

MAT 305: Mathematical Computing

Functions in computer programming

John Perry

University of Southern Mississippi

Spring 2016

Outline

- 1 Functions
- 2 Functions and arguments
- 3 Returning values
- 4 Summary

Functions

Functions and
arguments

Returning
values

Summary

Outline

- 1 Functions
- 2 Functions and arguments
- 3 Returning values
- 4 Summary

Functions?

Functions

Functions and arguments

Returning values

Summary

- **function:** a sequence of statements organized as one command
 - may return one or more values
- names in other languages
 - C family: “functions”
 - Pascal family: “procedures” (no result) or “functions” (result)
 - object-oriented languages: “methods” or “features”

Why functions?

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

Defining a function

```
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

Defining a function

```
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 function*

Calling a function

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

Example

```
def main():  
    print 'Hello, world'
```

- name of function is main
- no arguments
- one statement

terrible choice; do not use

Example

Try it!

```
sage: def main():  
        print 'Hello, world'  
sage: main()  
Hello, world
```

Outline

- 1 Functions
- 2 Functions and arguments
- 3 Returning values
- 4 Summary

Arguments?

argument: a placeholder for data

scope: name visible only inside function where it is defined

- data still exists outside function
- 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 function concludes

Example

```
def hello(name='world'):  
    print 'Hello,', name
```

- name of function is hello
- one argument, name
 - default value: 'world'

Example

Try it!

```
sage: def hello(name='world'):  
      print 'Hello,', name
```

```
sage: hello()  
Hello, world
```

Example

Try it!

```
sage: def hello(name='world'):  
      print 'Hello,', name
```

```
sage: hello()  
Hello, world
```

```
sage: hello('Pythagoras')  
Hello, Pythagoras
```

Example

Try it!

```
sage: def hello(name='world'):  
      print 'Hello,', name
```

```
sage: hello()  
Hello, world
```

```
sage: hello('Pythagoras')  
Hello, Pythagoras
```

```
sage: hello(pi)  
Hello, pi
```


Warning 1

Don't use uninitialized identifiers

```
sage: def hello(name='world'):  
      print 'Hello,', name
```

```
sage: hello(Pythagoras)
```

oops: no quotes!

...Output deleted...

```
NameError: name 'Pythagoras' is not defined
```

Warning 2

Functions

Functions and
arguments

Returning
values

Summary

Scope implies name does not exist outside hello

```
sage: def hello(name='world'):  
      print 'Hello,', name
```

```
sage: hello('Pythagoras')  
Hello, Pythagoras
```

```
sage: name  
'KodairaSymbol'
```

???

Warning 3

Functions

Functions and
arguments

Returning
values

Summary

Scope implies name forgotten once hello concludes

```
sage: def hello(name='world'):  
      print 'Hello,', name
```

```
sage: hello('Pythagoras')  
Hello, Pythagoras
```

```
sage: hello()  
Hello, world
```

name has value 'world' again

Warning 4

Functions

Functions and
arguments

Returning
values

Summary

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

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

Warning 4

Functions

Functions and
arguments

Returning
values

Summary

Can change value inside function, but value outside function 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 function
```

Warning 4

Functions

Functions and
arguments

Returning
values

Summary

Can change value inside function, but value outside function 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 function

```
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 hello(name='world'):  
      print 'Hello,', name
```

```
sage: def goodbye(name):  
      print 'Goodbye,', name
```

no default for name

Warning 5

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

```
sage: def hello(name='world'):  
      print 'Hello,', name
```

```
sage: def goodbye(name):  
      print 'Goodbye,', name
```

no default for name

```
sage: hello()  
Hello, world
```


Warning 5

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

```
sage: def hello(name='world'):  
      print 'Hello,', name
```

```
sage: def goodbye(name):  
      print 'Goodbye,', name
```

no default for name

```
sage: hello()  
Hello, world
```

```
sage: goodbye()  
...Output deleted...
```

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

Warning 5

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

```
sage: def hello(name='world'):  
      print 'Hello,', name
```

```
sage: def goodbye(name):  
      print 'Goodbye,', name
```

no default for name

```
sage: hello()  
Hello, world
```

```
sage: goodbye()  
...Output deleted...
```

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

```
sage: goodbye('cruel world')  
Goodbye, cruel world
```

Arguments, lists and sets

- Function does not change the value of an argument outside function
- However, if argument is a mutable collection C :
 - C cannot be changed, but
 - *elements* of C can be changed

Example: C does not change

Functions

Functions and
arguments

Returning
values

Summary

