delta x = \frac{b-a}{n}$
- $x_1 = a, x_2 = a + \Delta x, x_3 = a + 2\Delta x, \ldots\ x_n = a + (n-1)\Delta x$
    - short: $x_i = a + (i-1)\Delta x$

So...

- $L = (1, 2, \ldots, n)$
- repeat addition of $f(x_i)\,\Delta x$
    - use computer to remember previous value and add to it
    - $sum = sum + \ldots$

Pseudocode

Let $\Delta x = \frac{b-a}{n}$	this is not given
Let $L = (1, 2, \ldots, n)$	set up L—notice no Pythonese
Let $S = 0$	S must start at 0 (no sum)
for $i \in L$	
$\quad x_i = a + (i-1)\Delta x$	determine x_i
$\quad S = S + f(x_i)\Delta x$	add to S

Pseudocode

Let $\Delta x = \frac{b-a}{n}$	this is not given
Let $L = (1, 2, \ldots, n)$	set up L—notice no Pythonese
Let $S = 0$	S must start at 0 (no sum)
for $i \in L$	
$\quad x_i = a + (i-1)\Delta x$	determine x_i
$\quad S = S + f(x_i)\Delta x$	add to S

translates to Sage as…

```
Delta_x = (b - a)/x
L = range(1,n+1)
S = 0
for i in L:
  xi = a + (i - 1)*Delta_x
  S = S + f(x=xi)*Delta_x
```

now use Pythonese

Try it!

```
sage: f = x**2; a = 0; b = 1; n = 3
sage: Delta_x = (b - a)/n
sage: L = range(1,n+1)
sage: S = 0
sage: for i in L:
          xi = a + (i - 1)*Delta_x
          S = S + f(x=xi)*Delta_x
sage: S
```

Try it!

```
sage:  f = x**2; a = 0; b = 1; n = 3
sage:  Delta_x = (b - a)/n
sage:  L = range(1,n+1)
sage:  S = 0
sage:  for i in L:
         xi = a + (i - 1)*Delta_x
         S = S + f(x=xi)*Delta_x
sage:  S
5/27
```

What happened?

```
L = [1,2,3]
```

What happened?

```
L = [1,2,3]
```

loop 1: $i = 1$

```
        xi = a + (i - 1)*Delta_x
                    ⇝ xi = 0 + 0*(1/3) = 0
        S = S + f(x=xi)*Delta_x
                    ⇝ S = 0 + f(0)*(1/3) = 0
```

What happened?

`L = [1,2,3]`

loop 1: $i = 1$

```
xi = a + (i - 1)*Delta_x
```

$\rightsquigarrow$ `xi = 0 + 0*(1/3) = 0`

```
S = S + f(x=xi)*Delta_x
```

$\rightsquigarrow$ `S = 0 + f(0)*(1/3) = 0`

loop 2: $i = 2$

```
xi = a + (i - 1)*Delta_x
```

$\rightsquigarrow$ `xi = 0 + 1*(1/3) = 1/3`

```
S = S + f(x=xi)*Delta_x
```

$\rightsquigarrow$ `S = 0 + f(1/3)*(1/3) = 1/27`

What happened?

`L = [1,2,3]`

loop 1: $i = 1$

```
xi = a + (i - 1)*Delta_x
```

$\rightsquigarrow$ `xi = 0 + 0*(1/3) = 0`

```
S = S + f(x=xi)*Delta_x
```

$\rightsquigarrow$ `S = 0 + f(0)*(1/3) = 0`

loop 2: $i = 2$

```
xi = a + (i - 1)*Delta_x
```

$\rightsquigarrow$ `xi = 0 + 1*(1/3) = 1/3`

```
S = S + f(x=xi)*Delta_x
```

$\rightsquigarrow$ `S = 0 + f(1/3)*(1/3) = 1/27`

loop 3: $i = 3$

```
xi = a + (i - 1)*Delta_x
```

$\rightsquigarrow$ `xi = 0 + 2*(1/3) = 2/3`

```
S = S + f(x=xi)*Delta_x
```

$\rightsquigarrow$ `S = 1/27 + f(2/3)*(1/3) = 5/27`

Try it with larger n!

```
sage: f = x**2; a = 0; b = 1; n = 1000
sage: Delta_x = (b - a)/n
sage: L = range(1,n+1)
sage: S = 0
sage: for i in L:
        xi = a + (i - 1)*Delta_x
        S = S + f(x=xi)*Delta_x
sage: S
```

