# MAT 305: Mathematical Computing

## Looping with definite loops

John Perry

University of Southern Mississippi

Spring 2019

# Outline

# Outline

# Differential Equations

$$\frac{dy}{dx} = y$$

What is $y$ in terms of $x$?

# Differential Equations

$$\frac{dy}{dx} = y$$

What is $y$ in terms of $x$?

$y = Ce^x$:

$$\frac{dy}{dx} = \frac{d}{dx}(Ce^x) = C\left(\frac{d}{dx}e^x\right) = Ce^x = y$$

# Cannot always solve exactly

$$\frac{dy}{dx} = \sin y + 2\cos x$$

What is $y$ in terms of $x$?

# Cannot always solve exactly

$$\frac{dy}{dx} = \sin y + 2\cos x$$

What is $y$ in terms of $x$?

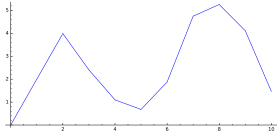
I don't know. But we need to estimate $y$ at various values of $x$.

# Euler's idea

Pick a starting point $(a, b)$.
**Repeat**

- Find tangent line at $(a, b)$.
- After all, we know point & slope $\left({}^{dy}/_{dx}\right)$
- Follow tangent line "a little ways" to another point.
- Make that point $(a, b)$.

**Until** you're "happy."




"a little ways" $= 1$

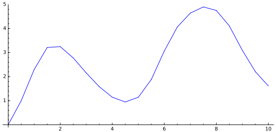
# Euler's idea

Pick a starting point $(a, b)$.

**Repeat**

- Find tangent line at $(a, b)$.
- After all, we know point & slope $(dy/dx)$
- Follow tangent line "a little ways" to another point.
- Make that point $(a, b)$.

**Until** you're "happy."




"a little ways" $= 0.5$

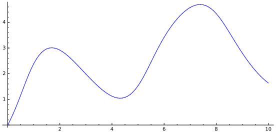
# Euler's idea

Pick a starting point $(a, b)$.

**Repeat**

- Find tangent line at $(a, b)$.
- After all, we know point & slope $({dy}/{dx})$
- Follow tangent line "a little ways" to another point.
- Make that point $(a, b)$.

**Until** you're "happy."



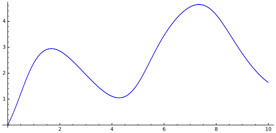

"a little ways" $= 0.1$

# Euler's idea

Pick a starting point $(a, b)$.
**Repeat**

- Find tangent line at $(a, b)$.
- After all, we know point & slope $\left({}^{dy}/_{dx}\right)$
- Follow tangent line "a little ways" to another point.
- Make that point $(a, b)$.

**Until** you're "happy."




"a little ways" $= 0.01$

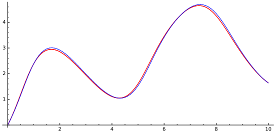
# Euler's idea

Pick a starting point $(a, b)$.

**Repeat**

- Find tangent line at $(a, b)$.
- After all, we know point & slope ($dy/dx$)
- Follow tangent line "a little ways" to another point.
- Make that point $(a, b)$.

**Until** you're "happy."




"a little ways" $= 0.1, 0.01$

# A more formal pseudocode

**algorithm** *Eulers_method*
**inputs**

- $df$, derivative of a function
- $(x_0, y_0)$, initial values of $x$ and $y$
- $\Delta x$, step size
- $n$, number of steps to take

**outputs** approximation to $f(x_0 + n\Delta x)$
**do**

let $a = x_0$, $b = y_0$
**repeat** $n$ times
  add $\Delta x \cdot df(a, b)$ to $b$
  add $\Delta x$ to $a$
**return** $b$

# Implementation

```
sage: def eulers_method(df, x0, y0, Delta_x, n):
       # starting point
       a, b = x0, y0
       # compute tangent lines & step forward
       for i in range(n):
         b = Delta_x * df(a, b) + b
         a = Delta_x + a
       return b
```

# Examples

```
sage:  df(x,y) = sin(y) + 2*cos(x)
```

# Examples

```
sage:  df(x,y) = sin(y) + 2*cos(x)
sage:  eulers_method(df, 0, 0, 1, 10)
2*cos(9) + 2*cos(8) + 2*cos(7) + 2*cos(6) + ...
```

# Examples

```
sage: df(x,y) = sin(y) + 2*cos(x)
sage: eulers_method(df, 0, 0, 1, 10)
2*cos(9) + 2*cos(8) + 2*cos(7) + 2*cos(6) + ...
```

Hmm. Anyone know what's going on here?

