DM536
Introduction to Programming

Peter Schneider-Kamp

petersk@imada.sdu.dk

http://imada.sdu.dk/~petersk/DM536/

DEFINING FUNCTIONS

Function Definitions

- functions are defined using the following grammar rule:

  <func.def> => def <function>(<arg$_1$>, …, <arg$_n$>):
  <instr$_1$>; …; <instr$_k$>

- can be used to reuse code:
  - Example:

```
def pythagoras():
    c = math.sqrt(a**2+b**2)
    print "Result:", c
a = 3; b = 4; pythagoras()
a = 7; b = 15; pythagoras()
```

- functions are values: `type(pythagoras)`

Functions Calling Functions

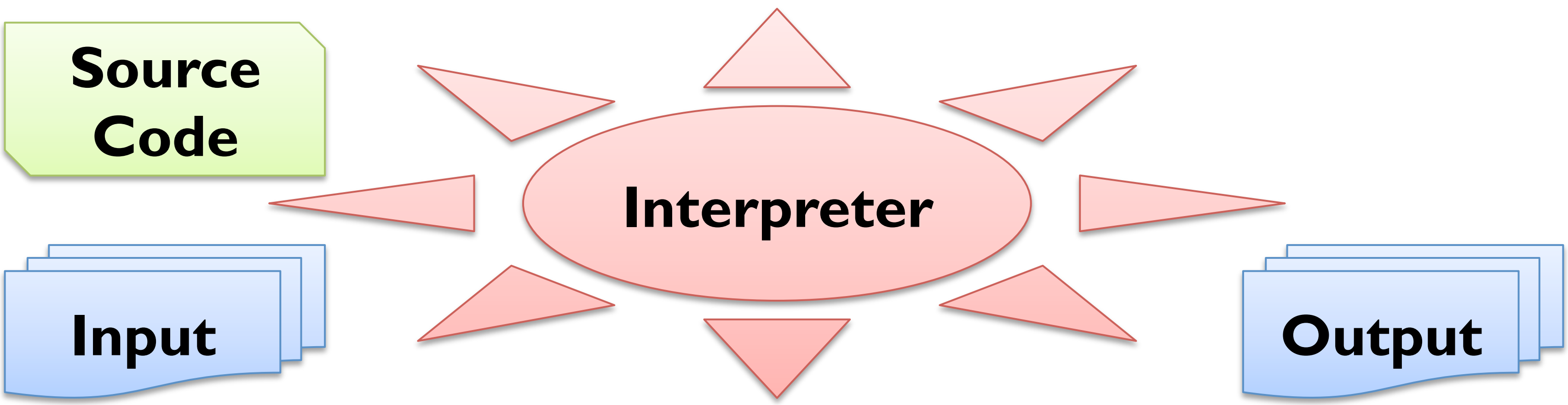
- functions can call other functions

- Example:

```
def white():
    print " #" * 8
def black():
    print "# " * 8
def all():
    white();  black();  white();  black()
    white();  black();  white();  black()
all()
```

Executing Programs (Revisited)

- Program stored in a file (*source code* file)
- Instructions in this file executed top-to-bottom
- Interpreter executes each instruction



Functions Calling Functions

- functions can call other functions

- Example:

```
def white():
    print " #" * 8
def black():
    print "# " * 8
def all():
    white();  black();  white();  black()
    white();  black();  white();  black()
all()
```

create new function variable "white"

Functions Calling Functions

- functions can call other functions

- Example:

```
def white():
    print " #" * 8
def black():
    print "# " * 8
def all():
    white(); black(); white(); black()
    white(); black(); white(); black()
all()
```

create new function variable "black"

Functions Calling Functions

- functions can call other functions

- Example:

```
def white():
    print " #" * 8
def black():
    print "# " * 8
def all():
    white();  black();  white();  black()
    white();  black();  white();  black()
all()
```

create new function variable “all”

Functions Calling Functions

- functions can call other functions
- Example:

```
def white():
    print " #" * 8
def black():
    print "# " * 8
def all():
    white(); black(); white(); black()
    white(); black(); white(); black()
all()
```

call function “all”

Functions Calling Functions

- functions can call other functions

- Example:

```
def white():
    print " #" * 8
def black():
    print "# " * 8
def all():
    white(); black(); white(); black()
    white(); black(); white(); black()
all()
```

call function "white"

Functions Calling Functions

- functions can call other functions

- Example:

```
def white():
    print " #" * 8
def black():
    print "# " * 8
def all():
    white();  black();  white();  black()
    white();  black();  white();  black()
all()
```

print
" # # # # # # # #"

Functions Calling Functions

- functions can call other functions

- Example:

```
def white():
    print " #" * 8
def black():
    print "# " * 8
def all():
    white(); black(); white(); black()
    white(); black(); white(); black()
all()
```

call function “black”

Functions Calling Functions

- functions can call other functions

- Example:

```
def white():
    print " #" * 8
def black():
    print "# " * 8
def all():
    white(); black(); white(); black()
    white(); black(); white(); black()
all()
```

print
"# # # # # # # # "

Functions Calling Functions

- functions can call other functions
- Example:

```
def white():
    print " #" * 8
def black():
    print "# " * 8
def all():
    white();  black();  white();  black()
    white();  black();  white();  black()
all()
```

call function "white"

Functions Calling Functions

- functions can call other functions

- Example:

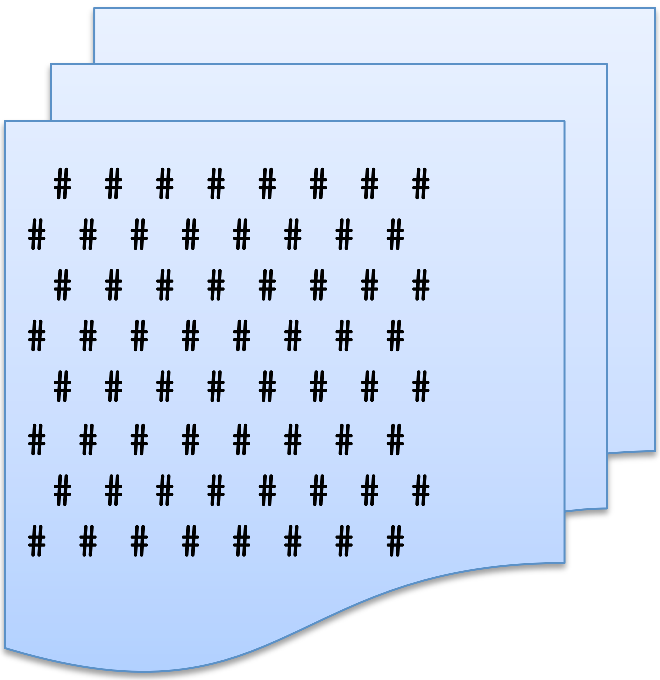
```
def white():
    print " #" * 8
def black():
    print "# " * 8
def all():
    white();  black();  white();  black()
    white();  black();  white();  black()
all()
```

print
" # # # # # # # #"

Functions Calling Functions

- functions can call other functions

- Example:




```
def white():
    print " #" * 8
def black():
    print "# " * 8
def all():
    white(); black(); white(); black()
    white(); black(); white(); black()
all()
```

Parameters and Arguments

- we have seen functions that need arguments:
  - math.sqrt(x) computes square root of x
  - math.log(x, base) computes logarithm of x w.r.t. base

- arguments are assigned to parameters of the function
  - Example:

```
def pythagoras():
    c = math.sqrt(a**2+b**2)
    print "Result:", c
a = 3; b = 4; pythagoras()
a = 7; b = 15; pythagoras()
```

Parameters and Arguments

- we have seen functions that need arguments:
  - math.sqrt(x) computes square root of x
  - math.log(x, base) computes logarithm of x w.r.t. base

- arguments are assigned to parameters of the function
  - Example:

```
def pythagoras(a, b):
    c = math.sqrt(a**2+b**2)
    print "Result:", c
a = 3; b = 4; pythagoras(a, b)
a = 7; b = 15; pythagoras(a, b)
```

Parameters and Arguments

- we have seen functions that need arguments:
  - math.sqrt(x) computes square root of x
  - math.log(x, base) computes logarithm of x w.r.t. base

- arguments are assigned to parameters of the function
  - Example:

```
def pythagoras(a, b):
    c = math.sqrt(a**2+b**2)
    print "Result:", c
pythagoras(3, 4)
pythagoras(7, 15)
```

Parameters and Arguments

- we have seen functions that need arguments:
  - math.sqrt(x) computes square root of x
  - math.log(x, base) computes logarithm of x w.r.t. base

- arguments are assigned to parameters of the function
  - Example:

```
def pythagoras(a, b):
    c = math.sqrt(a**2+b**2)
    print "Result:", c
pythagoras(3, 4)
pythagoras(2**3-1, 2**4-1)
```

Parameters and Arguments

- we have seen functions that need arguments:
  - math.sqrt(x) computes square root of x
  - math.log(x, base) computes logarithm of x w.r.t. base

- arguments are assigned to parameters of the function
  - Example:

```
def pythagoras(a, b):
    c = math.sqrt(a**2+b**2)
    print "Result:", c
pythagoras(3, 4)
x = 2**3-1; y = 2**4-1
pythagoras(x, y)
```

Variables are Local

- parameters and variables are local
- local = only available in the function defining them
- Example:

in module math:

```
def sqrt(x):
    …
```

in our program:

```
def pythagoras(a, b):
    c = math.sqrt(a**2+b**2)
    print "Result:", c
x = 3;  y =4;  pythagoras(x, y)
```

x local to math.sqrt

a local to pythagoras

b local to pythagoras

c local to pythagoras

x,y local to __main__

Stack Diagrams

__main__	x	→	3
	y	→	4
pythagoras	a	→	3
	b	→	4
math.sqrt	x	→	25

Tracebacks

- stack structure printed on runtime error
- Example:

```
def broken(x):
    print x / 0
def caller(a, b):
    broken(a**b)
caller(2,5)
```

```
Traceback (most recent call last):
  File "test.py", line 5, in <module>
    caller(2,5)
  File "test.py", line 4, in caller
    broken(a**b)
  File "test.py", line 2, in broken
    print x/0
ZeroDivisionError: integer division or modulo by zero
```

Return Values

- we have seen functions that return values:
  - math.sqrt(x) returns the square root of x
  - math.log(x, base) returns the logarithm of x w.r.t. base

- What is the return value of our function pythagoras(a, b)?
- special value None returned, if no return value given (*void*)

- declare return value using return statement: return <expr>
- Example:

```
def pythagoras(a, b):
    c = math.sqrt(a**2+b**2)
    return c
print pythagoras(3, 4)
```

Motivation for Functions

- functions give names to blocks of code
  - easier to read
  - easier to debug
- avoid repetition
  - easier to make changes
- functions can be debugged separately
  - easier to test
  - easier to find errors
- functions can be reused (for other programs)
  - easier to write new programs

Debugging Function Definitions

- make sure you are using latest files (save, then run python -i)

- biggest problem for beginners is *indentation*
  - all lines on the same level must have the same indentation
  - mixing spaces and tabs is very dangerous
  - try to use only spaces – a good editor helps!