Try it with larger n!

```
sage: f = x**2; a = 0; b = 1; n = 1000
sage: Delta_x = (b - a)/n
sage: L = range(1,n+1)
sage: S = 0
sage: for i in L:
        xi = a + (i - 1)*Delta_x
        S = S + f(x=xi)*Delta_x
sage: S
665667/2000000
```

correct answer is $\frac{1}{3}$; use `round()` to see how "close"

Typing and retyping is a pain

Make a function out of it!

algorithm *left_Riemann_sum*

Typing and retyping is a pain

Make a function out of it!

algorithm *left_Riemann_sum*

inputs

  f, a function on $[a,b] \subset \mathbb{R}$

  $n \in \mathbb{N}$

Typing and retyping is a pain

Make a function out of it!

algorithm *left_Riemann_sum*

inputs

f, a function on $[a,b] \subset \mathbb{R}$

$n \in \mathbb{N}$

outputs

left Riemann sum approximation of $\int_a^b f(x)\, dx$ w/n rectangles

Typing and retyping is a pain

Make a function out of it!

algorithm *left_Riemann_sum*

inputs

f, a function on $[a,b] \subset \mathbb{R}$

$n \in \mathbb{N}$

outputs

left Riemann sum approximation of $\int_a^b f(x)\,dx$ w/n rectangles

do

Let $\Delta x = \frac{b-a}{n}$

Let $L = (1, 2, \ldots, n)$

Let $S = 0$

for $i \in L$

$x_i = a + (i-1)\Delta x$

$S = S + f(x_i)\Delta x$

return S

don't forget to report the result!

Translate into Sage code…

…on your own. Raise your hand if you need help.

You should be able to compute:

- `left_Riemann_sum(x**2, 0, 1, 3)`
- `left_Riemann_sum(x**2, 0, 1, 1000)`

…and obtain the same answers as before.

Outline

Looping through nonexistent lists

- `for` *each* `in` L
  - L an "iterable collection"
- may not want to construct list of n elements; merely repeat n times
  - `for` *each* `in xrange(`L`)` has same effect
  - slightly faster, uses less computer memory

Building lists from lists

Python (Sage) has a handy list constructor

- Suppose L_{old} has n elements
- Let L_{new} `=` `[`$f(x)$ `for` $x \in L_{old}$`]`
    - L_{new} will be a list with n elements
    - L_{new}`[i] ==` f`(`L_{old}`[i])`

Example

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

Outline

The `while` command

`while` *condition* `:`

  statement1

  statement2

  ...

where

- statements are executed while *condition* remains true
- like definite loops, variables in *condition* can be modified
- unlike definite loops, variables in *condition* **should** be modified
- *warning:* statements will *not* be executed if *condition* is false from the get-go

Pseudocode for indefinite loop

while *condition*

  statement1

  statement2

  …

out-of-loop statement 1

Pseudocode for indefinite loop

while *condition*
    statement1
    statement2
    ...
out-of-loop statement 1

Notice:

- indentation ends at end of loop
- no colon

Silly example

```
sage: f = x**10
sage: while f != 0:
        f = diff(f)
        print f
10*x^9
90*x^8
720*x^7
5040*x^6
30240*x^5
151200*x^4
604800*x^3
1814400*x^2
3628800*x
3628800
0
```

More interesting example

Use the Method of Bisection to approximate a root of $\cos x - x$ on the interval $[0, 1]$, correct to the hundredths place.

More interesting example

Use the Method of Bisection to approximate a root of $\cos x - x$ on the interval $[0, 1]$, correct to the hundredths place.
Hunh?!?

Method of Bisection?

