Repetition means Loops

Looping in a collection

Looping on a collection

A useful trick w/loops

Summary

MAT 305: Mathematical Computing

Repeating a task on a set (or list, or tuple, or…)

John Perry

University of Southern Mississippi

Fall 2013

Outline

1. Repetition means Loops
2. Looping in a collection
3. Looping on a collection
4. A useful trick w/loops
5. Summary

Outline

1. Repetition means Loops
2. Looping in a collection
3. Looping on a collection
4. A useful trick w/loops
5. Summary

Repetition?

We often have to repeat a computation that is

- not a mere operation, *and*
- not convenient to do by hand.

Example

- Compute the first 100 derivatives of $f(x)$.

A difficulty

We may not know *how* many computations ahead of time!

Examples

- Add the first n numbers
  - What is n?
- Determine whether all elements of the set S are prime
  - What is in S?

Solution: loops!

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

big time bug: *infinite loops*

Solution: loops!

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

big time bug: *infinite loops*

“infinite loop”?

see infinite loop

— *AmigaDOS* manual, ca. 1993

Why loops?

- avoid retyping code
    - many patterns repeated
    - same behavior, different data
- may not know number of repetitions when programming

The `for` command

`for` c `in` C

where

- c is an identifier
- C is an "iterable collection" (tuples, lists, sets)

Outline

1. Repetition means Loops
2. Looping in a collection
3. Looping on a collection
4. A useful trick w/loops
5. Summary

What does it do?

[*statement* `for` c `in` C]

- suppose C has n elements
- *statement* is repeated (looped) n times
- on ith loop, c has the value of ith element of C

Examples

Example (Trivial)

```
sage: [cos(pi*t) for t in [1, 2, 3, 4]]
[-1, 1, -1, 1]
```

Example (Not quite so trivial)

Sampling $f(x) = x^2$ with 10 points on $[2, 5]$

```
sage: f(x) = x**2
sage: delta_x = (5-2)/10
sage: [f(2 + i*delta_x) for i in range(10)]
[4, 529/100, 169/25, 841/100, 256/25, 49/4, 361/25,
1681/100, 484/25, 2209/100]
```

What happened?

```
C == range(10) == [0, 1, ..., 9]
```

What happened?

`C == range(10) == [0, 1, ..., 9]`

loop 1: `i` $\longleftarrow$ `0`

`f(2 + i*delta_x)` $\rightsquigarrow$ 4

What happened?

`C == range(10) == [0, 1, ..., 9]`

loop 1: `i` $\longleftarrow$ `0`

`f(2 + i*delta_x)` $\rightsquigarrow$ `4`

loop 2: `i` $\longleftarrow$ `1`

`f(2 + i*delta_x)` $\rightsquigarrow$ `529/100`

What happened?

C == range(10) == [0, 1, ..., 9]

loop 1: i $\longleftarrow$ 0

f(2 + i*delta_x) $\rightsquigarrow$ 4

loop 2: i $\longleftarrow$ 1

f(2 + i*delta_x) $\rightsquigarrow$ 529/100

...

loop 10: i $\longleftarrow$ 9

f(2 + i*delta_x) $\rightsquigarrow$ 2209/100

More detailed example

Estimate $\int_0^1 e^{x^2}\,dx$ using n left Riemann sums.

More detailed example

Estimate $\int_0^1 e^{x^2}\,dx$ using n left Riemann sums.

- Excellent candidate for definite loop if n known from outset.
  - Riemann sum: *repeated* addition: loop!
  - n can be known to computer *but not to you*

First, *prepare pseudocode!*

Pseudocode?

description of activity

- format independent of computer language
- prefer mathematics to programming
    - “ith element of L” or “L_i”, not `L[i-1]`

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!**

Pseudocode for definite loop

statement **for** $c \in C$

Notice:

- $\in$, not `in` (mathematics, not Python)

Pseudocode for Riemann sum

— *Setup*

Let $f(x) = e^{x^2}$

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

Let $C = \{1, 2, \ldots, n\}$ — set up L—notice no Pythonese

Let $S = 0$ — S must start at 0 (no sum)