- do not forget to use ":" at end of first line
- indent body of function definition by e.g. 4 spaces

TURTLE WORLD & INTERFACE DESIGN

Turtle World

- available from
  - http://www.greenteapress.com/thinkpython/swampy/install.html
- basic elements of the library
  - can be imported using from TurtleWorld import *
  - w = TurtleWorld() creates new world w
  - t = Turtle() creates new turtle t
  - wait_for_user() can be used at the end of the program

Simple Repetition

- two basic commands to the turtle
  - fd(t, 100) advances turtle t by 100
  - lt(t) turns turtle t 90 degrees to the left
- drawing a square requires 4x drawing a line and turning left
  - fd(t,100); lt(t); fd(t,100); lt(t); fd(t,100); lt(t); fd(t,100); lt(t)
- simple repetition using for-loop

```
for <var> in range(<expr>):
    <instr1>; <instr2>
```

- Example:

```
for i in range(4):
    print i
```

Simple Repetition

- two basic commands to the turtle
  - fd(t, 100) advances turtle t by 100
  - lt(t) turns turtle t 90 degrees to the left
- drawing a square requires 4x drawing a line and turning left
  - fd(t,100); lt(t); fd(t,100); lt(t); fd(t,100); lt(t); fd(t,100); lt(t)
- simple repetition using for-loop

```
for <var> in range(<expr>):
    <instr_1>; <instr_2>
```

- Example:

```
for i in range(4):
    fd(t, 100)
    lt(t)
```

Encapsulation

- **Idea:** wrap up a block of code in a function
  - documents use of this block of code
  - allows reuse of code by using parameters

- Example:

```
def square(t):
    for i in range(4):
        fd(t, 100)
        lt(t)
square(t)
u = Turtle(); rt(u); fd(u,10); lt(u);
square(u)
```

Generalization

- square(t) can be reused, but size of square is fixed
- **Idea:** generalize function by adding parameters
  - more flexible functionality
  - more possibilities for reuse
- Example 1:

```
def square(t, length):
    for i in range(4):
        fd(t, length)
        lt(t)
square(t, 100)
square(t, 50)
```

Generalization

- Example 2: replace square by regular polygon with n sides

```
def square(t, length):
    for i in range(4):
        fd(t, length)
        lt(t)
```

Generalization

- Example 2: replace square by regular polygon with n sides

```
def polygon(t, length):
    for i in range(4):
        fd(t, length)
        lt(t)
```

Generalization

- Example 2: replace square by regular polygon with n sides

```
def polygon(t, n, length):
    for i in range(n):
        fd(t, length)
        lt(t)
```

Generalization

- Example 2: replace square by regular polygon with n sides

```
def polygon(t, n, length):
    for i in range(n):
        fd(t, length)
        lt(t, 360/n)
```

Generalization

- Example 2: replace square by regular polygon with n sides

```
def polygon(t, n, length):
    angle = 360/n
    for i in range(n):
        fd(t, length)
        lt(t, angle)
```

Generalization

- Example 2: replace square by regular polygon with n sides

```
def polygon(t, n, length):
    angle = 360/n
    for i in range(n):
        fd(t, length)
        lt(t, angle)
polygon(t, 4, 100)
polygon(t, 6, 50)
```

Generalization

- Example 2: replace square by regular polygon with n sides

```
def polygon(t, n, length):
    angle = 360/n
    for i in range(n):
        fd(t, length)
        lt(t, angle)
polygon(t, n=4, length=100)
polygon(t, n=6, length=50)
```

Generalization

- Example 2: replace square by regular polygon with n sides

```
def polygon(t, n, length):
    angle = 360/n
    for i in range(n):
        fd(t, length)
        lt(t, angle)


square(t, 100)
```

Generalization

- Example 2: replace square by regular polygon with n sides

```
def polygon(t, n, length):
    angle = 360/n
    for i in range(n):
        fd(t, length)
        lt(t, angle)
def square(t, length):
    polygon(t, 4, length)
square(t, 100)
```

Interface Design

- **Idea:** interface = parameters + semantics + return value
- should be general (= easy to reuse)
- but as simple as possible (= easy to read and debug)

- Example:

```
def circle(t, r):
    circumference = 2*math.pi*r
    n = 10
    length = circumference / n
    polygon(t, n, length)
circle(t, 10)
circle(t, 100)
```

Interface Design

- **Idea:** interface = parameters + semantics + return value
- should be general (= easy to reuse)
- but as simple as possible (= easy to read and debug)

- Example:

```
def circle(t, r, n):
    circumference = 2*math.pi*r
#   n = 10
    length = circumference / n
    polygon(t, n, length)
circle(t, 10, 10)
circle(t, 100, 40)
```

Interface Design

- **Idea:** interface = parameters + semantics + return value
- should be general (= easy to reuse)
- but as simple as possible (= easy to read and debug)

- Example:

```
def circle(t, r):
    circumference = 2*math.pi*r
    n = int(circumference / 3) + 1
    length = circumference / n
    polygon(t, n, length)
circle(t, 10)
circle(t, 100)
```

Refactoring

- we want to be able to draw arcs
- Example:

```
def arc(t, r, angle):
    arc_length = 2*math.pi*r*angle/360
    n = int(arc_length / 3) + 1
    step_length = arc_length / n
    step_angle = float(angle) / n


    for i in range(n):
        fd(t, step_length)
        lt(t, step_angle)
```

Refactoring

- we want to be able to draw arcs
- Example:

```
def arc(t, r, angle):
    arc_length = 2*math.pi*r*angle/360
    n = int(arc_length / 3) + 1
    step_length = arc_length / n
    step_angle = float(angle) / n

def polyline(t, n, length, angle):
    for i in range(n):
        fd(t, length)
        lt(t, angle)
```

Refactoring

- we want to be able to draw arcs
- Example:

```
def arc(t, r, angle):
    arc_length = 2*math.pi*r*angle/360
    n = int(arc_length / 3) + 1
    step_length = arc_length / n
    step_angle = float(angle) / n
    polyline(t, n, step_length, step_angle)
def polyline(t, n, length, angle):
    for i in range(n):
        fd(t, length)
        lt(t, angle)
```

Refactoring

- we want to be able to draw arcs
- Example:

```
def polyline(t, n, length, angle):
    for i in range(n):
        fd(t, length)
        lt(t, angle)
```

Refactoring

- we want to be able to draw arcs
- Example:

```
def polyline(t, n, length, angle):
    for i in range(n):
        fd(t, length)
        lt(t, angle)
def polygon(t, n, length):
    angle = 360/n
    polyline(t, n, length, angle):
```

Refactoring

- we want to be able to draw arcs
- Example:

```
def arc(t, r, angle):
    arc_length = 2*math.pi*r*angle/360
    n = int(arc_length / 3) + 1
    step_length = arc_length / n
    step_angle = float(angle) / n
    polyline(t, n, step_length, step_angle)
```

Refactoring

- we want to be able to draw arcs
- Example:

```
def arc(t, r, angle):
    arc_length = 2*math.pi*r*angle/360
    n = int(arc_length / 3) + 1
    step_length = arc_length / n
    step_angle = float(angle) / n
    polyline(t, n, step_length, step_angle)
def circle(t, r):
    arc(t, r, 360)
```

Simple Iterative Development

- first structured approach to develop programs:
  1. write small program without functions
  2. encapsulate code in functions
  3. generalize functions (by adding parameters)
  4. repeat steps 1–3 until functions work
  5. refactor program (e.g. by finding similar code)

- copy & paste helpful
  - reduces amount of typing
  - no need to debug same code twice

Debugging Interfaces

- interfaces simplify testing and debugging

1. test if pre-conditions are given:
   - do the arguments have the right type?
   - are the values of the arguments ok?
2. test if the post-conditions are given:
   - does the return value have the right type?
   - is the return value computed correctly?
3. debug function, if pre- or post-conditions violated

CONDITIONAL EXECUTION

Boolean Expressions

- expressions whose value is either True or False

- logic operators for computing with Boolean values:
  - x and y — True if, and only if, x is True and y is True
  - x or y — True if (x is True or y is True)
  - not x — True if, and only if, x is False

- Python also treats numbers as Boolean expressions:
  - 0 — False
  - any other number — True
  - Please, do NOT use this feature!

Relational Operators

- relational operators are operators, whose value is Boolean

- important relational operators are:

	Example True	Example False
x < y	23 < 42	"World" < "Hej!"
x <= y	42 <= 42.0	int(math.pi) <= 2
x == y	42 == 42.0	type(2) == type(2.0)
x >= y	42 >= 42	"Hej!" >= "Hello"
x > y	"World" > "Hej!"	42 > 42

- remember to use "==" instead of "=" (assignment)!

