Procedures

Arguments to procedures

Extended example

Returning values

Continuing the example

Pseudocode

Scripting

Summary

MAT 305: Mathematical Computing

Procedures

John Perry

University of Southern Mississippi

Spring 2019

Procedures

Arguments to procedures

Extended example

Returning values

Continuing the example

Pseudocode

Scripting

Summary

Outline

1. Procedures
2. Arguments to procedures
3. Extended example
4. Returning values
5. Continuing the example
6. Pseudocode
7. Scripting
8. Summary

Procedures

Arguments to procedures

Extended example

Returning values

Continuing the example

Pseudocode

Scripting

Summary

Outline

1 Procedures

2 Arguments to procedures

3 Extended example

4 Returning values

5 Continuing the example

6 Pseudocode

7 Scripting

8 Summary

Procedures?

- sequence of statements organized as one command
  - may return one or more values
- other languages:

Ada, Pascal	`procedure` and `function`
BASIC	subroutine
C, C++, Kotlin, Python, VB	function
Eiffel	`feature`
Java	method

- We use “procedure” to avoid confusion w/mathematical functions

Why procedures?

- avoid retyping code
  - many patterns repeated
  - same behavior, different data
- organization, abstraction
- easier to read, maintain

Defining a procedure

`def` *name*`(` *argument1*`=`*default1* `,` *argument2*`=`*default2* `, ...):`

statement1

statement2

...

where

- *name* is an identifier
- *arguments* (optional) are identifiers
- *defaults* (optional) are default values for the corresponding arguments

Procedures

Arguments to procedures

Extended example

Returning values

Continuing the example

Pseudocode

Scripting

Summary

Defining a procedure

`def` *name*`(` *argument1=default1* `,` *argument2=default2* `, ...):`

statement1

statement2

...

where

- *name* is an identifier
- *arguments* (optional) are identifiers
- *defaults* (optional) are default values for the corresponding arguments

***not** optional:*

- `:`, `()`, `def`
- *at least one statement*
- *indent all statements* in procedure

Calling a procedure

- once `p` is defined, call using `p()`
- supply data for arguments without default values

Example

```
def greetings():
  # greet the user
  print 'Greetings!'
```

- name of procedure is `greetings`
- no arguments
- one statement
- one comment (`# greet the user`)
  - ignored by computer
  - useful for programmers

Example

Try it!

```
sage:  def greetings():
         # greet the user
         print 'Greetings!'
sage:  greetings()
Greetings!
```

Procedures

Arguments to procedures

Extended example

Returning values

Continuing the example

Pseudocode

Scripting

Summary

Outline

1 Procedures

2 Arguments to procedures

3 Extended example

4 Returning values

5 Continuing the example

6 Pseudocode

7 Scripting

8 Summary

Arguments?

placeholders for data

scope: identifier valid only inside procedure where it is defined

- data still exists outside procedure
- modifying argument does not modify original data, but creates new data
  - *caveat:* contents of lists and sets can be modified
- value of data forgotten immediately after procedure concludes

Example

```
def greetings(name='Leonhard Euler'):
  print 'Greetings,', name
```

- name of procedure is `greetings`
- one argument, `name`
  - default value: `'Leonhard Euler'`

Example

Try it!

```
sage:  def greetings(name='Leonhard Euler'):
         print 'Greetings,', name

sage:  greetings()
Greetings, Leonhard Euler
```

Example

Try it!

```
sage:  def greetings(name='Leonhard Euler'):
           print 'Greetings,', name

sage:  greetings()
Greetings, Leonhard Euler

sage:  greetings('Pythagoras')
Greetings, Pythagoras
```

Example

Try it!

```
sage:  def greetings(name='Leonhard Euler'):
           print 'Greetings,', name

sage:  greetings()
Greetings, Leonhard Euler

sage:  greetings('Pythagoras')
Greetings, Pythagoras

sage:  greetings(name='Pythagoras')
Greetings, Pythagoras
```

Example

Try it!

```
sage:  def greetings(name='Leonhard Euler'):
         print 'Greetings,', name

sage:  greetings()
Greetings, Leonhard Euler

sage:  greetings('Pythagoras')
Greetings, Pythagoras

sage:  greetings(name='Pythagoras')
Greetings, Pythagoras

sage:  greetings(pi)
Greetings, pi
```

Warning 1

Don't use uninitialized identifiers

```
sage:  def greetings(name='Leonhard Euler'):
         print 'Greetings,', name

sage:  greetings(Pythagoras)                    oops: no quotes!
...Output deleted...
NameError:  name 'Pythagoras' is not defined
```