# Examples

```
sage: df(x,y) = sin(y) + 2*cos(x)
sage: eulers_method(df, 0, 0, 1, 10)
2*cos(9) + 2*cos(8) + 2*cos(7) + 2*cos(6) + ...
```

Hmm. Anyone know what's going on here?

```
sage: eulers_method(df, 0, 0, 1., 10)
1.46532385990369
```

# Examples

```
sage: df(x,y) = sin(y) + 2*cos(x)
sage: eulers_method(df, 0, 0, 1, 10)
2*cos(9) + 2*cos(8) + 2*cos(7) + 2*cos(6) + ...
```

Hmm. Anyone know what's going on here?

```
sage: eulers_method(df, 0, 0, 1., 10)
1.46532385990369
sage: eulers_method(df, 0, 0, 0.1, 100)
1.63761553387026
```

# Examples

```
sage: df(x,y) = sin(y) + 2*cos(x)
sage: eulers_method(df, 0, 0, 1, 10)
2*cos(9) + 2*cos(8) + 2*cos(7) + 2*cos(6) + ...
```

Hmm. Anyone know what's going on here?

```
sage: eulers_method(df, 0, 0, 1., 10)
1.46532385990369
sage: eulers_method(df, 0, 0, 0.1, 100)
1.63761553387026
sage: eulers_method(df, 0, 0, 0.01, 1000)
1.64289768319682
```

# Outline

# 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 complication

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

# The `for` command

`for` $c$ `in` $C$

where

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

# Outline

# What is a collection?

**Collection:** group of objects identified as single object

- indexed
  - tuples $(a_0, a_1, a_2, \ldots a_n)$
    - points $(x_0, y_0)$, $(x_0, y_0, z_0)$
  - lists $[a_0, a_1, \ldots, a_n]$
    - sequences $(a_0, a_1, a_2, \ldots)$
- not indexed
  - sets $\{a_0, a_5, a_3, a_2, a_1\}$
  - dictionaries

Standard Python collections

- *indexable* or *ordered* ("sequence types")
  - tuples, lists
  - access "element in position *i*" using `[i]`
    - but! start counting from `0`, **not 1**

# Tuples

**tuple**: immutable, ordered collection

- *immutable*: cannot change elements
- *indexable*: can access elements by their order
- defined using parentheses

# Example

```
sage: my_tuple = (1,5,0,5)
```
*4-tuple*

```
sage: my_tuple[2]
0
```
*access 3rd element (element 2)*

```
sage: my_tuple[2] = 1
```
*assign to 3rd element?*

*...Output deleted...*

```
TypeError: 'tuple' object does not support item
assignment
```

```
sage: my_tuple
(1,5,0,5)
```

# Lists

**list**: mutable, ordered collection

- *mutable*: can change elements
- *indexable*: can access elements by their order
- defined using square brackets

# Example

```
sage: my_list = [1,5,0,5]
```
*list of 4 elements*

```
sage: my_list[2]
0
```
*access 3rd element (element 2)*

```
sage: my_list[2] = 1
```
*assign to 3rd element?*

```
sage: my_list[2]
1
```
*no error! access gives new value!*

```
sage: my_list
[1,5,1,5]
```

# Sets

A **set** is a mutable, unordered collection

- *mutable*: can change elements
- *non-indexable*
  - cannot access elements by their order
  - computer arranges elements for efficiency
- defined using `{`*entries*`}`, `set(`*tuple or list*`)`, or `set()` (for empty set)
- redundant elements automatically deleted

# Example

```
sage: my_set = {1,5,0,5}
```
*set of 4 elements*

```
sage: my_set[2]
```
*access 3rd element?*

. . . *Output deleted*. . .

```
TypeError: 'set' object is unindexable
```

```
sage: my_set
set([0, 1, 5])
```
*so what's in there, anyway?*
*not original list!*

# More is available

You can do a *lot* with collections, but this is a *mathematics* course, not a *computer science* course. So we will stop with these basic ideas, and fill in more tools only as needed.

