DM536 / DM550 Part 1
Introduction to Programming

Peter Schneider-Kamp

petersk@imada.sdu.dk

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

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-Boooom!"
    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:

real problem

error reported

```
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)
```

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

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

1

Stack Diagram for Factorial

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

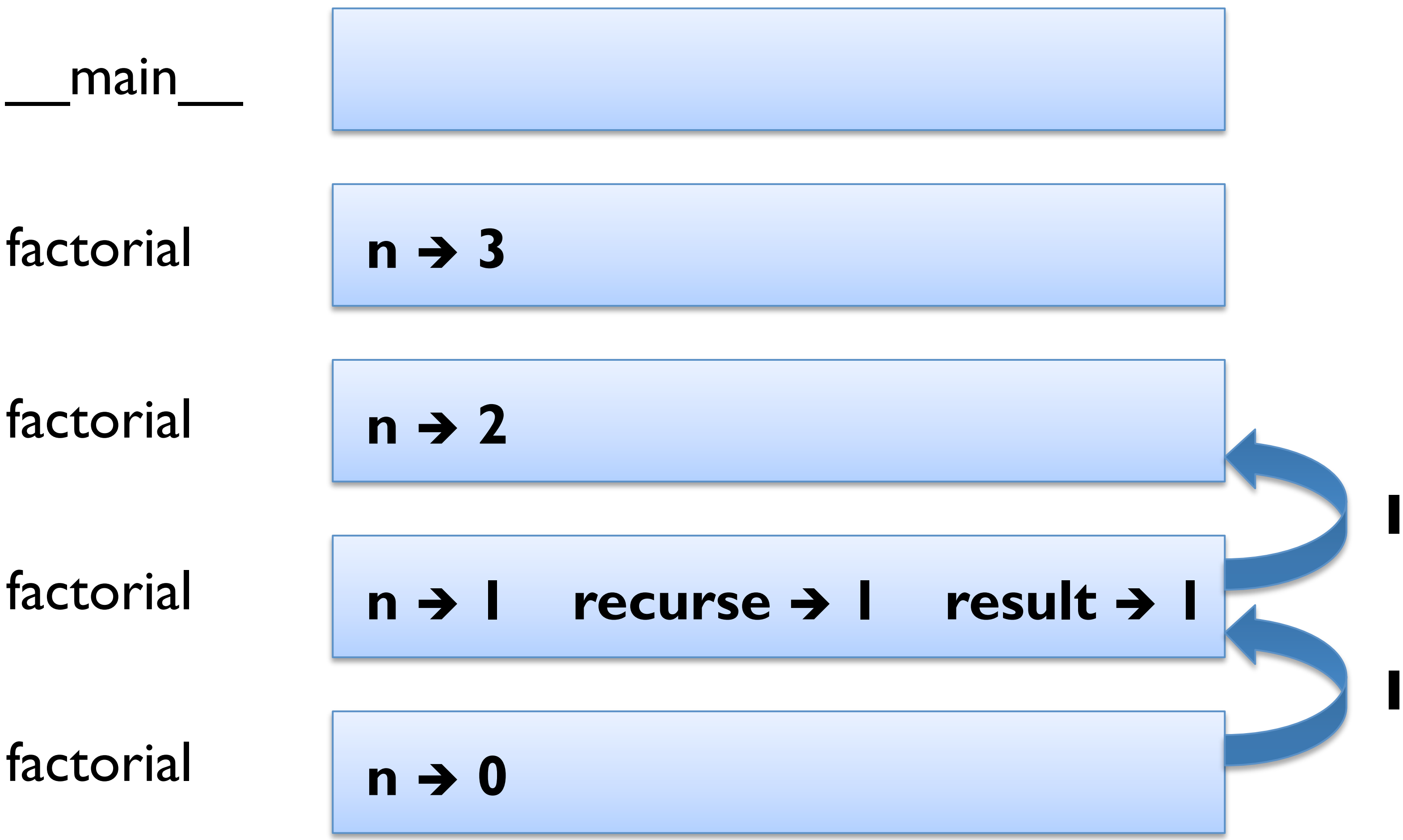
1

Stack Diagram for Factorial