Conditional Execution

- the if-then statement executes code only if a condition holds
- grammar rule:

  <if-then> => if <cond>:

  <instr$_1$>; …; <instr$_k$>

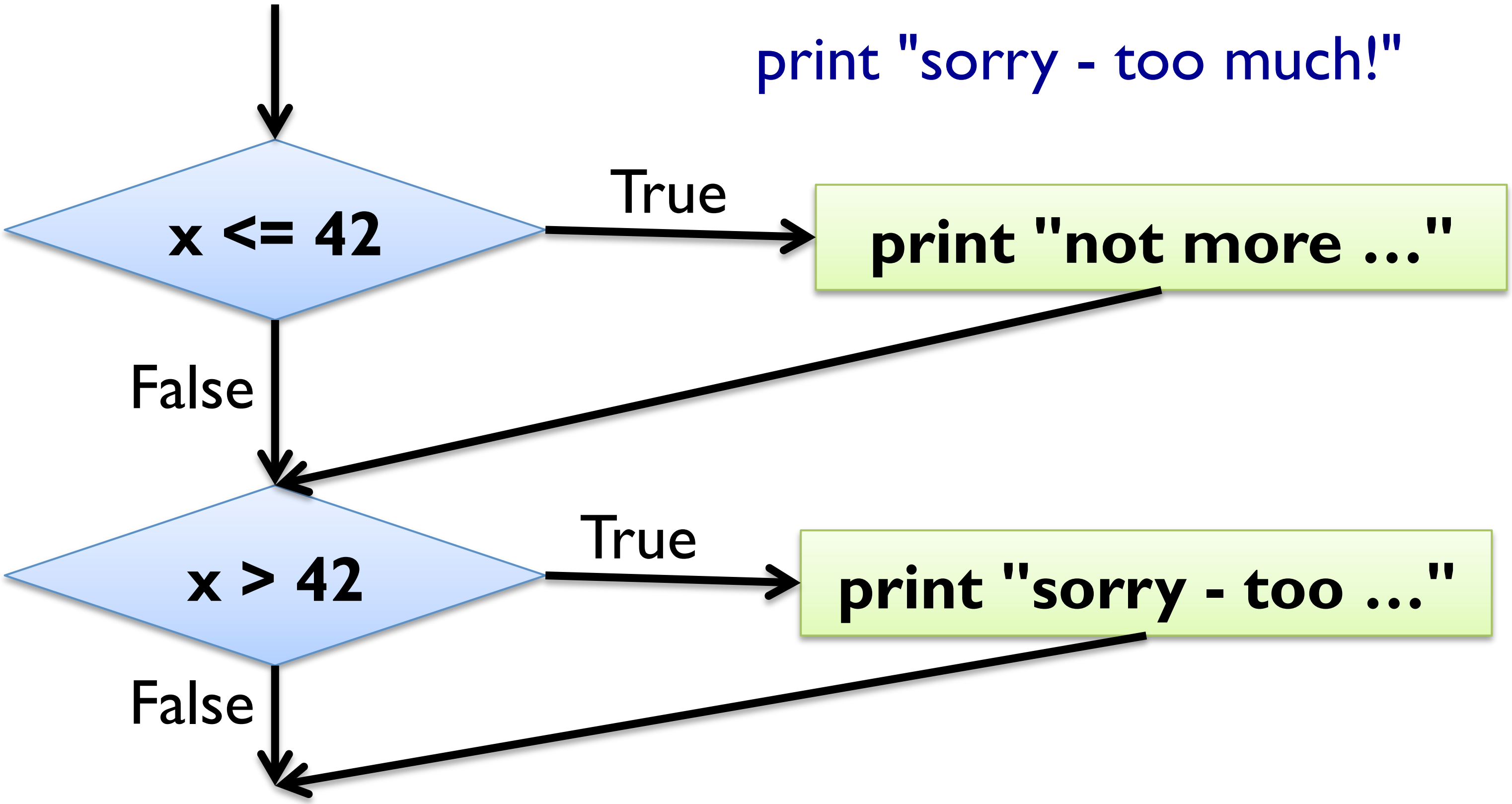
- Example:

```
if x <= 42:
    print "not more than the answer"
if x  >  42:
    print "sorry - too much!"
```

Control Flow Graph

- Example:

```
if x <= 42:
    print "not more than the answer"
if x  >  42:
    print "sorry - too much!"
```



Alternative Execution

- the if-then-else statement executes one of two code blocks
- grammar rule:

<if-then-else> => if <cond>:

    <instr$_1$>; …; <instr$_k$>

else:

    <instr'$_1$>; …; <instr'$_{k'}$>

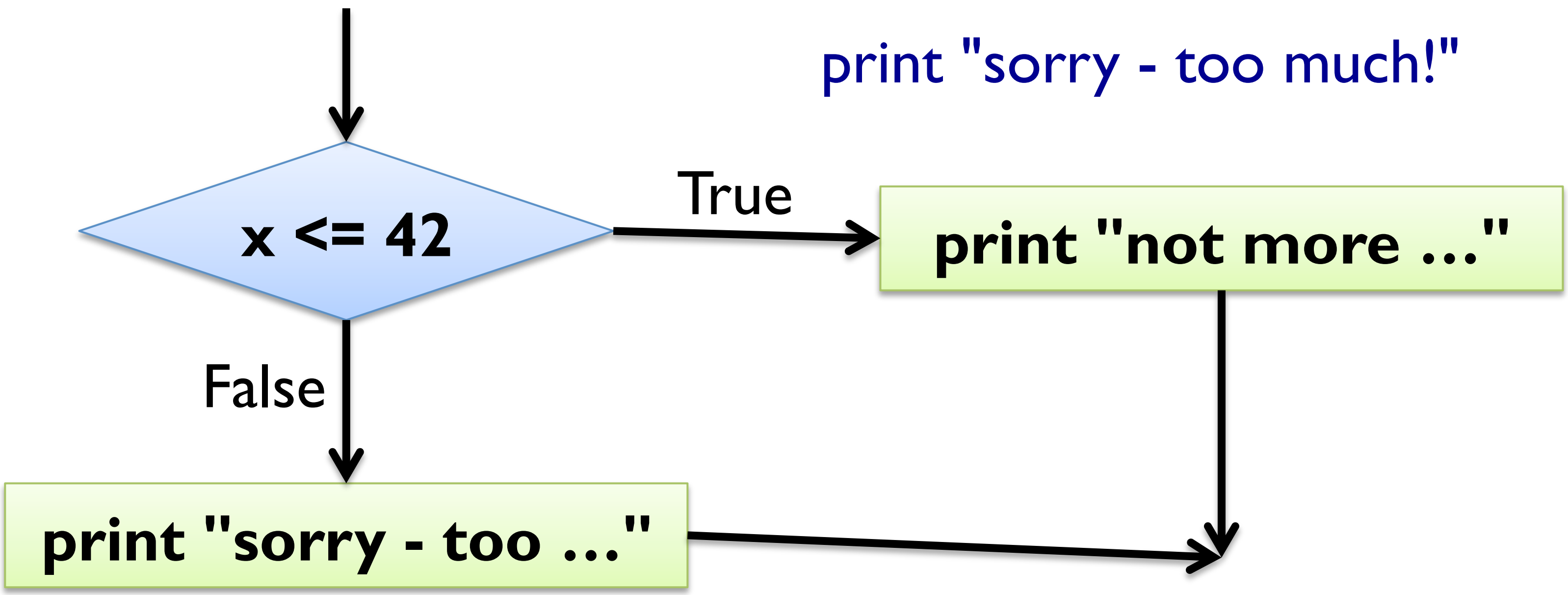
- Example:

```
if x <= 42:
    print "not more than the answer"
else:
    print "sorry - too much!"
```

Control Flow Graph

- Example:

```
if x <= 42:
    print "not more than the answer"
else:
    print "sorry - too much!"
```



Chained Conditionals

- alternative execution a special case of chained conditionals
- grammar rules:

<if-chained> => if <cond$_1$>:

<instr$_{1,1}$>; …; <instr$_{k1,1}$>

elif <cond$_2$>:

…

else:

<instr$_{1,m}$>; …; <instr$_{km,m}$>

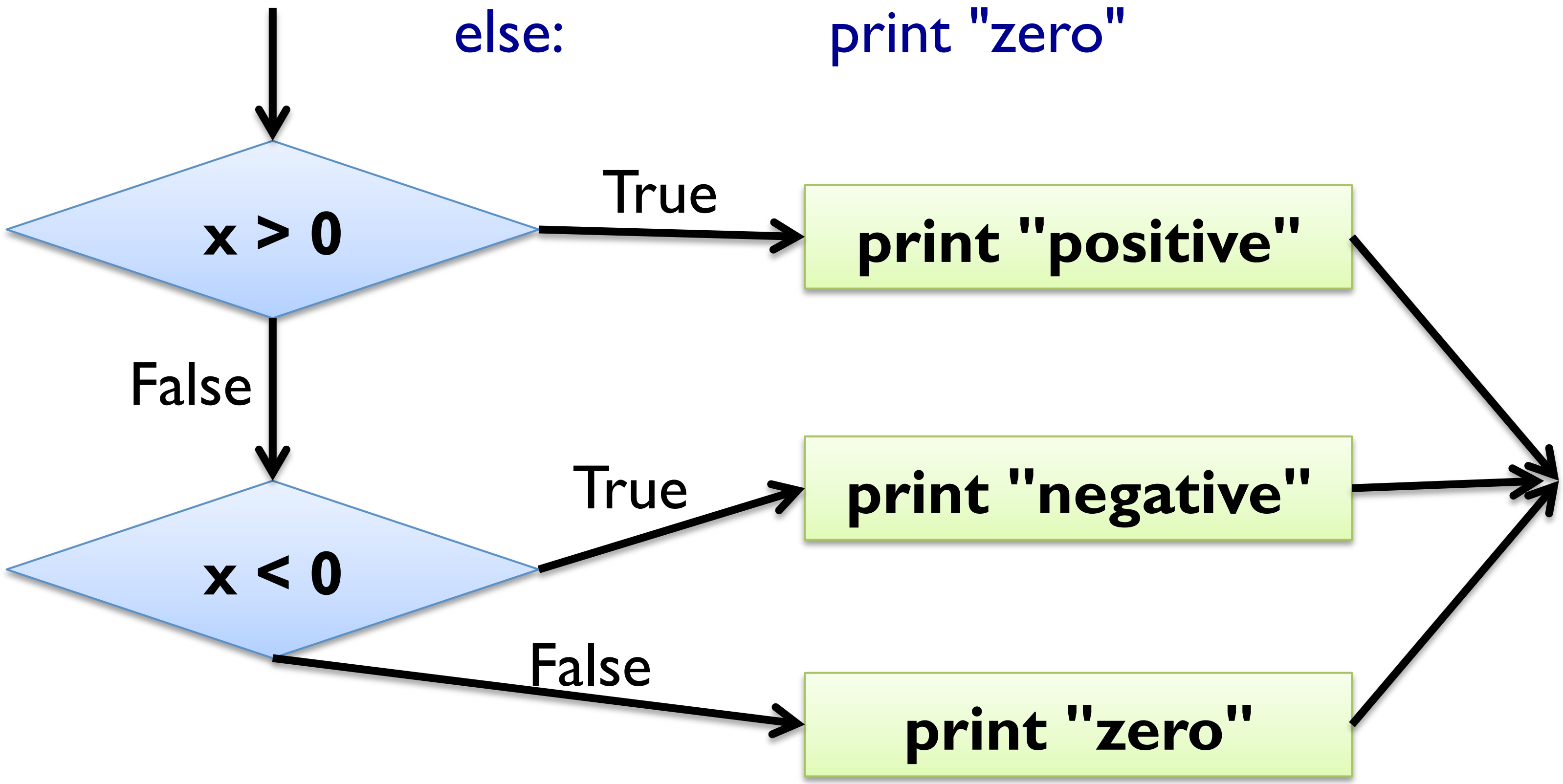
- Example:

```
if x > 0:      print "positive"
elif x < 0:    print "negative"
else:          print "zero"
```

Control Flow Diagram

- Example:

```
if x > 0:      print "positive"
elif x < 0:    print "negative"
else:          print "zero"
```