Warning 2

Scope implies `name` does not exist outside `hello`

```
sage:  def greetings(name='Leonhard Euler'):
         print 'Greetings,', name

sage:  greetings('Pythagoras')
Hello, Pythagoras

sage:  name
...Output deleted...
NameError:  name 'name' is not defined
```

`name` *out of scope*

Warning 3

Scope implies `name` forgotten once `hello` concludes

```
sage:  def greetings(name='Leonhard Euler'):
           print 'Greetings,', name

sage:  greetings('Pythagoras')
Hello, Pythagoras

sage:  greetings()
Hello, Leonhard Euler
```

name has value `'Leonhard Euler'` *again*

Warning 4

Can change value inside procedure, but value outside procedure remains the same

```
sage:  def mischievous_hello(name='world'):
           name = 'loser!'
           print 'Hello,', name
```

Warning 4

Can change value inside procedure, but value outside procedure remains the same

```
sage:  def mischievous_hello(name='world'):
           name = 'loser!'
           print 'Hello,', name

sage:  print_name = 'Dr.  Perry'
sage:  mischievous_hello(print_name)
Hello, loser!
```

value of `name` *changed in procedure*

Warning 4

Can change value inside procedure, but value outside procedure remains the same

```
sage:  def mischievous_hello(name='world'):
         name = 'loser!'
         print 'Hello,', name

sage:  print_name = 'Dr.  Perry'
sage:  mischievous_hello(print_name)
Hello, loser!                    value of name changed in procedure

sage:  print_name
'Dr.  Perry'                     value of print_name unchanged
```

Warning 5

If defaults are not given to arguments, you must supply something

```
sage:  def greetings(name):
           print 'Greetings,', name

sage:  greetings()
```

...*Output deleted*...

```
TypeError:  greetings() takes exactly 1 argument (0 given)
```

Procedures

Arguments to procedures

Extended example

Returning values

Continuing the example

Pseudocode

Scripting

Summary

Outline

1 Procedures

2 Arguments to procedures

3 Extended example

4 Returning values

5 Continuing the example

6 Pseudocode

7 Scripting

8 Summary

Procedures

Arguments to procedures

Extended example

Returning values

Continuing the example

Pseudocode

Scripting

Summary

A mathematical problem

Given function f and real number a

find line tangent to f at $x = a$

Procedures

Arguments to procedures

Extended example

Returning values

Continuing the example

Pseudocode

Scripting

Summary

A mathematical problem

Given function f and real number a

find line tangent to f at $x = a$

How should we implement this?

Procedures

Arguments to procedures

Extended example

Returning values

Continuing the example

Pseudocode

Scripting

Summary

A mathematical problem

Given function f and real number a

find line tangent to f at $x = a$

How should we implement this?

```
sage:  def tangent_line(f, a):
           # point-slope form of a line
```

Procedures

Arguments to procedures

Extended example

Returning values

Continuing the example

Pseudocode

Scripting

Summary

A mathematical problem

Given function f and real number a

find line tangent to f at $x = a$

How should we implement this?

```
sage:  def tangent_line(f, a):
           # point-slope form of a line
           b = f(a)
           df(x) = diff(f,x)
           m = df(a)
           result = m*(x - a) + b
```

A mathematical problem

Given function f and real number a

find line tangent to f at $x = a$

How should we implement this?

```
sage:  def tangent_line(f, a):
           # point-slope form of a line
           b = f(a)
           df(x) = diff(f,x)
           m = df(a)
           result = m*(x - a) + b
           print 'The line tangent to', f,
           print 'at x =', a, 'is',
           print result
```

Quick test

```
sage:  f(x) = x^2
sage:  tangent_line(x^2, 1)
The line tangent to x |--> x^2
at x = 1 is
2*x - 1
```

Trouble is...

...it sometimes complains!

```
sage:  tangent_line(x^2, 1)
```

...*Output deleted*...

```
DeprecationWarning:  Substitution using
function-call syntax and unnamed arguments is
deprecated and will be removed from a future release
of Sage; you can use named arguments instead, like
EXPR(x=..., y=...)
```

...*Output deleted*...

```
The line tangent to x^2 at x = 1 is 2*x - 1
```

Trouble is...

...it sometimes complains!

```
sage:  tangent_line(x^2, 1)
```

...*Output deleted*...

```
DeprecationWarning:  Substitution using
function-call syntax and unnamed arguments is
deprecated and will be removed from a future release
of Sage; you can use named arguments instead, like
EXPR(x=..., y=...)
```

...*Output deleted*...

```
The line tangent to x^2 at x = 1 is 2*x - 1
```

Problematic line: `b = f(a)`

- `f` has value `x^2` , an *expression*
  - `b = f(a)` wants f to be a *function*

Solution #1

Define `f` as function, call `tangent_line()` w/`f`

```
sage:  f(x) = x^2
sage:  tangent_line(f, 1)
The line tangent to x |--> x^2 at x = 1 is 2*x - 1
```

Solution #1

Define `f` as function, call `tangent_line()` w/`f`

```
sage:  f(x) = x^2
sage:  tangent_line(f, 1)
The line tangent to x |--> x^2 at x = 1 is 2*x - 1
```

Inconvenient, though: `tangent_line(x^2, 1)` would be useful.