Let $X = \{a + (i-1)\Delta x \textbf{ for } i \in C\}$ — x_i is left endpoint

Let $I = \sum_{i=1}^{n} f(x_i)\Delta x$

translates to Sage as…

```
sage: a, b, n = 0, 1, 10
sage: f(x) = e**(x**2)
sage: Delta_x = (b - a)/n
sage: C = range(1,n+1)
sage: X = [a + (i - 1)*Delta_x for i in C]
sage: sum(f(x)*Delta_x for x in X)
e^(9/100) + e^(9/25) + e^(81/100) + e^(36/25) +
e^(9/4) + e^(81/25) + e^(441/100) + e^(144/25) +
e^(729/100) + 1
sage: round(_, 5)
1.3812606013
```

now use Pythonese

another way to loop

Outline

1. Repetition means Loops
2. Looping in a collection
3. Looping on a collection
4. A useful trick w/loops
5. Summary

What does it do?

`for` c `in` C`:`
  statement1
  statement2
  ...
statement outside loop

- suppose C has n elements
- *statement1*, *statement2*, etc. are repeated (looped) n times
- on ith loop, c has the value of ith element of C
- if you modify c,
  - corresponding element of C does *not* change
  - on next loop, c takes next element of C anyway
- *statement outside loop* & subsequent not repeated

Trivial example

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

Less trivial example

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

Less trivial example

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

- loop variable can be any valid identifier
- Python programmers often use `each`

What happened?

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

What happened?

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

loop 1: `f` $\longleftarrow$ `x**2`

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

What happened?

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

loop 1: `f` $\longleftarrow$ `x**2`

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

loop 2: `f` $\longleftarrow$ `cos(x)`

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

What happened?

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

loop 1: `f` $\longleftarrow$ `x**2`

`print diff(f)` $\leadsto$ `2x`

loop 2: `f` $\longleftarrow$ `cos(x)`

`print diff(f)` $\leadsto$ `-sin(x)`

loop 3: `f` $\longleftarrow$ `e**x*cos(x)`

`print diff(f)` $\leadsto$ `-e^x*sin(x) + e^x*cos(x)`

Changing *each* ?

```
sage:  C = [1,3,5]
sage:  for c in C:
         c = c + 1
         print c
2
4
6
sage:  print C
[1, 3, 5]
```

What happened?

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

What happened?

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

loop 1: `c` $\longleftarrow$ `1`

`c = c + 1` $= 1 + 1$

`print c` $\rightsquigarrow$ 2

What happened?

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

loop 1: `c` $\longleftarrow$ `1`

`c = c + 1` $= 1 + 1$

`print c` $\rightsquigarrow$ 2

loop 2: `c` $\longleftarrow$ `2`

`c = c + 1` $= 2 + 1$

`print c` $\rightsquigarrow$ 3

What happened?

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

loop 1: `c` $\longleftarrow$ `1`

`c = c + 1` $=1+1$

`print c` $\rightsquigarrow$ `2`

loop 2: `c` $\longleftarrow$ `2`

`c = c + 1` $=2+1$

`print c` $\rightsquigarrow$ `3`

loop 3: `c` $\longleftarrow$ `3`

`c = c + 1` $=3+1$

`print c` $\rightsquigarrow$ `4`

Changing C ?

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

Changing C?

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

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

Changing C ?

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

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

...infinite loop!

More detailed example

Use **Euler approximation** with 200 points to plot an approximate solution to a differential equation

$$y' = f(x,y)$$

starting at the point $(1,1)$ and ending at $x = 4$ (we'll define f later)

More detailed example

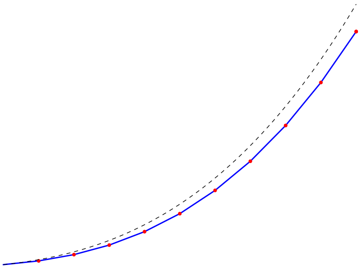
Use **Euler approximation** with 200 points to plot an approximate solution to a differential equation

$$y' = f(x, y)$$

starting at the point $(1, 1)$ and ending at $x = 4$ (we'll define f later)

Euler approximation?!?