Nested Conditionals

- conditionals can be nested below conditionals:

```
x = input()
y = input()
if x > 0:
    if y > 0:      print "Quadrant 1"
    elif y < 0:    print "Quadrant 4"
    else:          print "positive x-Axis"
elif x < 0:
    if y > 0:      print "Quadrant 2"
    elif y < 0:    print "Quadrant 3"
    else:          print "negative x-Axis"
else:   print "y-Axis"
```

RECURSION

Recursion

- a function can call other functions
- a function can call **itself**
- such a function is called a *recursive* function

- Example 1:

```
def countdown(n):
    if n <= 0:
        print "Ka-Booooom!"
    else:
        print n, "seconds left!"
        countdown(n-1)
countdown(3)
```

Stack Diagrams for Recursion

__main__	
countdown	**n → 3**
countdown	**n → 2**
countdown	**n → 1**
countdown	**n → 0**

Recursion

- a function can call other functions
- a function can call **itself**
- such a function is called a *recursive* function

- Example 2:

```
def polyline(t, n, length, angle):
    for i in range(n):
        fd(t, length)
        lt(t, angle)
```

Recursion

- a function can call other functions
- a function can call **itself**
- such a function is called a *recursive* function

- Example 2:

```
def polyline(t, n, length, angle):
    if n > 0:
        fd(t, length)
        lt(t, angle)
        polyline(t, n-1, length, angle)
```

Infinite Recursion

- base case = no recursive function call reached
- we say the function call *terminates*
  - Example 1: n == 0 in countdown / polyline

- infinite recursion = no base case is reached
- also called *non-termination*

- Example:

```
def infinitely_often():
    infinitely_often()
```

- Python has *recursion limit* 1000 – ask sys.getrecursionlimit()

Keyboard Input

- so far we only know input()
  - what happens when we enter Hello?
  - input() treats all input as Python expression <expr>
- for string input, use raw_input()
  - what happens when we enter 42?
  - raw_input() treats all input as string
- both functions can take one argument prompt
  - Example 1: a = input("first side: ")
  - Example 2: name = raw_input("Your name:\n")
  - "\n" denotes a new line: print "Hello\nWorld\n!"

Debugging using Tracebacks

- error messages in Python give important information:
  - where did the error occur?
  - what kind of error occurred?

- unfortunately often hard to localize real problem
- Example:

```
def determine_vat(base_price, vat_price):
    factor = base_price / vat_price
    reverse_factor = 1 / factor
    return reverse_factor - 1
print determine_vat(400, 500)
```

real problem (pointing to `factor = base_price / vat_price`)

error reported (pointing to `reverse_factor = 1 / factor`)

Debugging using Tracebacks

- error messages in Python give important information:
  - where did the error occur?
  - what kind of error occurred?

- unfortunately often hard to localize real problem
- Example:

```
def determine_vat(base_price, vat_price):
    factor = float(base_price) / vat_price
    reverse_factor = 1 / factor
    return reverse_factor - 1
print determine_vat(400, 500)
```

FRUITFUL FUNCTIONS

Return Values

- so far we have seen only functions with one or no return
- sometimes more than one return makes sense

- Example 1:

```
def sign(x):
    if x < 0:
        return -1
    elif x == 0:
        return 0
    else:
        return 1
```

Return Values

- so far we have seen only functions with one or no return
- sometimes more than one return makes sense

- Example 1:

```
def sign(x):
    if x < 0:
        return -1
    elif x == 0:
        return 0
    return 1
```

- important that all paths reach one return

Incremental Development

- Idea: test code while writing it
- Example: computing the distance between (x_1,y_1) and (x_2,y_2)

```
def distance(x1, y1, x2, y2):
    print "x1 y1 x2 y2:", x1, y1, x2, y2
```

Incremental Development

- Idea: test code while writing it
- Example: computing the distance between (x_1,y_1) and (x_2,y_2)

```
def distance(x1, y1, x2, y2):
    print "x1 y1 x2 y2:", x1, y1, x2, y2
    dx = x2 - x1          # horizontal distance
    print "dx:", dx
```

Incremental Development

- Idea: test code while writing it
- Example: computing the distance between (x_1,y_1) and (x_2,y_2)

```
def distance(x1, y1, x2, y2):
    print "x1 y1 x2 y2:", x1, y1, x2, y2
    dx = x2 - x1          # horizontal distance
    print "dx:", dx
    dy = y2 - y1          # vertical distance
    print "dy:", dy
```

Incremental Development

- Idea: test code while writing it
- Example: computing the distance between (x_1,y_1) and (x_2,y_2)

```
def distance(x1, y1, x2, y2):
    print "x1 y1 x2 y2:", x1, y1, x2, y2
    dx = x2 - x1            # horizontal distance
    print "dx:", dx
    dy = y2 - y1            # vertical distance
    print "dy:", dy
    dxs = dx**2;  dys = dy**2
    print "dxs dys:", dxs, dys
```

Incremental Development

- Idea: test code while writing it
- Example: computing the distance between (x_1,y_1) and (x_2,y_2)

```
def distance(x1, y1, x2, y2):
    print "x1 y1 x2 y2:", x1, y1, x2, y2
    dx = x2 - x1        # horizontal distance
    dy = y2 - y1        # vertical distance
    dxs = dx**2;  dys = dy**2
    print "dxs dys:", dxs, dys
```

Incremental Development

- Idea: test code while writing it
- Example: computing the distance between (x_1,y_1) and (x_2,y_2)

```
def distance(x1, y1, x2, y2):
    print "x1 y1 x2 y2:", x1, y1, x2, y2
    dx = x2 - x1          # horizontal distance
    dy = y2 - y1          # vertical distance
    dxs = dx**2;  dys = dy**2
    print "dxs dys:", dxs, dys
    ds = dxs + dys        # square of distance
    print "ds:", ds
```

Incremental Development

- Idea: test code while writing it
- Example: computing the distance between (x_1,y_1) and (x_2,y_2)

```
def distance(x1, y1, x2, y2):
    print "x1 y1 x2 y2:", x1, y1, x2, y2
    dx = x2 - x1          # horizontal distance
    dy = y2 - y1          # vertical distance
    dxs = dx**2;  dys = dy**2
    ds = dxs + dys        # square of distance
    print "ds:", ds
```

Incremental Development

- Idea: test code while writing it
- Example: computing the distance between (x_1,y_1) and (x_2,y_2)

```
def distance(x1, y1, x2, y2):
    print "x1 y1 x2 y2:", x1, y1, x2, y2
    dx = x2 - x1          # horizontal distance
    dy = y2 - y1          # vertical distance
    dxs = dx**2;  dys = dy**2
    ds = dxs + dys        # square of distance
    print "ds:", ds
    d = math.sqrt(ds)     # distance
    print d
```

Incremental Development

- Idea: test code while writing it
- Example: computing the distance between (x_1,y_1) and (x_2,y_2)

```
def distance(x1, y1, x2, y2):
    print "x1 y1 x2 y2:", x1, y1, x2, y2
    dx = x2 - x1          # horizontal distance
    dy = y2 - y1          # vertical distance
    dxs = dx**2;  dys = dy**2
    ds = dxs + dys        # square of distance
    d = math.sqrt(ds)     # distance
    print d
```

Incremental Development

- Idea: test code while writing it
- Example: computing the distance between (x_1,y_1) and (x_2,y_2)

```
def distance(x1, y1, x2, y2):
    print "x1 y1 x2 y2:", x1, y1, x2, y2
    dx = x2 - x1          # horizontal distance
    dy = y2 - y1          # vertical distance
    dxs = dx**2;  dys = dy**2
    ds = dxs + dys        # square of distance
    d = math.sqrt(ds)     # distance
    print d
    return d
```

Incremental Development

- Idea: test code while writing it
- Example: computing the distance between (x_1,y_1) and (x_2,y_2)

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1           # horizontal distance
    dy = y2 - y1           # vertical distance
    dxs = dx**2;  dys = dy**2
    ds = dxs + dys         # square of distance
    d = math.sqrt(ds)      # distance
    return d
```

Incremental Development

- Idea: test code while writing it
- Example: computing the distance between (x_1,y_1) and (x_2,y_2)

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1          # horizontal distance
    dy = y2 - y1          # vertical distance
    return math.sqrt(dx**2 + dy**2)
```

Incremental Development

- Idea: test code while writing it

1. start with minimal function
2. add functionality piece by piece
3. use variables for intermediate values
4. print those variables to follow your progress
5. remove unnecessary output when function is finished

Composition

- function calls can be arguments to functions
- direct consequence of arguments being expressions
- Example: area of a circle from center and peripheral point

```
def area(radius):
    return math.pi * radius**2

def area_from_points(xc, yc, xp, yp):
    return area(distance(xc, yc, xp, yp))
```

Boolean Functions

- boolean functions = functions that return True or False
- useful e.g. as <cond> in a conditional execution
- Example:

```
def divides(x, y):
    if y / x * x == y:    # remainder of integer division is 0
        return True
    return False
```

Boolean Functions

- boolean functions = functions that return True or False
- useful e.g. as <cond> in a conditional execution
- Example:

```
def divides(x, y):
    if y % x == 0:        # remainder of integer division is 0
        return True
    return False
```

Boolean Functions

- boolean functions = functions that return True or False
- useful e.g. as <cond> in a conditional execution
- Example:

```
def divides(x, y):
    return y % x == 0
```

Boolean Functions

- boolean functions = functions that return True or False
- useful e.g. as <cond> in a conditional execution
- Example:

```
def divides(x, y):
    return y % x == 0

def even(x):
    return divides(2, x)
```

Boolean Functions

- boolean functions = functions that return True or False
- useful e.g. as <cond> in a conditional execution
- Example:

```
def divides(x, y):
    return y % x == 0

def even(x):
    return divides(2, x)

def odd(x):
    return not divides(2, x)
```

Boolean Functions

- boolean functions = functions that return True or False
- useful e.g. as <cond> in a conditional execution
- Example:

```
def divides(x, y):
    return y % x == 0

def even(x):
    return divides(2, x)

def odd(x):
    return not even(x)
```

RECURSION:
SEE RECURSION

Recursion is "Complete"

- so far we know:
  - values of type integer, float, string
  - arithmetic expressions
  - (recursive) function definitions
  - (recursive) function calls
  - conditional execution
  - input/output

- ALL possible programs can be written using these elements!
- we say that we have a "Turing complete" language