# Outline

# What does it do?

Comparable to set-builder notation. Mathematics:

$$\{ f(c) : c \in C \}$$

# What does it do?

Comparable to set-builder notation. Mathematics:

$$\{f(c) : c \in C\}$$

Python/sage:

[ $f(c)$ for $c$ in $C$ ]

or { $f(c)$ for $c$ in $C$ }

or ( $f(c)$ for $c$ in $C$ )

- suppose $C$ has $n$ elements
- result is a list/set/tuple
  - $i$th value is value of $f$ at $i$th element of $C$

# What does it do?

Comparable to set-builder notation. Mathematics:

$$\{ f(c) : c \in C \}$$

Python/sage:

```
[f(c) for c in C ]
or {f(c) for c in C }
or (f(c) for c in C )
```

- suppose $C$ has $n$ elements
- result is a list/set/tuple
  - $i$th value is value of $f$ at $i$th element of $C$
- loop variable $c$ can be any valid identifier

# Examples

## Example

Build a list of even numbers from 2 to 40. Use `range(20)` to help.

# Examples

## Example

Build a list of even numbers from 2 to 40. Use `range(20)` to help.

- `range(20)` gives us the list `[ 0, 1, ... 19 ]`
- How can we map those numbers to 2, 4, ..., 40?

# Examples

## Example

Build a list of even numbers from 2 to 40. Use `range(20)` to help.

- `range(20)` gives us the list `[ 0, 1, ... 19 ]`
- How can we map those numbers to 2, 4, …, 40?
  $f(x) = 2(x+1)$

# Examples

## Example

Build a list of even numbers from 2 to 40. Use `range(20)` to help.

- `range(20)` gives us the list `[ 0, 1, ... 19 ]`
- How can we map those numbers to 2, 4, ..., 40? $f(x) = 2(x+1)$

```
sage: f(x) = 2*(x + 1)
sage: [ f(i) for i in range(20) ]
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28,
30, 32, 34, 36, 38, 40]
```

# Examples

## Example

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` ⟵ `0`

`f(2 + i*delta_x)` ⇝ `4`

loop 2: `i` ⟵ `1`

`f(2 + i*delta_x)` ⇝ `529/100`

# What happened?

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

loop 1: `i` ⟵ `0`

`f(2 + i*delta_x)` ⇝ `4`

loop 2: `i` ⟵ `1`

`f(2 + i*delta_x)` ⇝ `529/100`

...

loop 10: `i` ⟵ `9`

`f(2 + i*delta_x)` ⇝ `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
  - 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
  - "$i$th 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)

# Riemann sum: review

Let $\Delta x$ be width of partition
Let $X$ be left endpoints of partition
Add areas of rectangles on each partition

# Riemann sum: pseudocode

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

Let $X = \{a + (i-1)\Delta x \textbf{ for } i \in \{1, \ldots, n\}\}$ $x_i$ 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                              setup
sage: f(x) = e^(x^2)                                  setup
sage: Delta_x = (b - a)/n                             translation
sage: C = range(1,n+1)                                Python shortcut
sage: X = [a + (i - 1)*Delta_x for i in C]
sage: I = sum(f(x)*Delta_x for x in X)                thanks, Sage!
sage: I
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
```

# Outline

# 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 $i$th loop, $c$ has the value of $i$th 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

# 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)
```