__main__	
factorial	n → 3
factorial	n → 2
factorial	n → 1 recurse → 1 result → 1
factorial	n → 0

1

Stack Diagram for Factorial



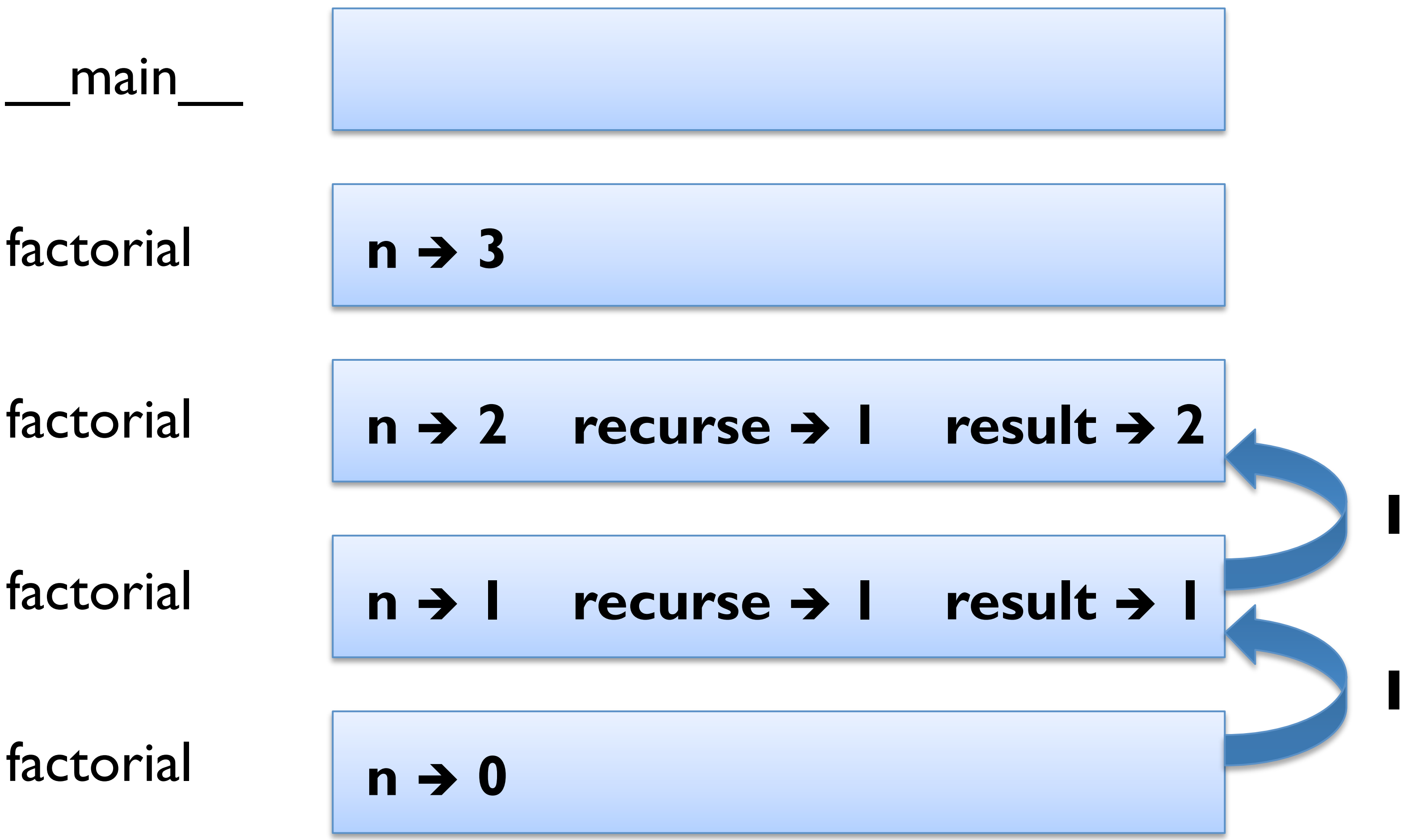
Stack Diagram for Factorial

__main__	
factorial	n → 3