Factorial

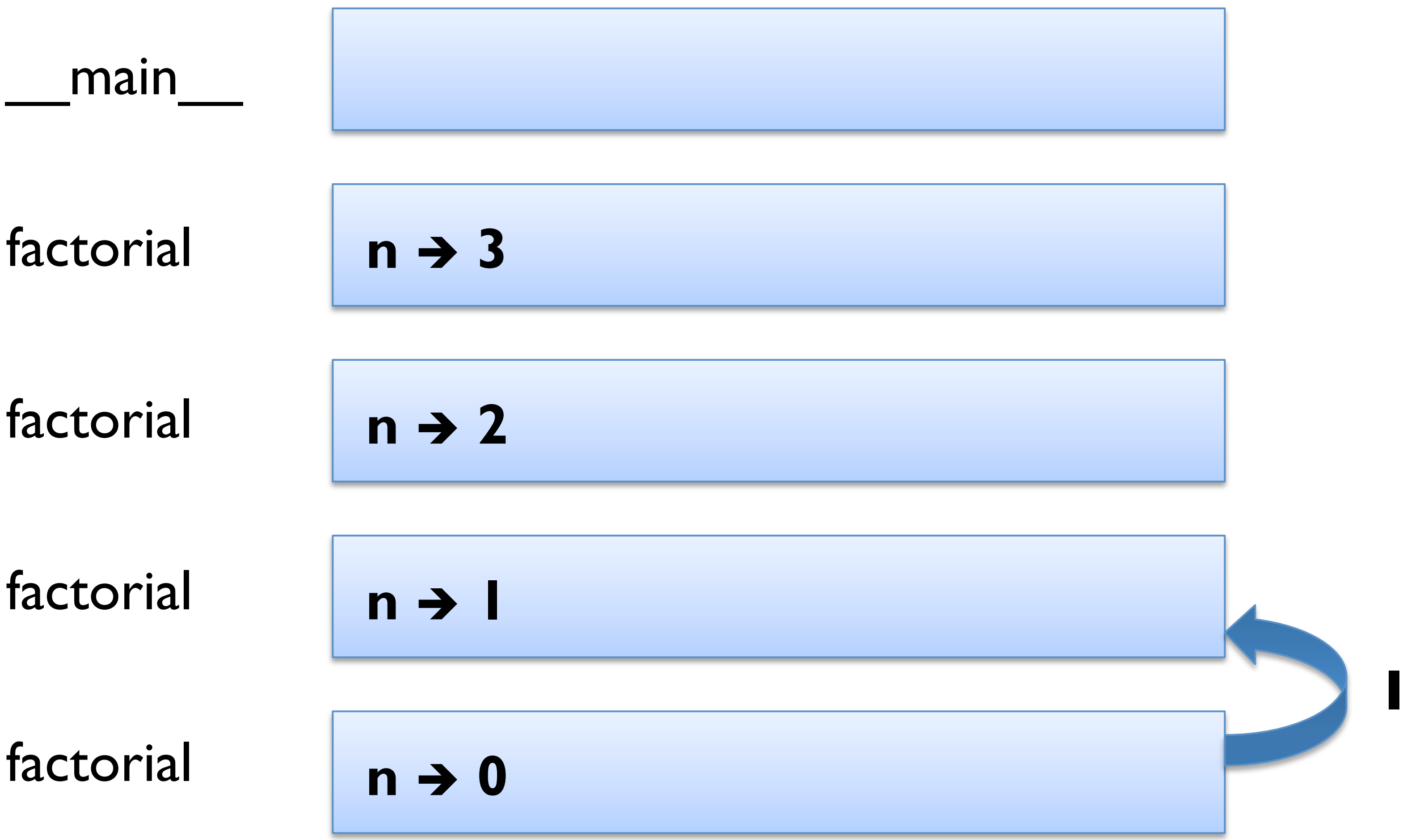
- in mathematics, the factorial function is defined by
  - 0! = 1
  - n! = n * (n-1)!
- such *recursive* definitions can trivially be expressed in Python
- Example:

```
def factorial(n):
    if n == 0:
        return 1
    recurse = factorial(n-1)
    result = n * recurse
    return result
x = factorial(3)
```