Solution #2

Specify variable of substitution in procedure:

```
sage:  def tangent_line(f, a):
         # point-slope form of a line
         b = f(x=a)
         df(x) = diff(f,x)
         ...
```

Solution #2

Specify variable of substitution in procedure:

```
sage:  def tangent_line(f, a):
         # point-slope form of a line
         b = f(x=a)
         df(x) = diff(f,x)
         ...
sage:  tangent_line(x^2, 1)
The line tangent to x |--> x^2 at x = 1 is 2*x - 1
```

Also undesirable. *Why?*

Solution #2

Specify variable of substitution in procedure:

```
sage:  def tangent_line(f, a):
         # point-slope form of a line
         b = f(x=a)
         df(x) = diff(f,x)
         ...
sage:  tangent_line(x^2, 1)
The line tangent to x |--> x^2 at x = 1 is 2*x - 1
```

Also undesirable. *Why?*

```
sage:  var('t')
sage:  tangent_line(t^2, 1)
The line tangent to x |--> t^2 at x = 1 is t^2
```

Solution #3

Redefine expression as function:

```
sage:  def tangent_line(f, a):
           # redefine f
           f(x) = f
           # point-slope form of a line
           b = f(a)
           ...
```

Solution #3

Redefine expression as function:

```
sage:  def tangent_line(f, a):
           # redefine f
           f(x) = f
           # point-slope form of a line
           b = f(a)
           ...
sage:  tangent_line(x^2, 1)
The line tangent to t |--> t^2 at x = 1 is 2*x - 1
```

Better, but insufficient. *Why?*

Solution #3

Redefine expression as function:

```
sage:  def tangent_line(f, a):
           # redefine f
           f(x) = f
           # point-slope form of a line
           b = f(a)
           ...
sage:  tangent_line(x^2, 1)
The line tangent to t |--> t^2 at x = 1 is 2*x - 1
```

Better, but insufficient. ***Why?***
Why should t change to x?

Best (?) solution

Redefine as function in indeterminate:

```
sage:  def tangent_line(f, a, x=x):
           # redefine f
           f(x) = f
           # point-slope form of a line
           b = f(a)
           df(x) = diff(f,x)
           m = df(a)
           result = m*(x - a) + b
           print 'The line tangent to', f,
           print 'at', x, '=', a, 'is',
           print result
```

Best (?) solution

Redefine as function in indeterminate:

```
sage:  def tangent_line(f, a, x=x):
           # redefine f
           f(x) = f
           # point-slope form of a line
           b = f(a)
           df(x) = diff(f,x)
           m = df(a)
           result = m*(x - a) + b
           print 'The line tangent to', f,
           print 'at', x, '=', a, 'is',
           print result
sage:  tangent_line(t^2, 1)
The line tangent to t |--> t^2 at t = 1 is 2*t - 1
```

Procedures

Arguments to procedures

Extended example

Returning values

Continuing the example

Pseudocode

Scripting

Summary

Outline

1 Procedures

2 Arguments to procedures

3 Extended example

4 Returning values

5 Continuing the example

6 Pseudocode

7 Scripting

8 Summary

Returning values?

- procedures compute values
- Often want to work with what we've computed
  - `print` command unhelpful
  - copy and paste annoying

Example

Derivative necessary to compute many things:

- graph tangent line
- analyze concavity
- identify optimum values
- …

Procedures

Arguments to procedures

Extended example

Returning values

Continuing the example

Pseudocode

Scripting

Summary

The `return` command

`return` *value1*, *value2*, ...

- reports the data values *value1*, *value2*, etc. to caller
- only works inside procedures

Procedures

Arguments to procedures

Extended example

Returning values

Continuing the example

Pseudocode

Scripting

Summary

Outline

1 Procedures

2 Arguments to procedures

3 Extended example

4 Returning values

5 Continuing the example

6 Pseudocode

7 Scripting

8 Summary

Problem

Write a procedure whose arguments are

- a function f,
- numbers $[a, b]$,
- a number $c \in (a, b)$,
- an indeterminate x,

and returns both f and the line tangent to it at $x = c$ on $[a, b]$.

Procedures

Arguments to procedures

Extended example

Returning values

Continuing the example

Pseudocode

Scripting

Summary

Divide and conquer

How can we break this into smaller tasks?