factorial	n → 2 recurse → 1
factorial	n → 1 recurse → 1 result → 1
factorial	n → 0

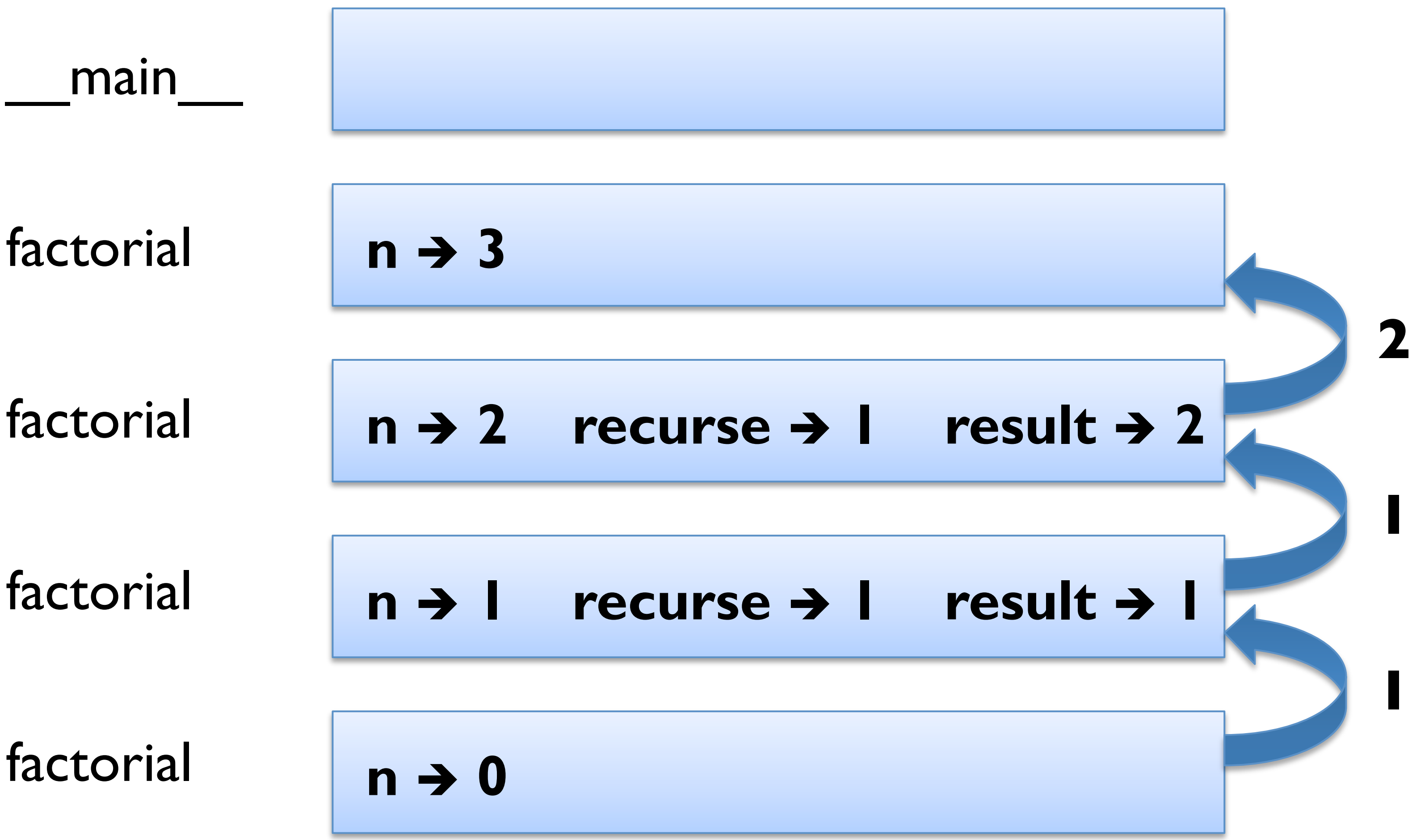
1

1

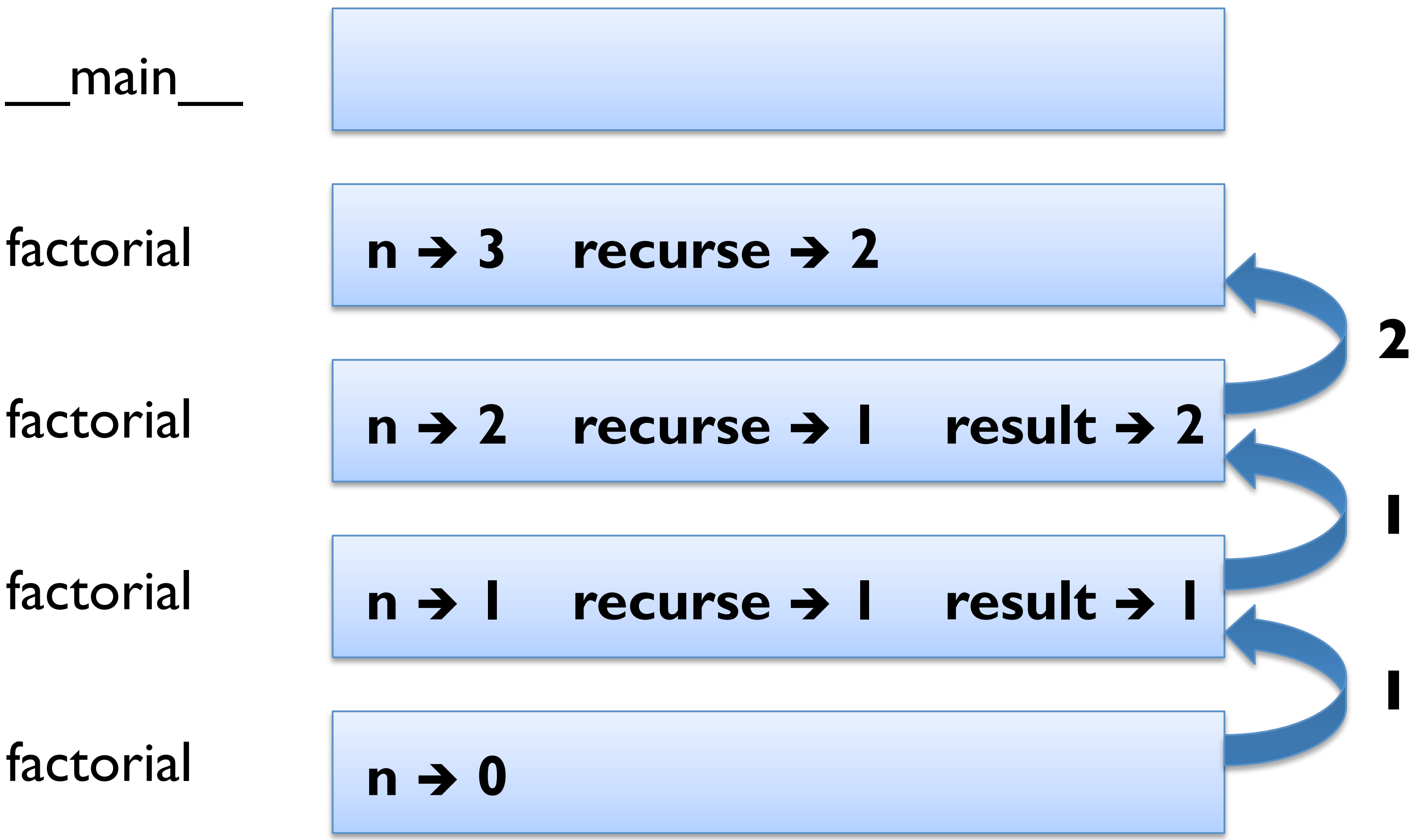
Stack Diagram for Factorial