The Method of Bisection is based on:

Theorem (Intermediate Value Theorem)

If

- f *is a continuous function on* $[a,b]$, *and*
- $f(a) \neq f(b)$,

then

- *for any* C *between* $f(a)$ *and* $f(b)$,
- $\exists c \in (a,b)$ *such that* $f(c) = C$.

Method of Bisection?

The Method of Bisection is based on:

Theorem (Intermediate Value Theorem)

If

- f *is a continuous function on* $[a,b]$, *and*
- $f(a) \neq f(b)$,

then

- *for any* C *between* $f(a)$ *and* $f(b)$,
- $\exists c \in (a,b)$ *such that* $f(c) = C$.

Upshot: To find a root of a continuous function f, start with two x values a and b such that $f(a)$ and $f(b)$ have different signs, then bisect the interval.

Back to the example...

Given

- $f(x) = \cos x - x$
  - continuous because it is a difference of continuous functions
- $a = 0$ and $b = 1$
  - $f(a) = 1 > 0$
  - $f(b) \approx -0.4597 < 0$

Intermediate Value Theorem applies: can start Method of Bisection.

How to solve it?

Idea: Interval endpoints a and b are not close enough as long as their digits differ through the hundredths place.

How to solve it?

Idea: Interval endpoints a and b are not close enough as long as their digits differ through the hundredths place.

Application: While their digits differ through the hundredths place, halve the interval.

How to solve it?

Idea: Interval endpoints a and b are not close enough as long as their digits differ through the hundredths place.

Application: While their digits differ through the hundredths place, halve the interval.

"Halve" the interval? Pick the half containing a root!

Pseudocode

algorithm *method_of_bisection*

Pseudocode

algorithm *method_of_bisection*

inputs

f, a continuous function

$a, b \in \mathbb{R}$ such that $a \neq b$ **and** $f(a)$ and $f(b)$ have different signs

Pseudocode

algorithm *method_of_bisection*

inputs

f, a continuous function

$a, b \in \mathbb{R}$ such that $a \neq b$ **and** $f(a)$ and $f(b)$ have different signs

outputs

$c \in \mathbb{R}$ such that $f(c) \approx 0$ **and** c accurate to hundredths place

Pseudocode

algorithm *method_of_bisection*

inputs

f, a continuous function

$a, b \in \mathbb{R}$ such that $a \neq b$ **and** $f(a)$ and $f(b)$ have different signs

outputs

$c \in \mathbb{R}$ such that $f(c) \approx 0$ **and** c accurate to hundredths place

do

while the digits of a and b differ through the hundredths

Let $c = \frac{a+b}{2}$

if $f(a)$ **and** $f(c)$ have the same sign

Let $a = c$ — Interval now $\left(\frac{a+b}{2}, b\right)$

else if $f(a)$ and $f(c)$ have opposite signs

Let $b = c$ — Interval now $\left(a, \frac{a+b}{2}\right)$

else — we must have $f(c) = 0$

return c

return a, rounded to hundredths place

Try it!

```
sage:  def method_of_bisection(f,x,a,b):
         while round(a,2) != round(b,2):
```

Try it!

```
sage:  def method_of_bisection(f,x,a,b):
         while round(a,2) != round(b,2):
           c = (a + b)/2
```

Try it!

```
sage: def method_of_bisection(f,x,a,b):
        while round(a,2) != round(b,2):
          c = (a + b)/2
          if f(x=a)*f(x=c) > 0:
            a = c
          elif f(x=a)*f(x=c) < 0:
            b = c
          else:
            return c
        return round(a,2)
```

Try it!

```
sage: def method_of_bisection(f,x,a,b):
        while round(a,2) != round(b,2):
          c = (a + b)/2
          if f(x=a)*f(x=c) > 0:
            a = c
          elif f(x=a)*f(x=c) < 0:
            b = c
          else:
            return c
        return round(a,2)

sage: method_of_bisection(cos(x)-x,x,0,1)
0.74
```

Outline

Summary

Two types of loops

- definite: n repetitions known at outset
  - **for** $i \in L$
    - list L of n elements controls loop
    - don't modify L
- indefinite: number of repetitions not known at outset
  - **while** *condition*
    - Boolean *condition* controls loop