Divide and conquer

How can we break this into smaller tasks?

1. Compute line tangent to f at c
2. Plot f on $[a, b]$
3. Plot line on $[a, b]$
4. Return their sum

Divide and conquer

How can we break this into smaller tasks?

1. Compute line tangent to f at c — already solved
2. Plot f on $[a, b]$
3. Plot line on $[a, b]$
4. Return their sum

Procedures

Arguments to procedures

Extended example

Returning values

Continuing the example

Pseudocode

Scripting

Summary

Tangent line: modify implementation

```
sage:  def tangent_line(f, a, x=x):
           # redefine f
           f(x) = x
           # point-slope form of a line
           b = f(x=a)
           df(x) = diff(f,x)
           m = df(a)
           result = m*(x - a) + b
           return result
```

Procedures

Arguments to procedures

Extended example

Returning values

Continuing the example

Pseudocode

Scripting

Summary

Tangent line: modify implementation

```
sage:  def tangent_line(f, a, x=x):
         # redefine f
         f(x) = x
         # point-slope form of a line
         b = f(x=a)
         df(x) = diff(f,x)
         m = df(a)
         result = m*(x - a) + b
         return result
sage:  tangent_line(t^2, 1, t)
2*t - 1
```

Use to plot

```
sage:  def tangent_plot(f, a, b, c, x=x):
```

Use to plot

```
sage:  def tangent_plot(f, a, b, c, x=x):
           # compute the line
```

Use to plot

```
sage:  def tangent_plot(f, a, b, c, x=x):
           # compute the line
           tan_line = tangent_line(f, c, x)
```

Use to plot

```
sage:  def tangent_plot(f, a, b, c, x=x):
         # compute the line
         tan_line = tangent_line(f, c, x)
         # plot of f
```

Use to plot

```
sage:  def tangent_plot(f, a, b, c, x=x):
           # compute the line
           tan_line = tangent_line(f, c, x)
           # plot of f
           f_plot = plot(f, a, b)
```

Use to plot

```
sage:  def tangent_plot(f, a, b, c, x=x):
           # compute the line
           tan_line = tangent_line(f, c, x)
           # plot of f
           f_plot = plot(f, a, b)
           # plot of line
```

Use to plot

```
sage:  def tangent_plot(f, a, b, c, x=x):
         # compute the line
         tan_line = tangent_line(f, c, x)
         # plot of f
         f_plot = plot(f, a, b)
         # plot of line
         tan_plot = plot(tan_line, a, b)
```

Use to plot

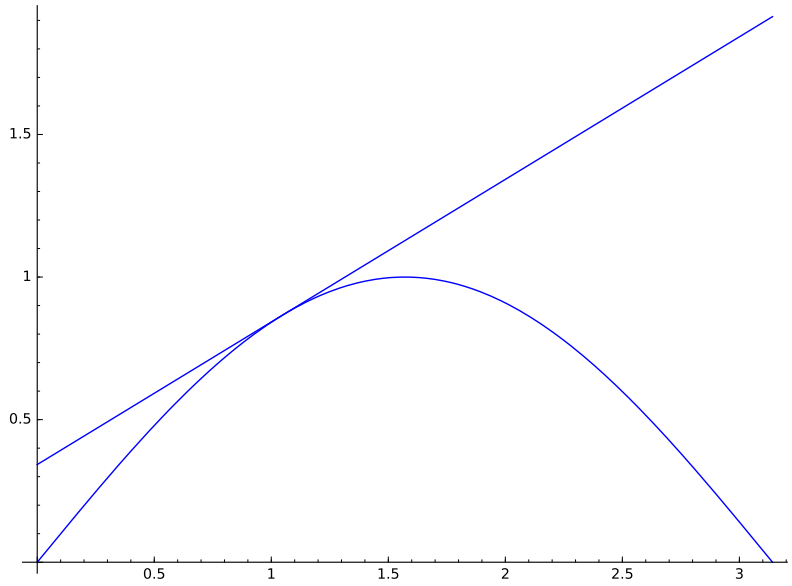
```
sage:  def tangent_plot(f, a, b, c, x=x):
         # compute the line
         tan_line = tangent_line(f, c, x)
         # plot of f
         f_plot = plot(f, a, b)
         # plot of line
         tan_plot = plot(tan_line, a, b)
         # combine & return
```

Use to plot

```
sage:  def tangent_plot(f, a, b, c, x=x):
         # compute the line
         tan_line = tangent_line(f, c, x)
         # plot of f
         f_plot = plot(f, a, b)
         # plot of line
         tan_plot = plot(tan_line, a, b)
         # combine & return
         return f_plot + tan_plot
```