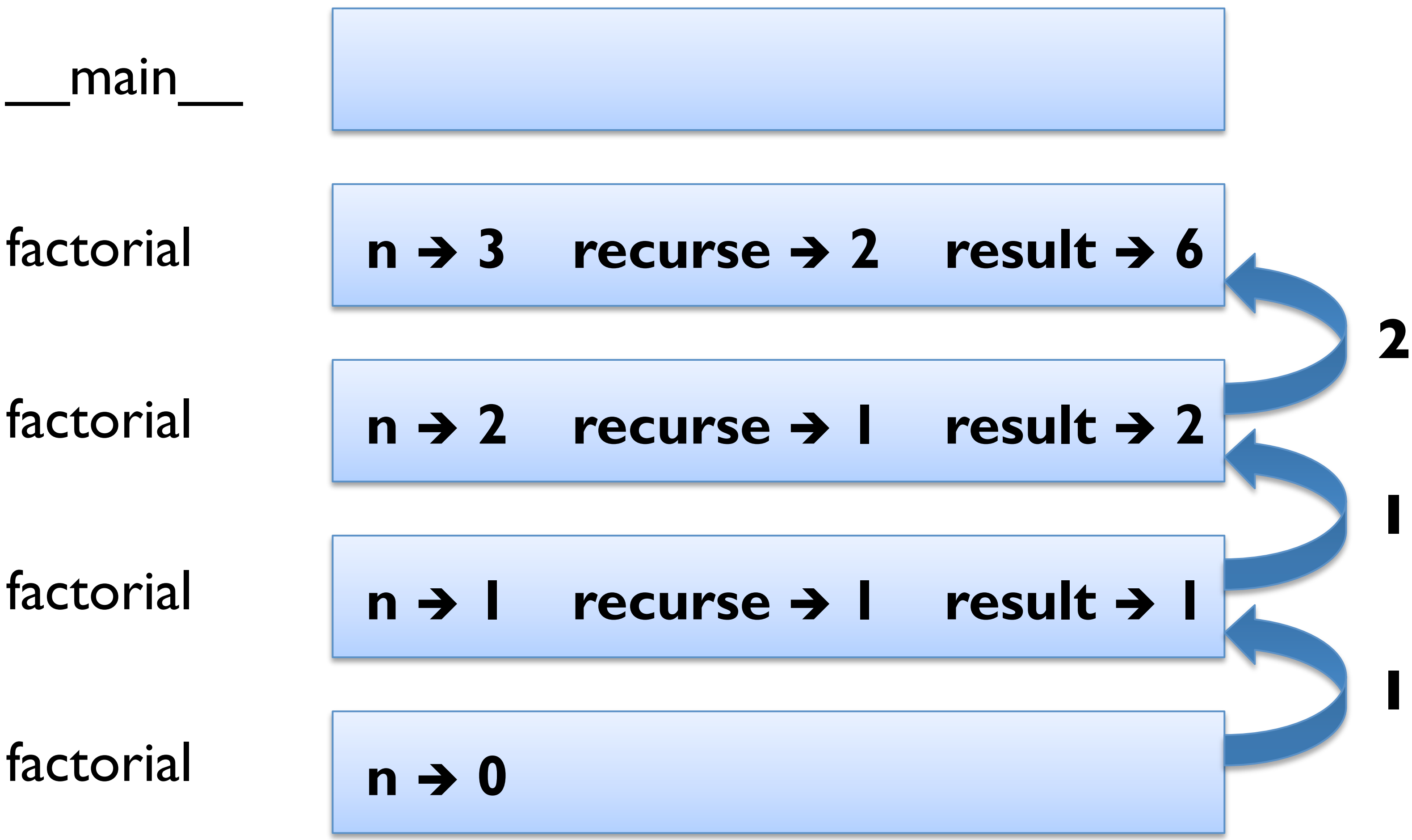
Stack Diagram for Factorial



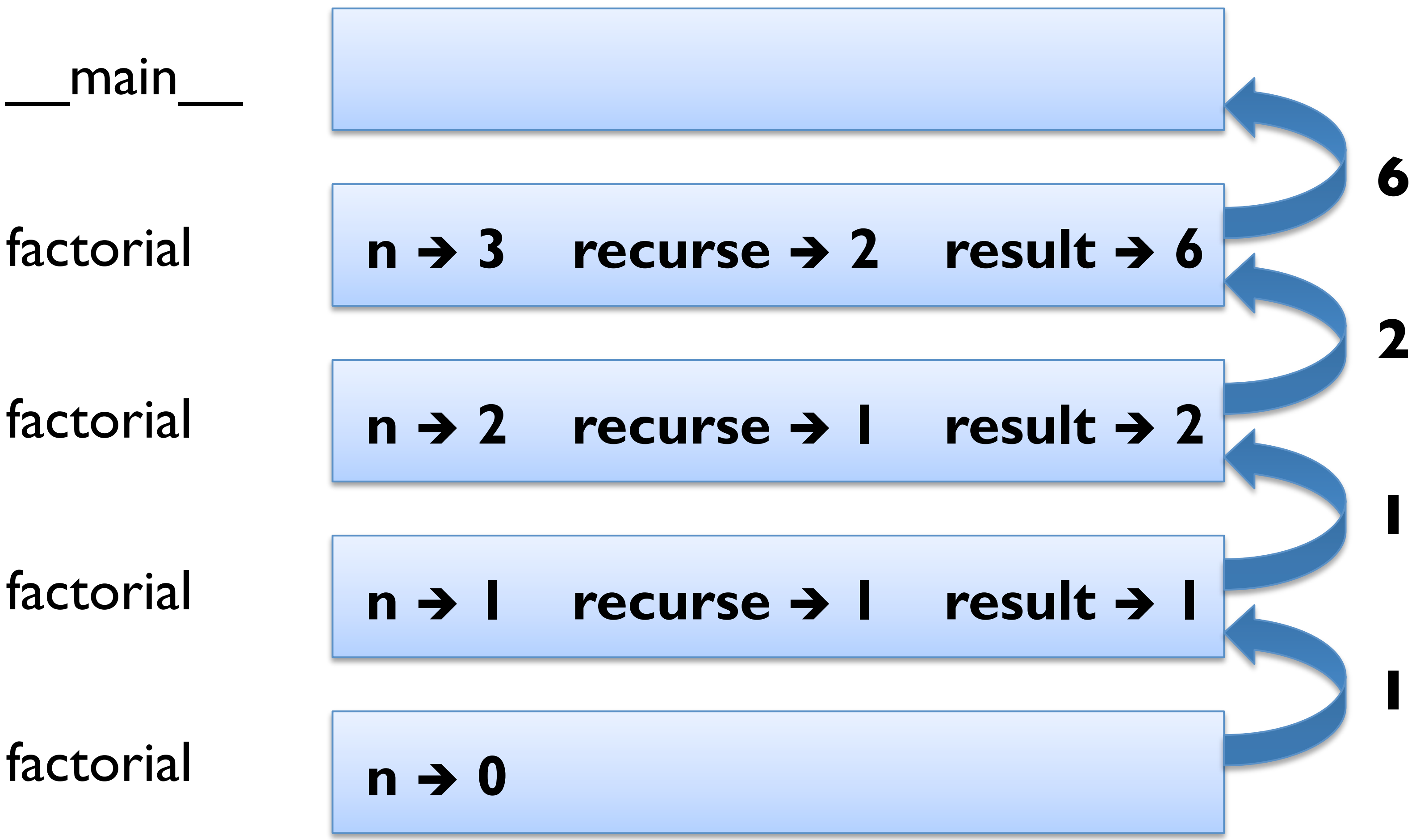
Stack Diagram for Factorial



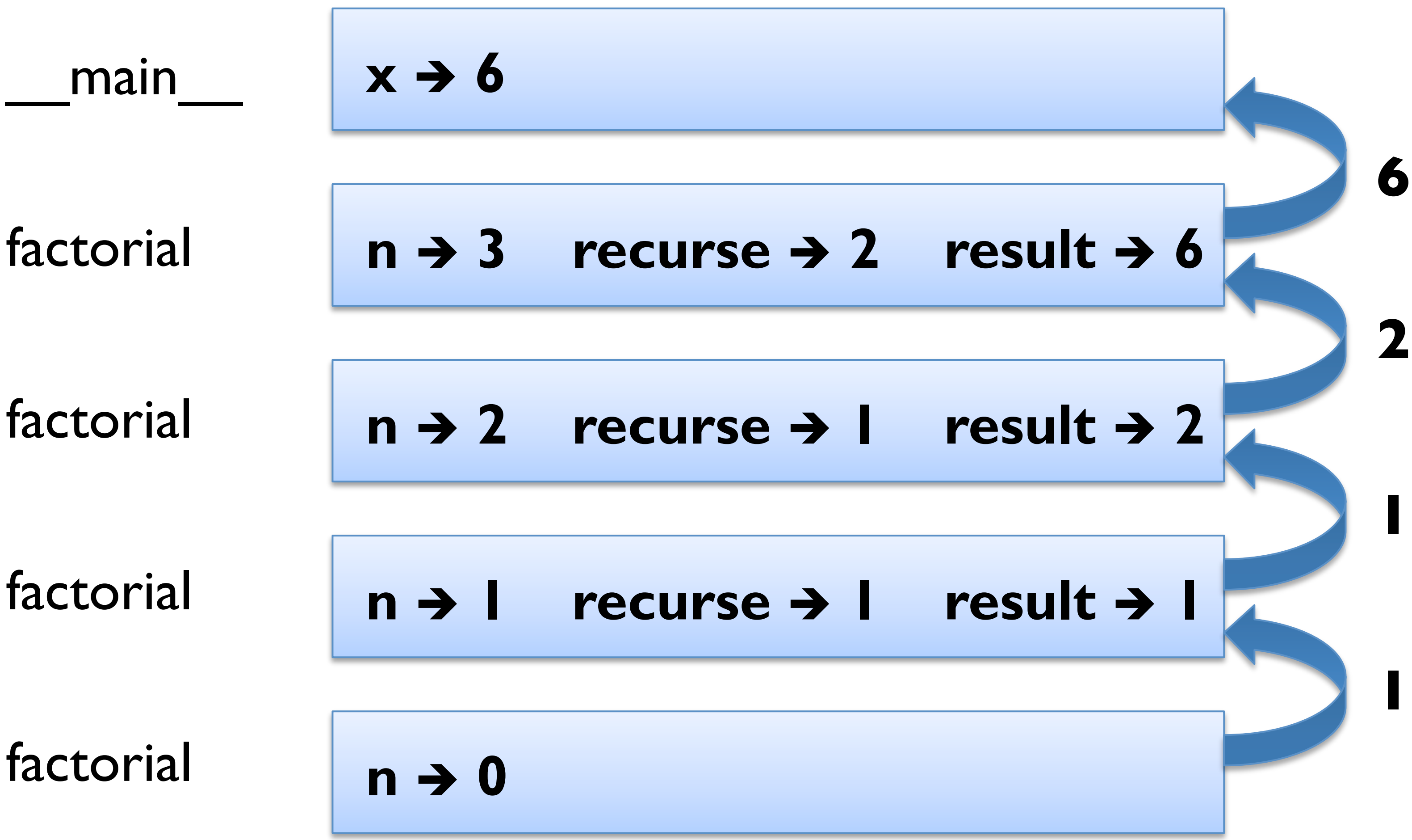
Stack Diagram for Factorial