```
sage: def modify_C(C):  
      C = [0,1,2,3]
```

```
sage: L = [-1,0,1]
```

```
sage: modify_C(L)
```

```
sage: L  
[-1, 0, 1]
```

Example: elements of C change

Functions

Functions and
arguments

Returning
values

Summary

```
sage: def modify_els_of_C(C):  
      C[0] = 0
```

```
sage: L = [-1,0,1]
```

```
sage: modify_els_of_C(L)
```

```
sage: L  
[0, 0, 1]
```

Why does this happen?

Hand-waving / Lawyer's argument

- L is a list of 3 elements
 - data does not change
 - function concludes: L is still a list of 3 elements
- $L[0]$, $L[1]$, $L[2]$ are *elements* of L
 - these data **are not** “arguments” to function
 - \therefore can be changed

Why does this happen?

Analogy: defacing library books doesn't change catalog

- L is address of a location in memory
 - similar to library's reference number for book

Why does this happen?

Analogy: defacing library books doesn't change catalog

- L is address of a location in memory
 - similar to library's reference number for book
- Python *copies* L's value
 - write reference number on a scrap sheet of paper
 - original reference still in catalog

Why does this happen?

Analogy: defacing library books doesn't change catalog

- L is address of a location in memory
 - similar to library's reference number for book
- Python *copies* L's value
 - write reference number on a scrap sheet of paper
 - original reference still in catalog
- Function learns (and cannot change) L's value, *but...*
 - can deface book at that location, *even though*
 - changing number on scrap sheet of paper (C) doesn't change catalog entry (L)
 - \therefore function can change information at location

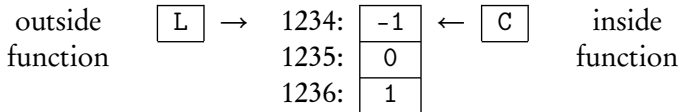
Why does this happen?

Analogy: defacing library books doesn't change catalog

- L is address of a location in memory
 - similar to library's reference number for book
- Python *copies* L's value
 - write reference number on a scrap sheet of paper
 - original reference still in catalog
- Function learns (and cannot change) L's value, *but...*
 - can deface book at that location, *even though*
 - changing number on scrap sheet of paper (C) doesn't change catalog entry (L)
 - \therefore function can change information at location
- Function concludes: data changed but L unchanged
 - books defaced, but catalog still references them

Why does this happen?

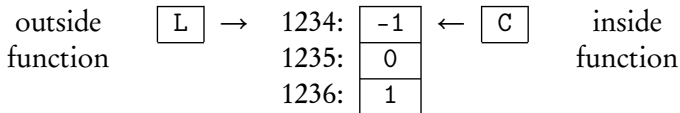
Precise answer: lists are pointers



- List @ location 1234 \implies L \longrightarrow 1234

Why does this happen?

Precise answer: lists are pointers



- List @ location 1234 $\implies L \longrightarrow 1234$
- $\therefore C \longrightarrow 1234$
- Function now has access to memory *at* L
 - changing C won't change L
 - changing C[0] changes L[0]

Functions

Functions and
arguments

Returning
values

Summary

Outline

① Functions

② Functions and arguments

③ Returning values

④ Summary

Returning values?

- Functions compute values
- Often want to work with what we've computed
 - `print` command not very helpful

Example

Compute derivative of special function, want to use it to:

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

The return command

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

- reports the data values *value1*, *value2*, etc.
- only works inside functions
- only reports to caller of current function

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

Pseudocode?

description of algorithm

- many formats
- format independent of computer language
- prefer mathematics to programming
 - “ i th 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 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 $x = x_0$

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*

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*

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

```
def tangent_line(f, x0, x=x):  
    # returns the line tangent to f at x=x0  
    y0 = f({x:x0}) # needs dictionary substitution  
    df = diff(f,x)  
    m = df({x:x0})  
    tanline = m*(x - x0) + y0  
    return tanline
```

Functions

Functions and
arguments

Returning
values

Summary

Example implementation

```
def tangent_line(f, x0, x=x):  
    # returns the line tangent to f at x=x0  
    y0 = f({x:x0}) # needs dictionary substitution  
    df = diff(f,x)  
    m = df({x:x0})  
    tanline = m*(x - x0) + y0  
    return tanline
```

Why have x as an input? Many reasons:

- $f(t)$
- other variables in function

so let's specify variable as well

Notice line that begins with # (--- in psuedocode)

- Sage *ignores* anything after this symbol
- Use to explain intent to reader
 - you, too, are reader!

Example run