Result

```
sage: tangent_plot(sin(x), 0, pi, pi/3)
```



Procedures

Arguments to procedures

Extended example

Returning values

Continuing the example

Pseudocode

Scripting

Summary

Outline

1 Procedures

2 Arguments to procedures

3 Extended example

4 Returning values

5 Continuing the example

6 Pseudocode

7 Scripting

8 Summary

Pseudocode?

description of algorithm

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

Our pseudocode format

algorithm *name*

inputs

input1 $\in$ *domain1*

input2, *description of type*

…

outputs

output1, *relationship to inputs*

output2, *relationship to inputs*

…

do

English or mathematical statement 1

English or mathematical statement 2

…

Our pseudocode format

algorithm *name*

inputs

input1 $\in$ *domain1*

input2, *description of type*

…

outputs

output1, *relationship to inputs*

output2, *relationship to inputs*

…

do

English or mathematical statement 1

English or mathematical statement 2

…

Try it now on the given problem

Example problem

Write a program to compute the line tangent to $f(x)$ at $x = x_0$.

- Write pseudocode answering:
  1. What inputs will we need?
     - domain of each input (what set/type of object)
  2. What outputs do we expect?
     - inputs' purpose & relationship to output
  3. How do we use the inputs to generate the output?
     - think step-by-step
     - do a sample problem: $f(x) = x^2$, $x_0 = 3$
     - think about possible errors errors
- Implement pseudocode

Example pseudocode

algorithm *tangent_line*

inputs

f, a function of a variable x

$x_0 \in \mathbb{R}$

Example pseudocode

algorithm *tangent_line*

inputs

f, a function of a variable x

$x_0 \in \mathbb{R}$

outputs

the line tangent to $f(x)$ at $x = x_0$

Procedures

Arguments to procedures

Extended example

Returning values

Continuing the example

Pseudocode

Scripting

Summary

Example pseudocode

algorithm *tangent_line*

inputs

f, a function of a variable x

$x_0 \in \mathbb{R}$

outputs

the line tangent to $f(x)$ at $x = x_0$

do

— *We need two things for a line: a point* (x_0, y_0) *and the slope* m

Procedures

Arguments to procedures

Extended example

Returning values

Continuing the example

Pseudocode

Scripting

Summary

Example pseudocode

algorithm *tangent_line*

inputs

f, a function of a variable x

$x_0 \in \mathbb{R}$

outputs

the line tangent to $f(x)$ at $x = x_0$

do

— *We need two things for a line: a point* (x_0, y_0) *and the slope* m

Let $y_0 = f(x_0)$

— *Use Calculus to find* m

Procedures

Arguments to procedures

Extended example

Returning values

Continuing the example

Pseudocode

Scripting

Summary

Example pseudocode

algorithm *tangent_line*

inputs

f, a function of a variable x

$x_0 \in \mathbb{R}$

outputs

the line tangent to $f(x)$ at $x = x_0$

do

— *We need two things for a line: a point* (x_0, y_0) *and the slope* m

Let $y_0 = f(x_0)$

— *Use Calculus to find* m

Let $fderiv = f'(x)$

Let $m = fderiv(x_0)$

Example pseudocode

algorithm *tangent_line*

inputs

f, a function of a variable x

$x_0 \in \mathbb{R}$

outputs

the line tangent to $f(x)$ at $x = x_0$

do

— *We need two things for a line: a point* (x_0, y_0) *and the slope* m

Let $y_0 = f(x_0)$

— *Use Calculus to find* m

Let $fderiv = f'(x)$

Let $m = fderiv(x_0)$

— *Point-slope form:* $y - y_0 = m(x - x_0)$

Let $line = m(x - x_0) + y_0$

Example pseudocode

algorithm *tangent_line*

inputs

f, a function of a variable x

$x_0 \in \mathbb{R}$

outputs

the line tangent to $f(x)$ at $x = x_0$

do

— *We need two things for a line: a point* (x_0, y_0) *and the slope* m

Let $y_0 = f(x_0)$

— *Use Calculus to find* m

Let $fderiv = f'(x)$

Let $m = fderiv(x_0)$

— *Point-slope form:* $y - y_0 = m(x - x_0)$

Let $line = m(x - x_0) + y_0$

return *line*

Example implementation

Compare to implementation:

```
def tangent_line(f, a, x=x):
        # redefine f
        f(x) = x
        # point-slope form of a line
        b = f(x=a)
        df(x) = diff(f,x)
        m = df(a)
        result = m*(x - a) + b
        return result
```

Example implementation

Compare to implementation:

```
def tangent_line(f, a, x=x):
        # redefine f
        f(x) = x
        # point-slope form of a line
        b = f(x=a)
        df(x) = diff(f,x)
        m = df(a)
        result = m*(x - a) + b
        return result
```

Specify x as input to pseudocode?