# 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` ⟵ `x^2`

`print diff(f)` ⇝ 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` ⟵ `x^2`
`print diff(f)` ⇝ `2x`

loop 2: `f` ⟵ `cos(x)`
`print diff(f)` ⇝ `-sin(x)`

loop 3: `f` ⟵ `e^x*cos(x)`
`print diff(f)` ⇝ `-e^x*sin(x) + e^x*cos(x)`

# Changing loop variable?

```
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

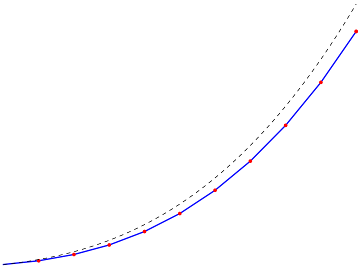
Use **Euler approximation** with 200 points to plot an approximate solution to a differential equation

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

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 $\left(x_{i+1}, y_{i+1}\right) \approx \left(x_i + \Delta x, y_i + y' \cdot \Delta x\right)$



# 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 $x_0, y_0 = (1, 1)$ | setup |
| Let $a = 1$ and $b = 4$ | ... |
| Let $\Delta x = {}^{b-a}/_{n}$ | ... |
| Let $C = (1, 2, \ldots, n)$ | collection over which to iterate |
| **for** $i \in C$ | loop |
| $\quad y_i = y_{i-1} + \Delta x \cdot f\left(x_{i-1}, y_{i-1}\right)$ | Euler approximation |
| $\quad x_i = x_{i-1} + \Delta x$ | move to next $x$ |

# Translates to sage as…

```
sage: XY = [(1,1)]                                   XY will be list of points
sage: a,b,n = 1,4,200                                                    setup
sage: Delta_x = (b-a)/n                                                    ...
sage: for i in range(n):                                                  loop
        XY.append((X[i] + Delta_x,
                   Y[i] + Delta_x * f(X[i],Y[i])))
```

## Try it!

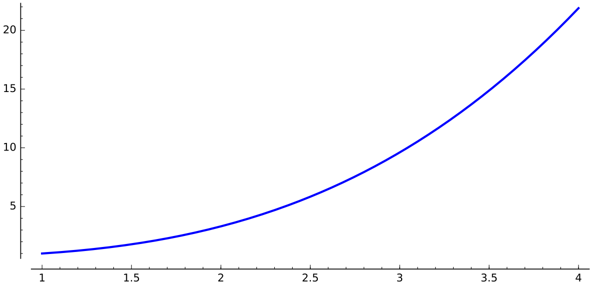
```
sage:  f(x,y) = x^2
sage:  [type the above]
sage:  XY[-1]
(4, 1751009/80000)
sage:  round(_,5)
21.88761
```

last entry in XY

# Try it!

```
sage: f(x,y) = x^2
sage: [type the above]
sage: XY[-1]
(4, 1751009/80000)
sage: round(_,5)
21.88761
sage: line(XY,thickness=2)
```

last entry in XY



# What happened?

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

# What happened?

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

loop 1: $i \longleftarrow$ `0`

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

$y_i$ = $y_i$ + Delta_x * f($x_{i-1}$,$y_{i-1}$)

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

# What happened?

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

loop 1: $i \longleftarrow$ `0`

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

$y_i$ = $y_i$ + `Delta_x * f(`$x_{i-1}$`,`$y_{i-1}$`)`

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

loop 2: $i \longleftarrow 1$

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

$y_i$ = $y_i$ + `Delta_x * f(`$x_{i-1}$`,`$y_{i-1}$`)`

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

# What happened?

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

loop 1: $i \longleftarrow$ `0`

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

$y_i$ `= ` $y_i$ `+ Delta_x * f(`$x_{i-1}$`,`$y_{i-1}$`)`

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

loop 2: $i \longleftarrow 1$

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

$y_i$ `= ` $y_i$ `+ Delta_x * f(`$x_{i-1}$`,`$y_{i-1}$`)`

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

loop 3: $i \longleftarrow 2$

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

$y_i$ `= ` $y_i$ `+ Delta_x * f(`$x_{i-1}$`,`$y_{i-1}$`)`

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

etc.

# Outline

# Example problem

## Problem

Given $f$, $a$, $b$, and $n$, use $n$ rectangles to approximate $\int_a^b f(x)\, dx$. Use left endpoints to approximate the height of each rectangle.

# Function definition

How can we make this interactive? Let user define:

- $f$, $a$, $b$ as input boxes
- $n$ as slider from 2 to 10
- color of boxes

# Function definition

How can we make this interactive? Let user define:

- $f$, $a$, $b$ as input boxes
- $n$ as slider from 2 to 10
- color of boxes

∴ function definition:

```
@interact
def
i_left_sums(f=input_box(default=x^2,label='$f$'),
            a=input_box(default=0,label='$a$'),
            b=input_box(default=1,label='$b$'),
            n=slider(start=range(2,11),default=2,
                     label='$n$'),
            boxcolor=Color(0.5,0.5,0.5)):
```

# Avoid complicated functions

Major subtasks ⟶ functions:

- `left_Riemann_sum()` to approximate area
- `left_Riemann_rectangles()` to make plots

# Approximating area

- Already solved approximation of $\int_a^b f(x)\,dx$ using left endpoints. ***Reuse old work!***
- Prior to `@interact`, paste old left Riemann sum code.

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

# Graphics

- plotting $f$ is easy

```
fplot = plot(f,a,b)
```

# Graphics

- plotting $f$ is easy
  `fplot = plot(f,a,b)`
- plotting rectangles: use `polygon2d()` command
  `polygon2d([`*lower_left*`,` *upper_left*`,`
  *upper_right*`,` *lower_right*`])`
- use **for** loop to combine rectangles into plot

# Graphics

- plotting $f$ is easy

```
fplot = plot(f,a,b)
```

- plotting rectangles: use `polygon2d()` command

```
polygon2d([lower_left, upper_left,
           upper_right, lower_right])
```

- use **for** loop to combine rectangles into plot

```
combo = fplot
L = range(n)
for i in L:
  xi = a + i*Delta_x
  yi = f(x)
  combo = combo + polygon2d([(xi,0),(xi,yi),
          (xi+Delta_x,yi),(xi+Delta_x,0)],
          color=boxcolor,alpha=0.75)
```

# Encapsulate as function

Also prior to `@interact`:

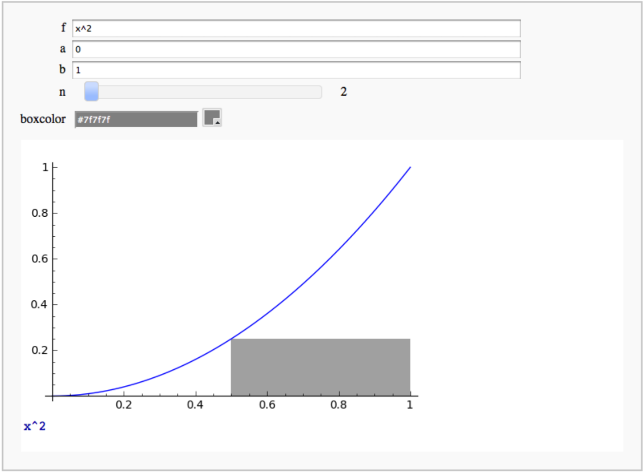
```
def left_Riemann_rectangles(f, a, b, n,
                                 x=x, boxcolor='red'):
  fplot = plot(f,a,b)
  combo = fplot
  Delta_x = (b-a)/n
  L = range(n)
  for i in L:
    xi = a + i*Delta_x
    yi = f({x:xi})
    combo = combo + polygon2d([(xi,0),(xi,yi),
                (xi+Delta_x,yi),(xi+Delta_x,0)],
                color=boxcolor,alpha=0.75)
  return combo
```

# Combine pieces

Call both from `i_left_sums()`:

```
@interact
def i_left_sums(f=input_box(default=x^2),
                ...
                boxcolor=Color(0.5,0.5,0.5)):
  approx = left_Riemann_sum(f,a,b,n)
  riemann_plot = left_Riemann_rectangles(f,a,b,n,
                                         boxcolor)
  show(riemann_plot)
  print approx
```

# The final product



# Outline

## 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*
- watch for *infinite loops*

"infinite loop": *see* infinite loop

— *AmigaDOS* Glossary, ca. 1993