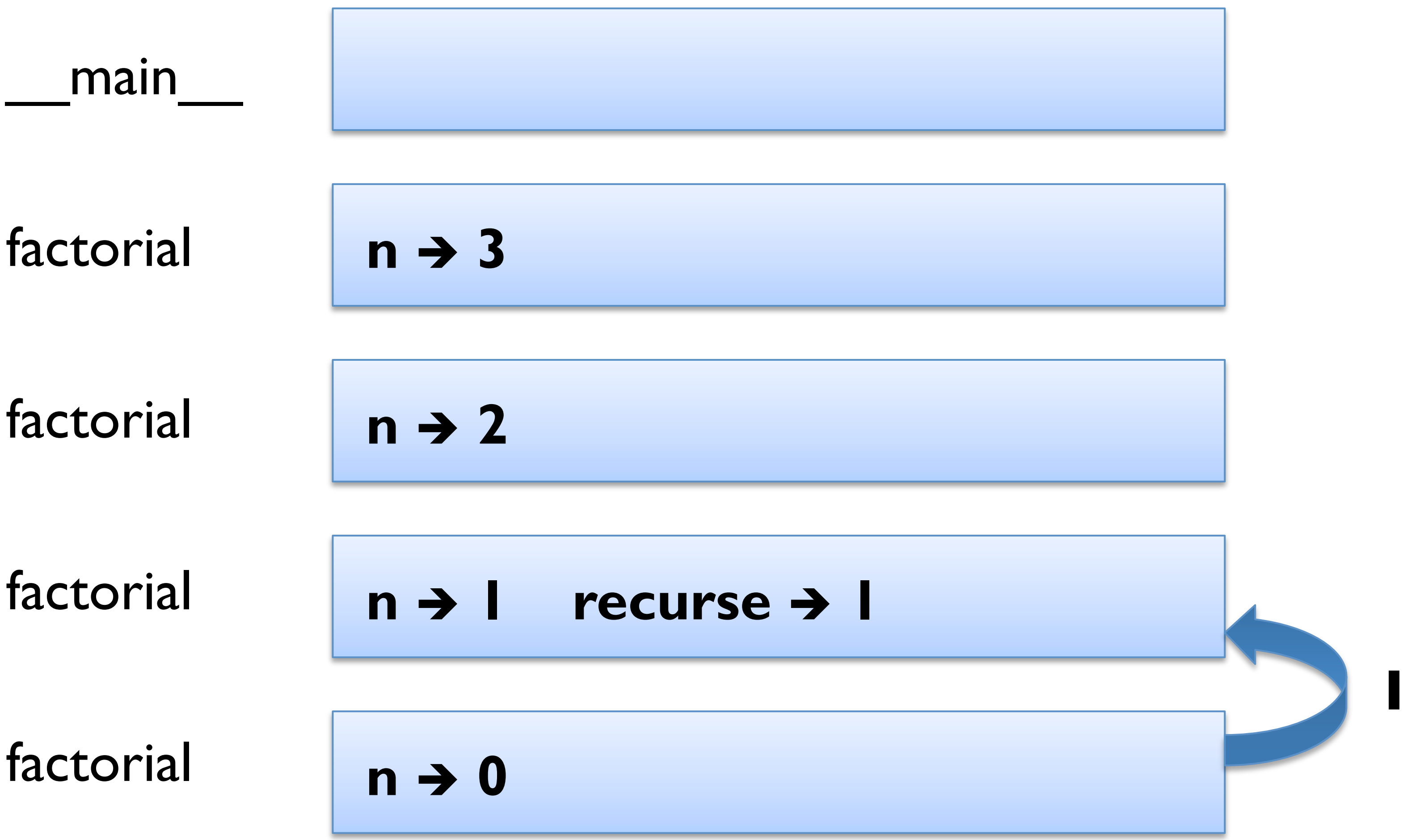
Stack Diagram for Factorial

__main__	
factorial	**n → 3**
factorial	**n → 2**
factorial	**n → 1**
factorial	**n → 0**

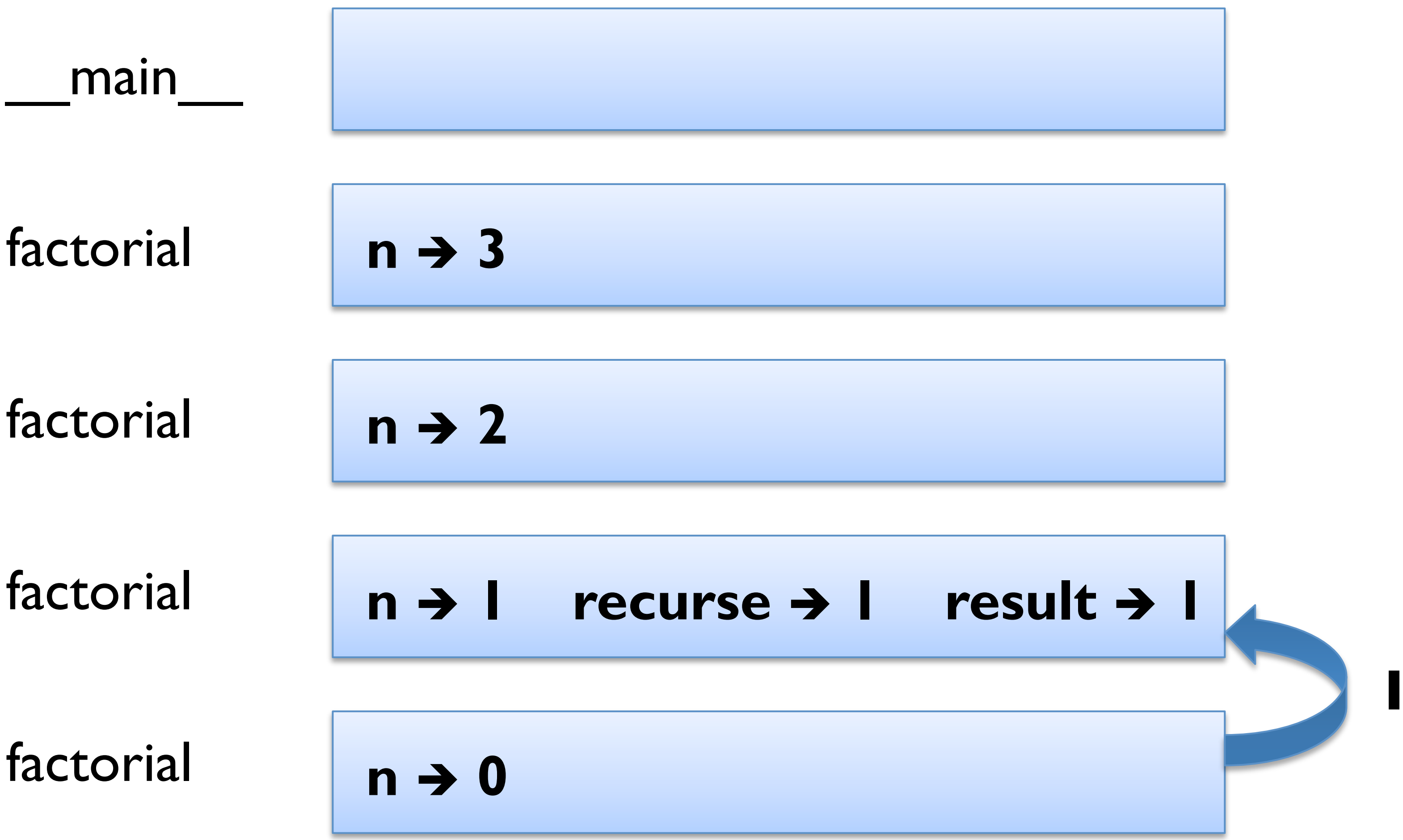
Stack Diagram for Factorial



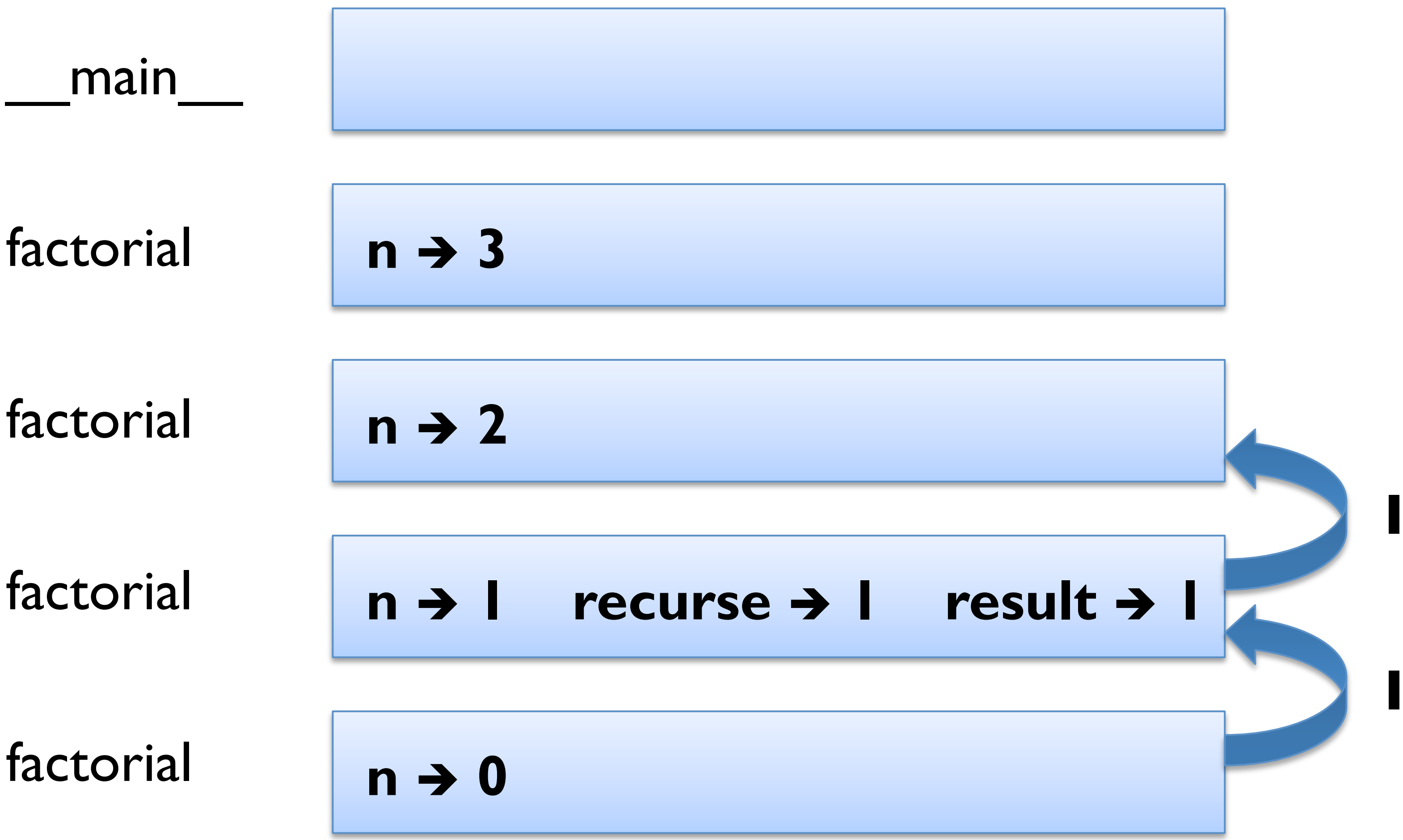
Stack Diagram for Factorial



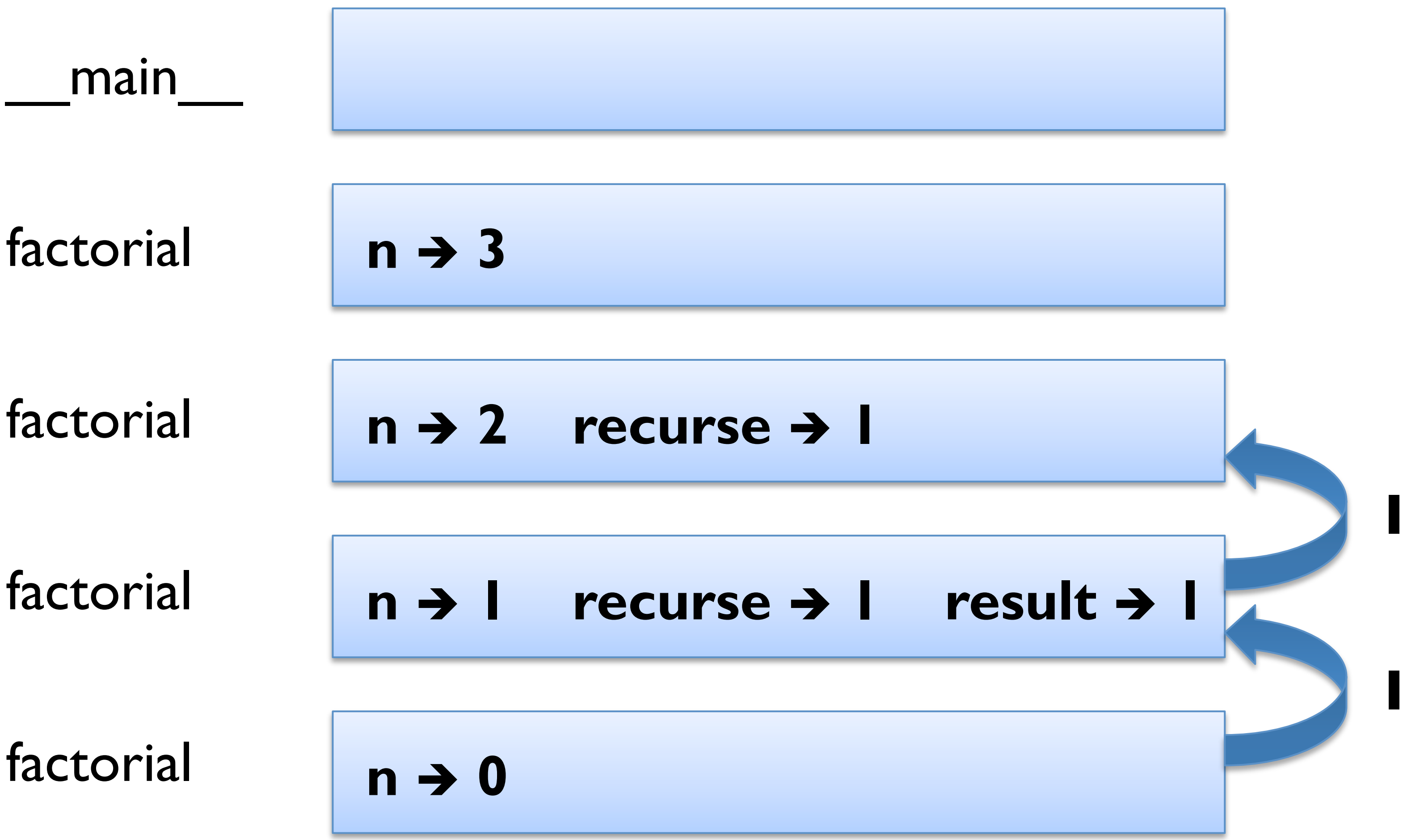
Stack Diagram for Factorial



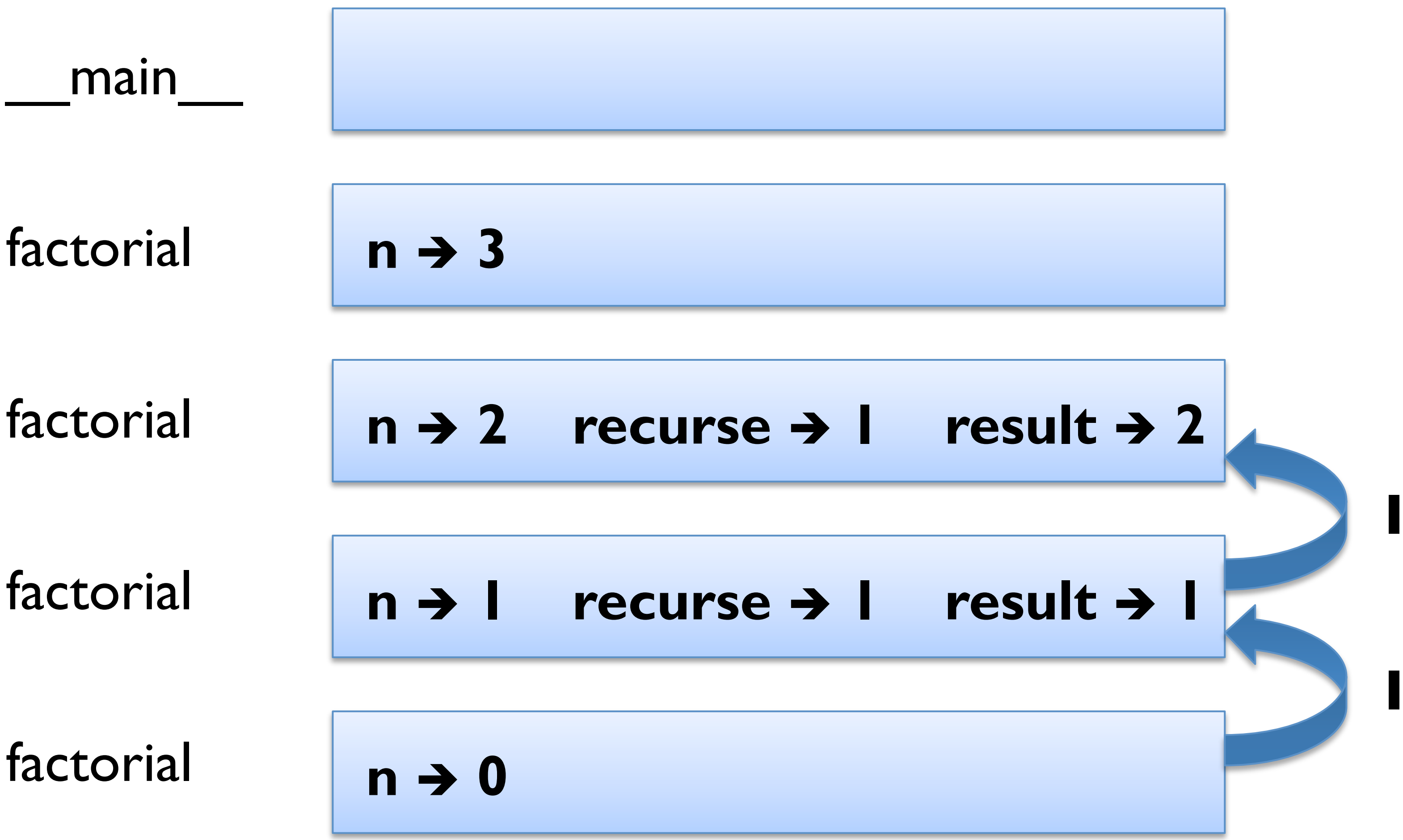
Stack Diagram for Factorial