- not inherent to *problem*, but to *programming*, so
- do not include in pseudocode, but
- this is arguable

Example run

Behold: *the power of symbolic computation!*

```
sage:  var('a b c')
sage:  tangent_line(a*x^2 + b*x + c, 1, x)
(2*a + b)*(x - 1) + a + b + c
```

Combine with plots

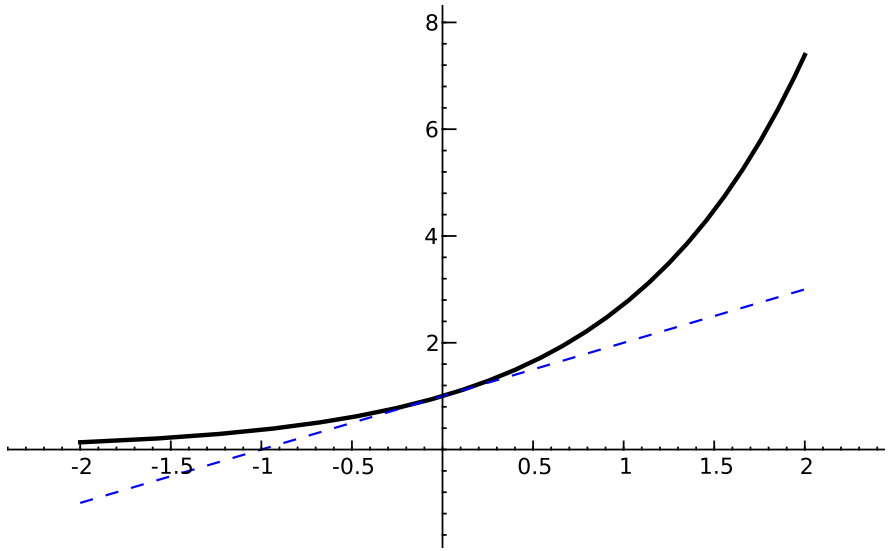
We show the plots of e^x and its tangent line at $x = 0$

```
sage:  f = e^t
sage:  tanline = tangent_line(f, 0, t)
sage:  fplot = plot(f, -2, 2, color='black',
                    thickness=2)
sage:  lineplot = plot(tanline, -2, 2,
linestyle='dashed')
sage:  fplot + lineplot
```

Combine with plots

We show the plots of e^x and its tangent line at $x = 0$

```
sage:  f = e^t
sage:  tanline = tangent_line(f, 0, t)
sage:  fplot = plot(f, -2, 2, color='black',
                    thickness=2)
sage:  lineplot = plot(tanline, -2, 2,
linestyle='dashed')
sage:  fplot + lineplot
```



Procedures

Arguments to procedures

Extended example

Returning values

Continuing the example

Pseudocode

Scripting

Summary

Combining procedures

It would be nice to have a procedure that graphs an arbitrary $f(x)$ and its tangent line at $x = x_0$. Options include:

Combining procedures

It would be nice to have a procedure that graphs an arbitrary $f(x)$ and its tangent line at $x = x_0$. Options include:

- Repeat previous commands for each f and each x_0

a lot of work!

Combining procedures

It would be nice to have a procedure that graphs an arbitrary $f(x)$ and its tangent line at $x = x_0$. Options include:

- Repeat previous commands for each f and each x_0

  a lot of work!

- Encapsulate commands in another procedure

Pseudocode

algorithm *plot_function_and_tangent*

inputs

f, a function of a variable x

$x_0 \in \mathbb{R}$

outputs

the plot of $f(x)$ and the line tangent to f at $x = x_0$

do

Let P_1 be the plot of $f(x)$ in a neighborhood of x_0

Let $g(x)$ be the line tangent to f at x_0 *Already solved!*

Let P_2 be the plot of $g(x)$ in the same neighborhood of x_0

return P_1 and P_2 combined

Implementation

```
def plot_function_and_tangent(f, x0, x=x,
      xmin=-2, xmax=2):
```