```
sage: def tangent_line(f, x0, x=x):
      # returns the line tangent to f at x=x0
      y0 = f({x:x0})
      df = diff(f,x)
      m = df({x:x0})
      tanline = m*(x - x0) + y0
      return tanline

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

Example run

```
sage: def tangent_line(f, x0, x=x):  
      # returns the line tangent to f at x=x0  
      y0 = f({x:x0})  
      df = diff(f,x)  
      m = df({x:x0})  
      tanline = m*(x - x0) + y0  
      return tanline
```

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

Behold: the power of symbolic computation!

```
sage: var('a b c')
```

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

Combine with plots

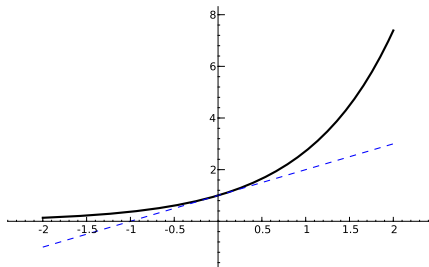
We show the plots of e^x and its tangent line at $x = 0$

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



Combining functions

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

Combining functions

It would be nice to have a function 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 functions

It would be nice to have a function 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 function

Pseudocode

Functions

Functions and
arguments

Returning
values

Summary

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, x, x0=0,  
                             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, x, x0)
```

tasks

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

Examples

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

Functions

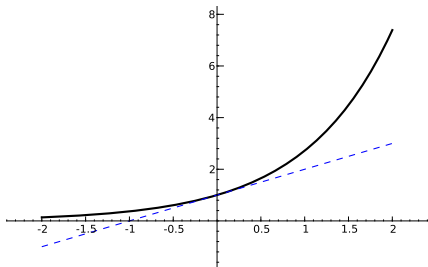
Functions and
arguments

Returning
values

Summary

Examples

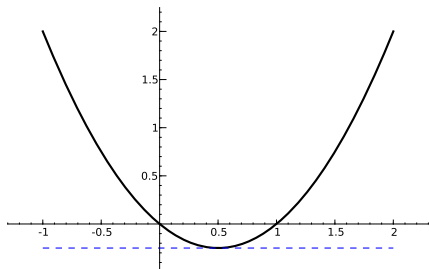
```
sage: def plot_function_and_tangent...  
sage: plot_function_and_tangent(e**x,x)
```



Examples

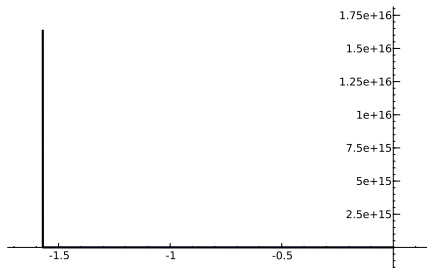
```
sage: def plot_function_and_tangent...
```

```
sage: plot_function_and_tangent(x**2-x,x,xmin=-1,  
                                x0=0.5,xmax=2)
```



Examples

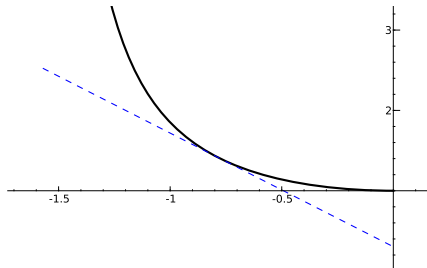
```
sage: def plot_function_and_tangent...  
sage: plot_function_and_tangent(sec(x),x,  
      xmin=-pi/2,x0=-pi/4,xmax=0)
```



ouch. need to adjust ymax

Examples

```
sage: def plot_function_and_tangent...
sage: good_sec = plot_function_and_tangent(
        sec(x), 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

Functions

Functions and
arguments

Returning
values

Summary

Outline

- 1 Functions
- 2 Functions and arguments
- 3 Returning values
- 4 Summary

Summary

- Functions collect several commands into one
 - organizes solutions to problems
 - abstraction makes problem-solving easier
- define using `def (...):`
- Functions receive *arguments* as data
 - can specify default values
 - function does not change arguments, *but...*
 - elements of collections can be changed
- Return value(s) using `return`