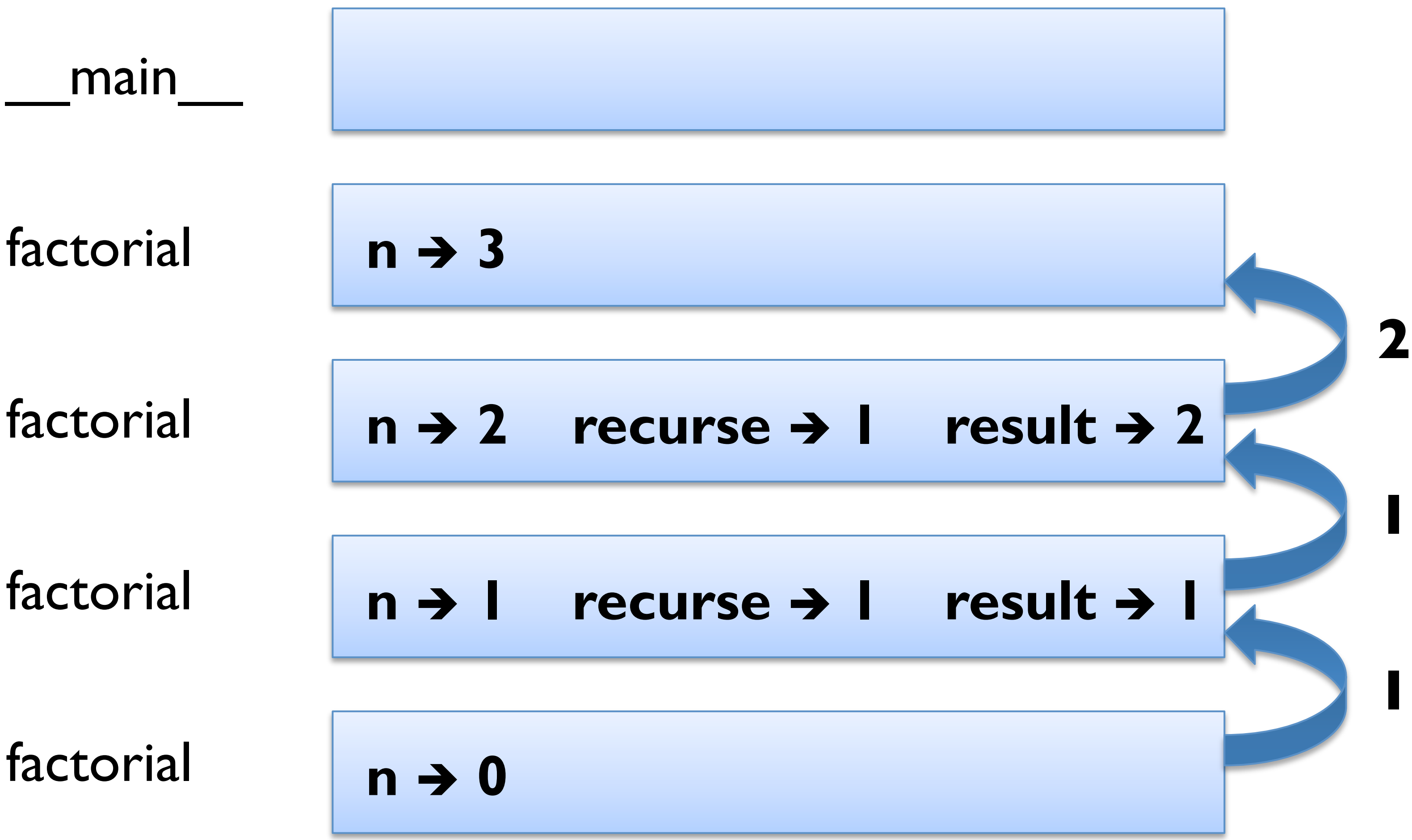
Stack Diagram for Factorial



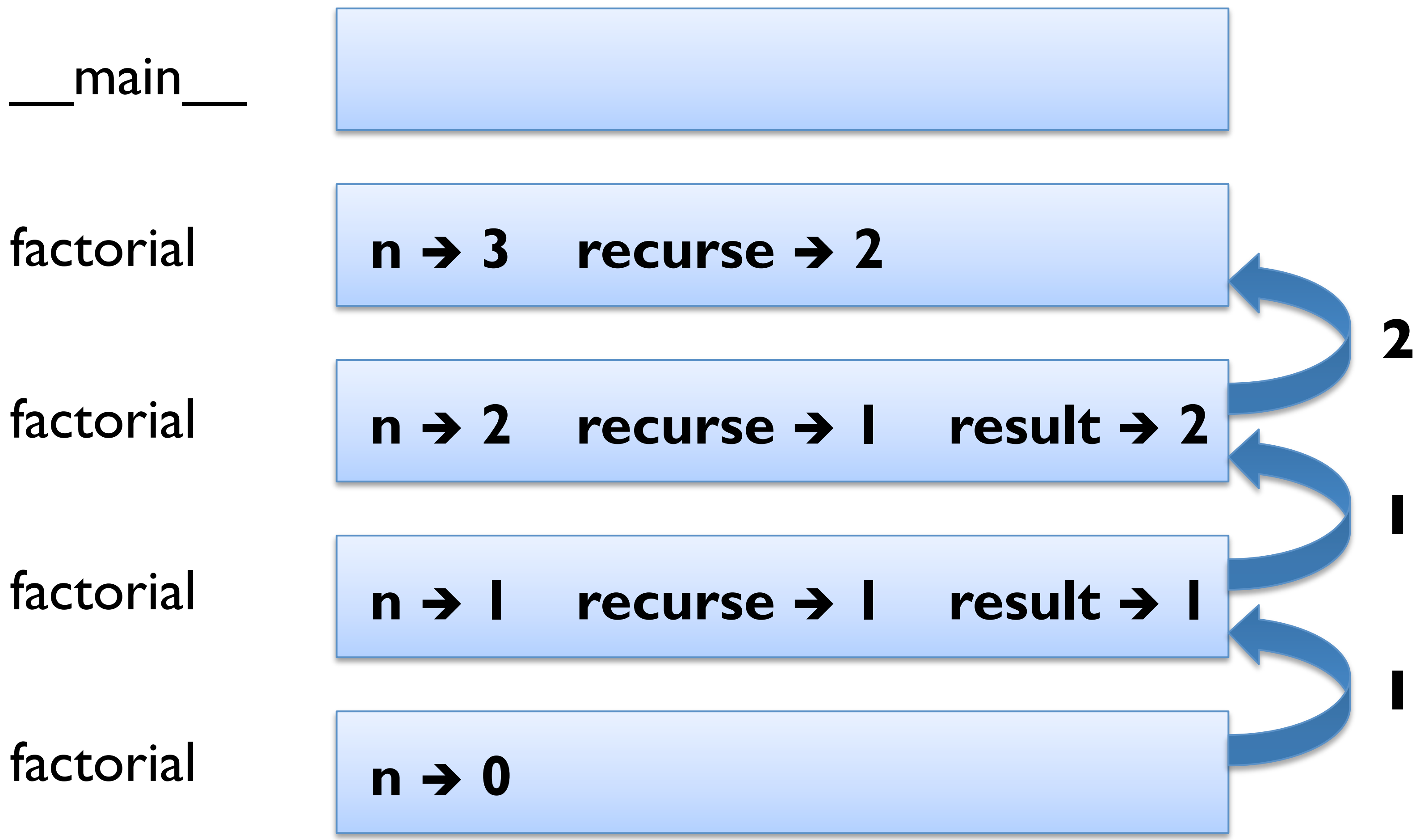
Stack Diagram for Factorial



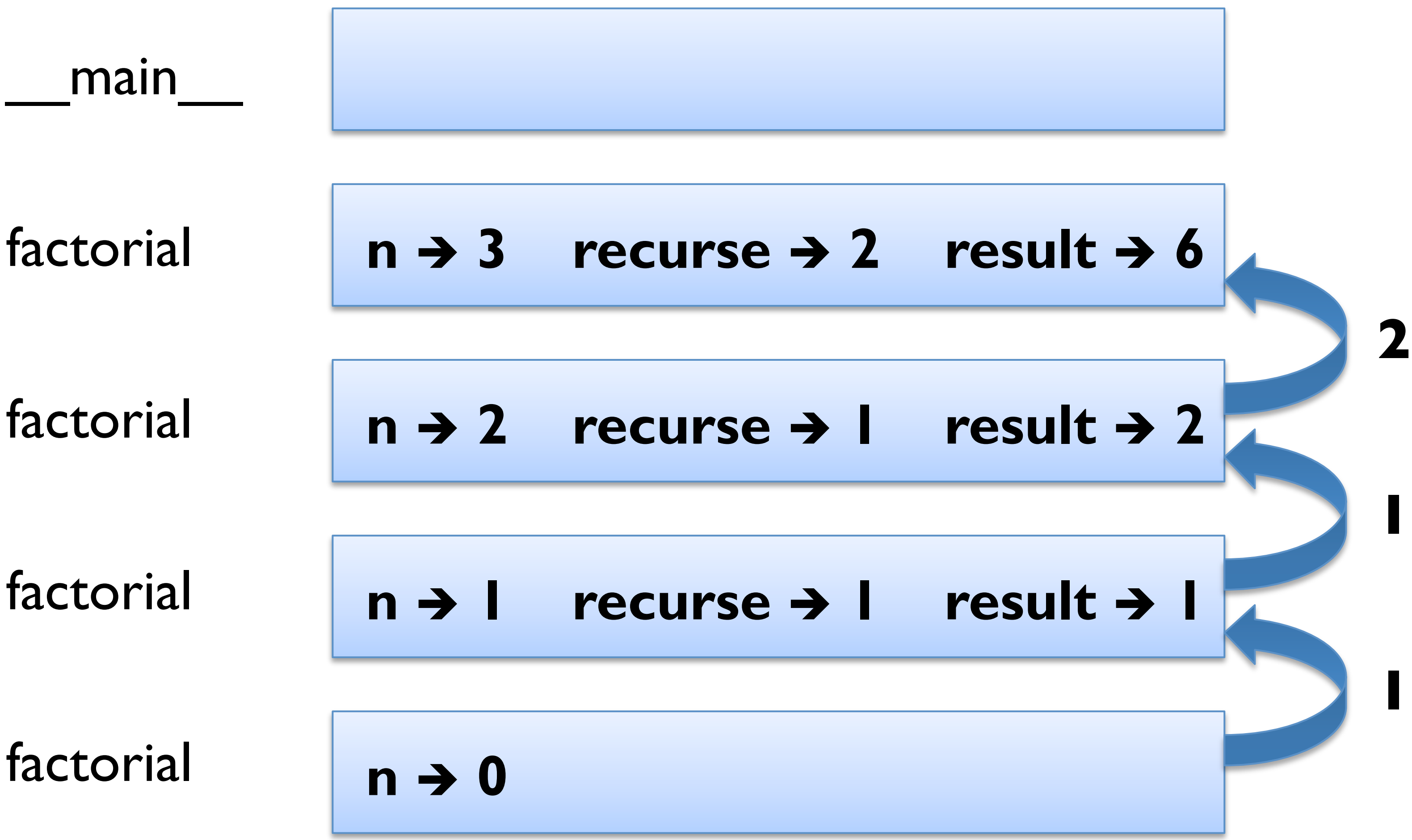
Stack Diagram for Factorial



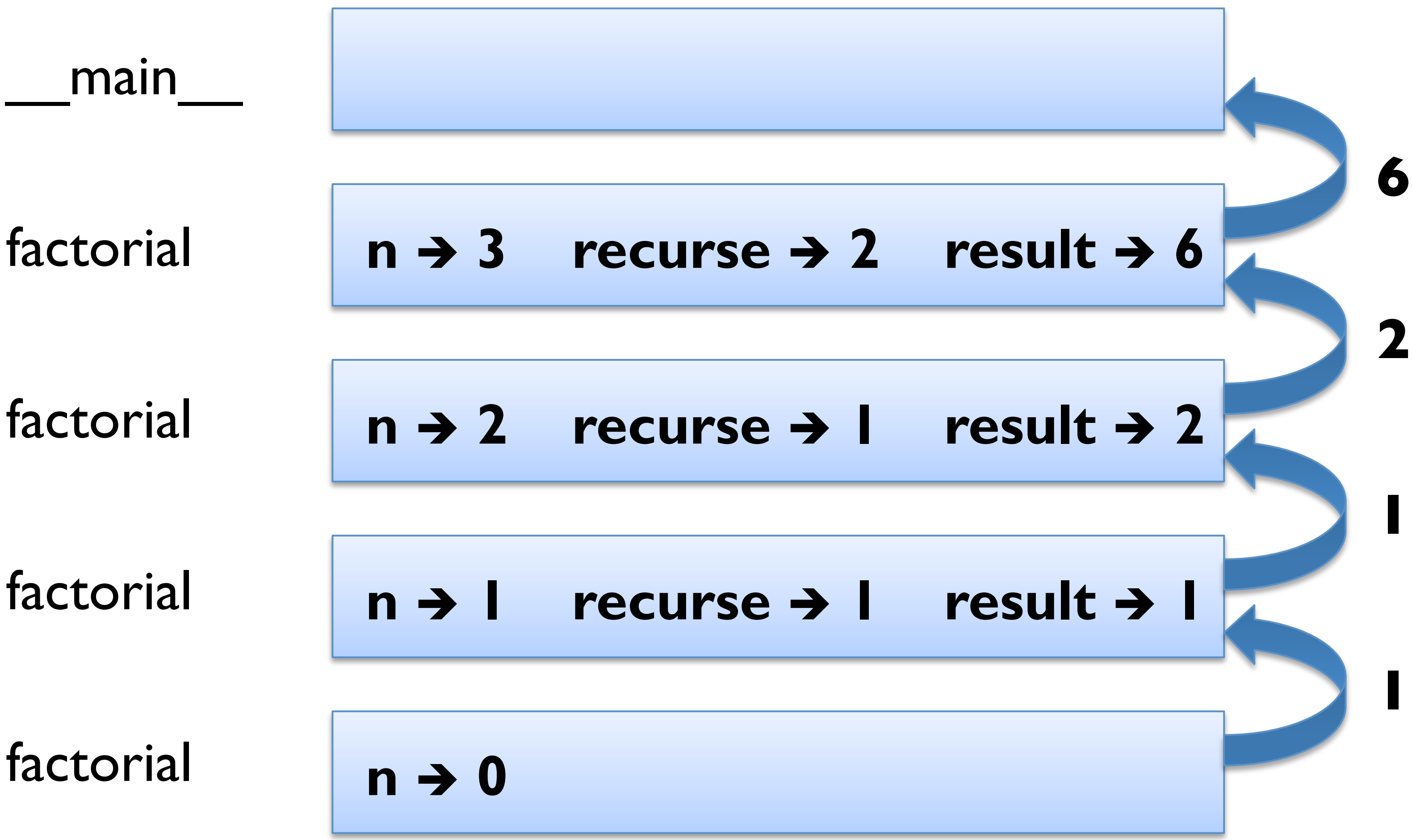
Stack Diagram for Factorial



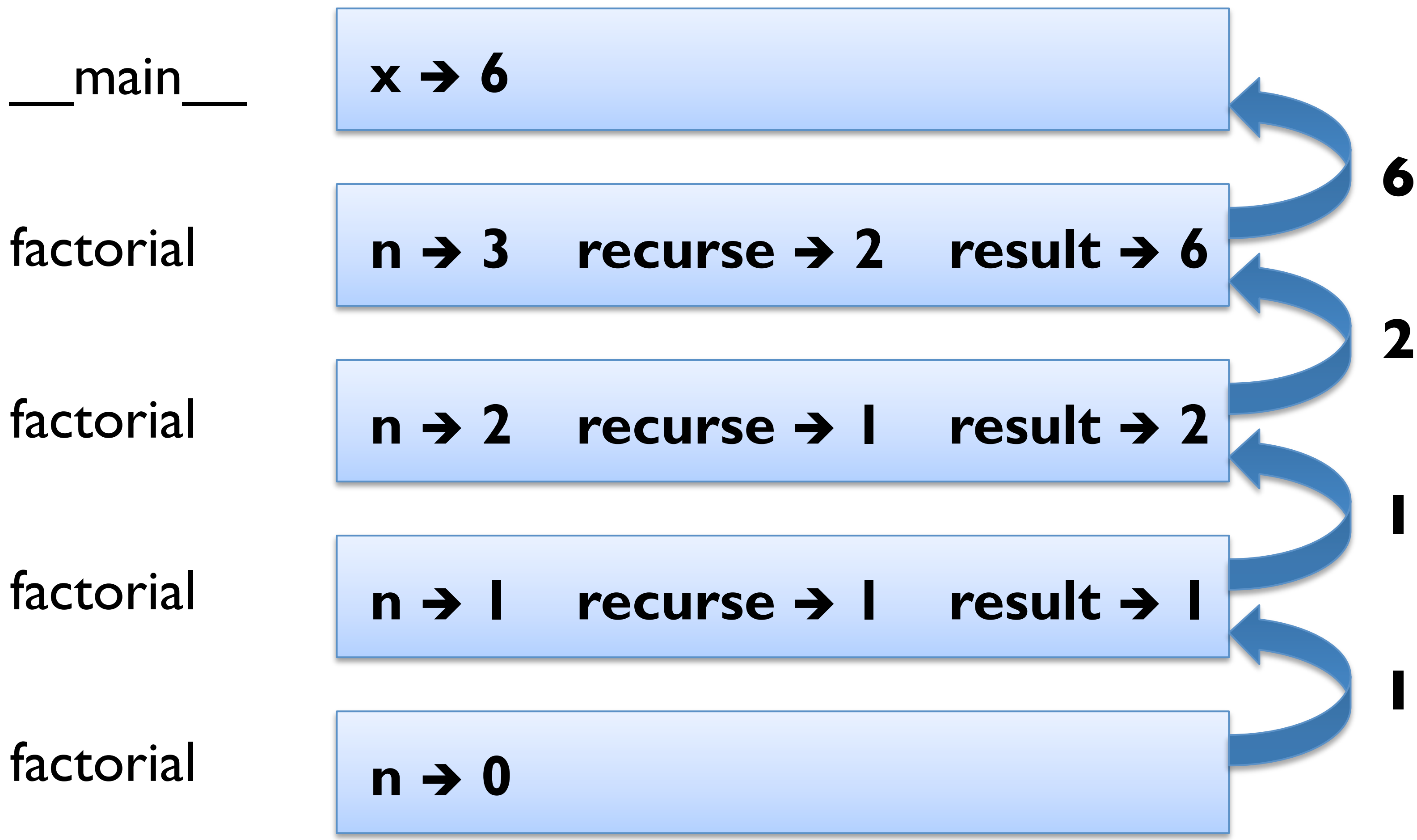
Stack Diagram for Factorial



Stack Diagram for Factorial



Stack Diagram for Factorial



Leap of Faith

- following the flow of execution difficult with recursion
- alternatively take the "leap of faith" (*induction*)

- Example:

```
def factorial(n):
    if n == 0:
        return 1
    recurse = factorial(n-1)
    result = n * recurse
    return result
x = factorial(3)
```

check the base case

assume recursive call is correct

check the step case

Control Flow Diagram

- Example:

```
def factorial(n):
    if n == 0:
        return 1
    recurse = factorial(n-1)
    result = n * recurse
    return result
```