Whitespace

```
  # plots f(x) and line tangent to f at x0
  # over [ xmin, xmax ];
  # returns combination of these plots
```

distinguishes

```
  P1 = plot(f, xmin, xmax, rgbcolor='black',
        thickness=2)
```

different

```
  # next line reuses previous code
  g = tangent_line(f, x0, x)
```

tasks

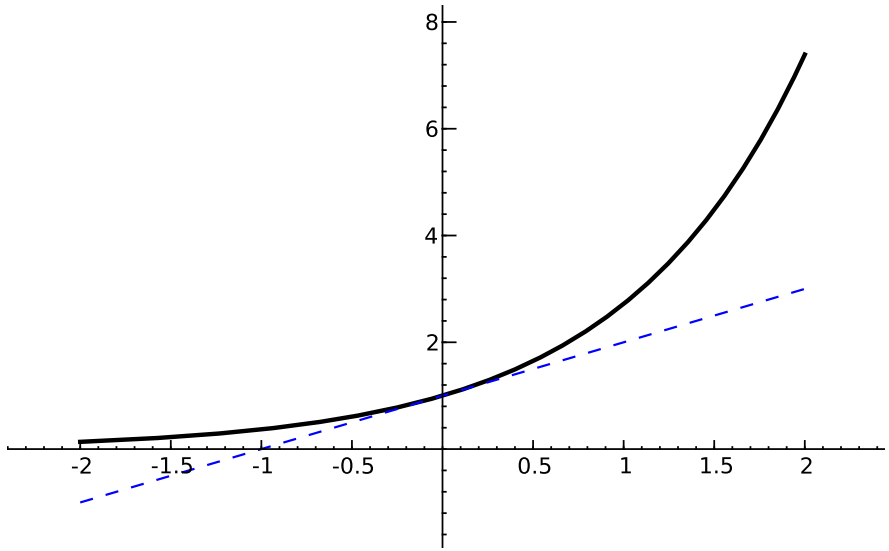
```
  P2 = plot(g, xmin, xmax, linestyle='dashed')
  return P1 + P2
```

Examples

```
sage:  def plot_function_and_tangent...
```

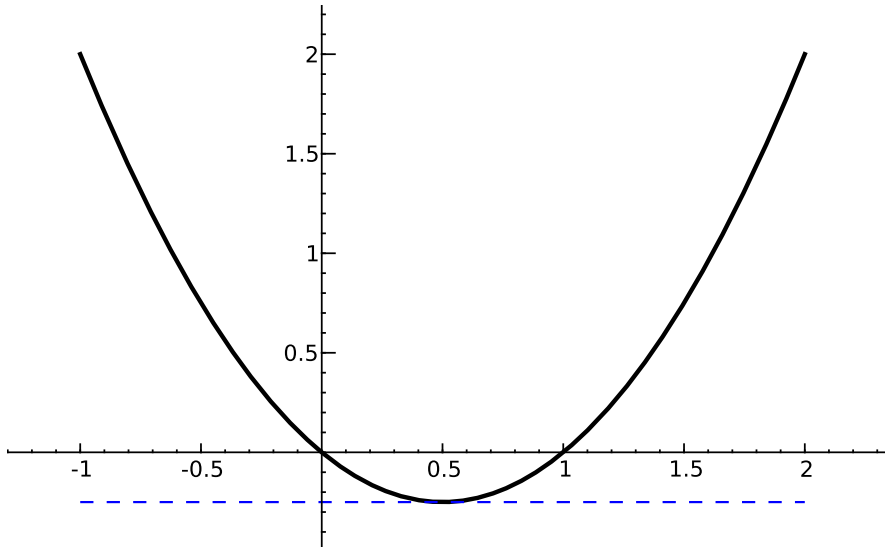
Examples

```
sage:  def plot_function_and_tangent...
sage:  plot_function_and_tangent(e^x,x)
```



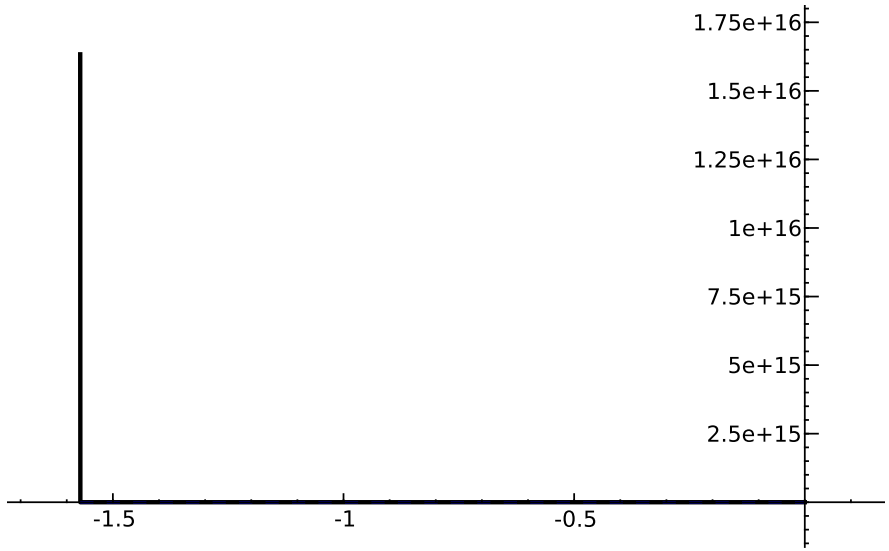
Examples

```
sage:  def plot_function_and_tangent...
sage:  plot_function_and_tangent(x^2-x,0.5,xmin=-1,
         xmax=2)
```