- given a point (x_i, y_i) on the curve,
- the *next* point $(x_{i+1}, y_{i+1}) \approx (x_i + \Delta x, y_i + y' \cdot \Delta x)$



Building pseudocode

Ask yourself:

- What list(s) 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!**

Pseudocode

Let $x_0, y_0 = (1,1)$
Let $a = 1$ and $b = 4$
Let $\Delta x = {}^{b-a}/_{n}$
Let $C = (1, 2, \ldots, n)$
for $i \in C$
    $y_i = y_{i-1} + \Delta x \cdot f\left(x_{i-1}, y_{i-1}\right)$
    $x_i = x_{i-1} + \Delta x$

Translates to sage as...

```
sage: xi,yi = 1,1
sage: a,b,n = 1,4,200
sage: Delta_x = (b-a)/n
sage: for i in range(n):

  yi = yi + Delta_x * f(xi,yi)

  xi = xi + Delta_x
```

Try it!

```
sage: f(x,y) = x**2
sage: [repeat the above]
sage: xi, yi
(4, 1751009/80000)
sage: round(yi,5)
21.88761
```

What happened?

`range(n)` $\longleftarrow$ `[0, ..., 199]`

What happened?

range(n) $\longleftarrow$ [0, ..., 199]

loop 1: $i \longleftarrow$ 0

```
yi = yi + Delta_x * f(xi,yi)
```

$\rightsquigarrow$ yi = 1 + .015*1 = 1.015

```
xi = xi + Delta_x
```

$\rightsquigarrow$ xi = 1 + .015 = 1.015

What happened?

range(n) $\longleftarrow$ [0, ..., 199]

loop 1: $i \longleftarrow$ 0

```
yi = yi + Delta_x * f(xi,yi)
```

$\rightsquigarrow$ yi = 1 + .015*1 = 1.015

```
xi = xi + Delta_x
```

$\rightsquigarrow$ xi = 1 + .015 = 1.015

loop 2: $i \longleftarrow 1$

```
yi = yi + Delta_x * f(xi,yi)
```

$\rightsquigarrow$ yi = 1.015 + .015*1.030225 = 1.030453375

```
xi = xi + Delta_x
```

$\rightsquigarrow$ xi = 1.015 + .015 = 1.03

What happened?

`range(n)` $\longleftarrow$ `[0, ..., 199]`

loop 1: $i \longleftarrow$ `0`

```
yi = yi + Delta_x * f(xi,yi)
```

$\rightsquigarrow$ `yi = 1 + .015*1 = 1.015`

```
xi = xi + Delta_x
```

$\rightsquigarrow$ `xi = 1 + .015 = 1.015`

loop 2: $i \longleftarrow 1$

```
yi = yi + Delta_x * f(xi,yi)
```

$\rightsquigarrow$ `yi = 1.015 + .015*1.030225 = 1.030453375`

```
xi = xi + Delta_x
```

$\rightsquigarrow$ `xi = 1.015 + .015 = 1.03`

loop 3: $i \longleftarrow 2$

```
yi = yi + Delta_x * f(xi,yi)
```

$\rightsquigarrow$ `yi = 1.03...  + .015*1.0609 = 1.046366875`

```
xi = xi + Delta_x
```

$\rightsquigarrow$ `xi = 1.03 + .015 = 1.045`

etc.

Outline

1. Repetition means Loops
2. Looping in a collection
3. Looping on a collection
4. A useful trick w/loops
5. Summary

Looping through nonexistent lists

- `range(`n`)` creates a list of n elements
  - `for` *each* `in range(`n`)` creates the list before looping
- constructing a list, merely to repeat n times, is wasteful
  - `for` *each* `in xrange(`n`)` has same effect
  - slightly faster, uses less computer memory

Repetition means Loops

Looping in a collection

Looping on a collection

A useful trick w/loops

Summary

Outline

1. Repetition means Loops
2. Looping in a collection
3. Looping on a collection
4. A useful trick w/loops
5. Summary

Summary

- definite loop: n repetitions known at outset
- collection C of n elements controls loop
  - don't modify C
- two forms
  - loop *in* a collection, [*expression* **for** $c \in C$]
  - loop *on* a collection,

    for $c \in C$

        statement1

        statement2

        ...

    statement outside loop