Examples

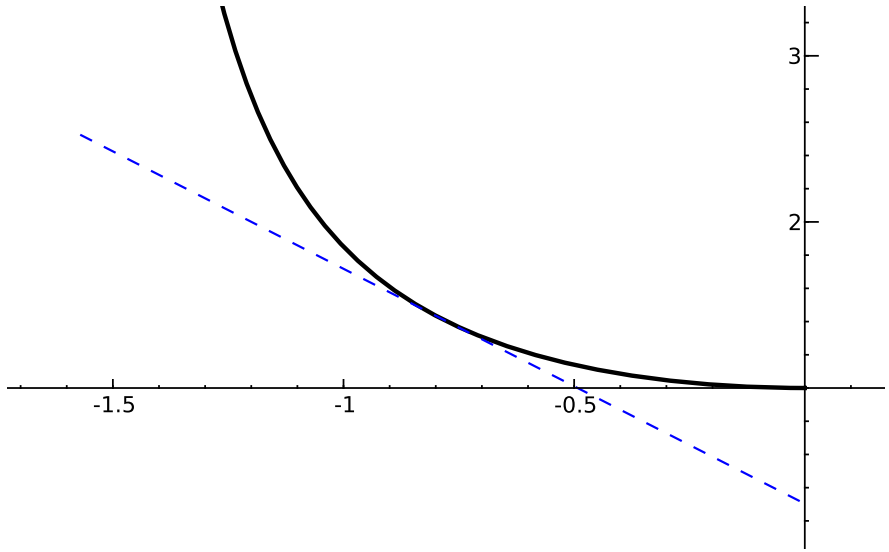
```
sage:  def plot_function_and_tangent...
sage:  plot_function_and_tangent(sec(x),-pi/2,
         xmin=-pi/2,xmax=0)
```



ouch. need to adjust `ymax`

Examples

```
sage:  def plot_function_and_tangent...
sage:  good_sec = plot_function_and_tangent(
         sec(x),
         xmin=-pi/2,x0=-pi/4,xmax=0)
sage:  show(good_sec,ymax=3)
```



Note: `sec(x)` *does not work in older versions, apparently because its derivative is not computed*

Procedures

Arguments to procedures

Extended example

Returning values

Continuing the example

Pseudocode

Scripting

Summary

Outline

1 Procedures

2 Arguments to procedures

3 Extended example

4 Returning values

5 Continuing the example

6 Pseudocode

7 Scripting

8 Summary

Scripting?

script a sequence of Sage statements saved to a file

- write, save
- load at later date
- no need to type, re-type statements

Procedures

Arguments to procedures

Extended example

Returning values

Continuing the example

Pseudocode

Scripting

Summary

Scripting?

script a sequence of Sage statements saved to a file

- write, save
- load at later date
- no need to type, re-type statements

Let's script our `tangent_line()` procedure.

Creating a script

- Back to 'Files'
- Type `calc_utils.sage` in "Filename" box
- Choose "Sage Code (.sage)" in "Create" drop-down
- put `tangent_line()` code in new file
- Save, close
  - `calc_utils.sage` appears in list of files!

Using a script

- Create new Sage worksheet
  - or open old one
- "Attach" the script:

```
sage:  attach calc_utils.sage
```

- Try it:

```
sage:  tangent_line(x^2, 1)
2*x - 1
```

Using a script

- Create new Sage worksheet
  - or open old one
- "Attach" the script:

```
sage:  attach calc_utils.sage
```

- Try it:

```
sage:  tangent_line(x^2, 1)
2*x - 1
```

- You have now "extended" Sage!
- Can use `tangent_line()` in any worksheet, or even in other scripts, using `%attach` directive

Procedures

Arguments to procedures

Extended example

Returning values

Continuing the example

Pseudocode

Scripting

Summary

Outline

1 Procedures

2 Arguments to procedures

3 Extended example

4 Returning values

5 Continuing the example

6 Pseudocode

7 Scripting

8 Summary

Summary

- procedures collect several Sage statements into one
  - organizes solutions to problems
  - abstraction makes problem-solving easier
- define using `def(...):`
- procedures receive *arguments* as data
  - can specify default values
  - procedure does not change arguments, *but*...
  - elements of collections can be changed
- Return value(s) using `return`
- Scripts save procedures for later usage