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)
```

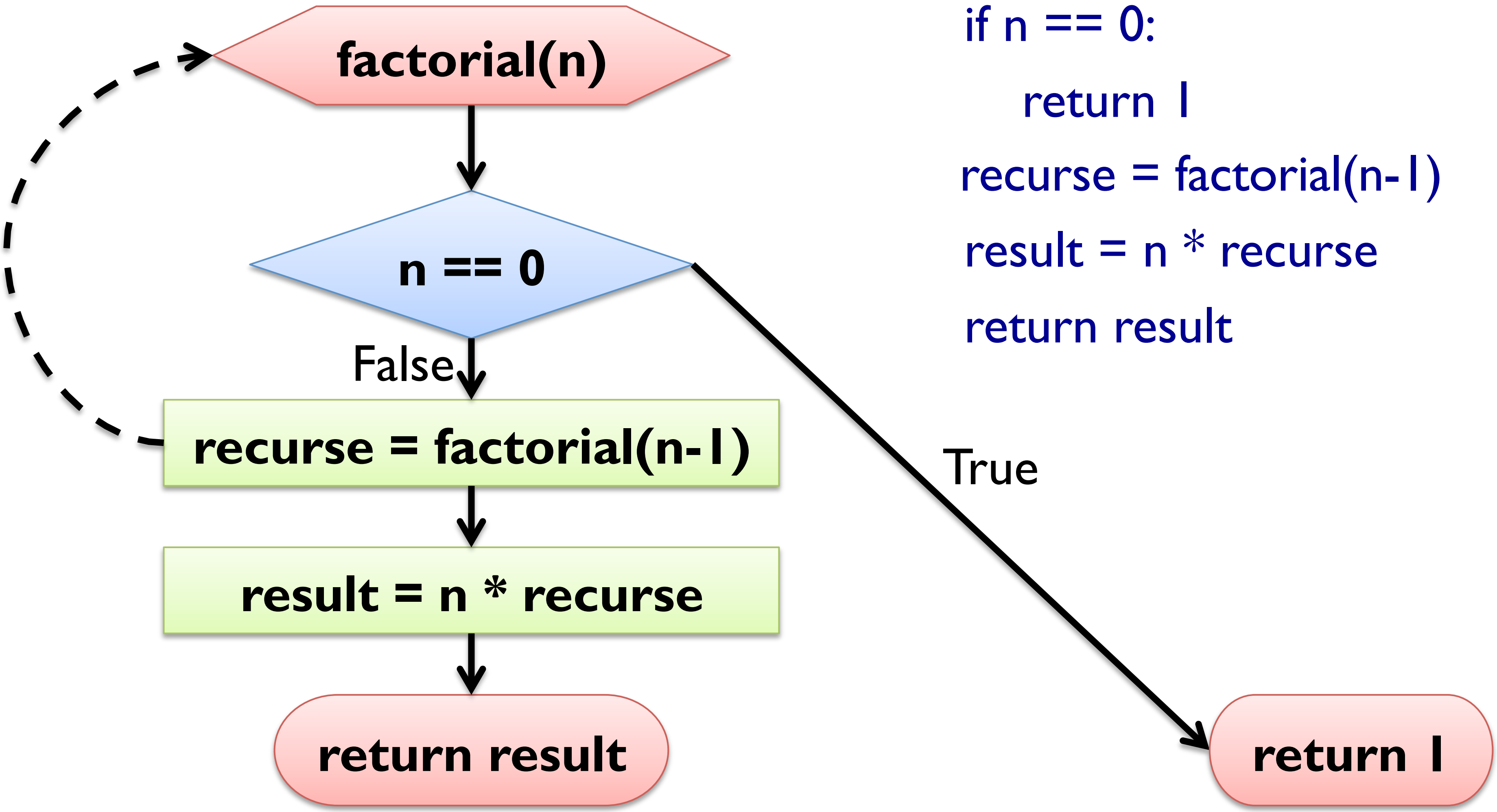
check the base case

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
```

Fibonacci

- Fibonacci numbers model for unchecked rabbit population
- rabbit pairs at generation n is sum of rabbit pairs at generation n-1 and generation n-2
- mathematically:
  - fib(0) = 0, fib(1) = 1, fib(n) = fib(n-1) + fib(n-2)
- Pythonically:

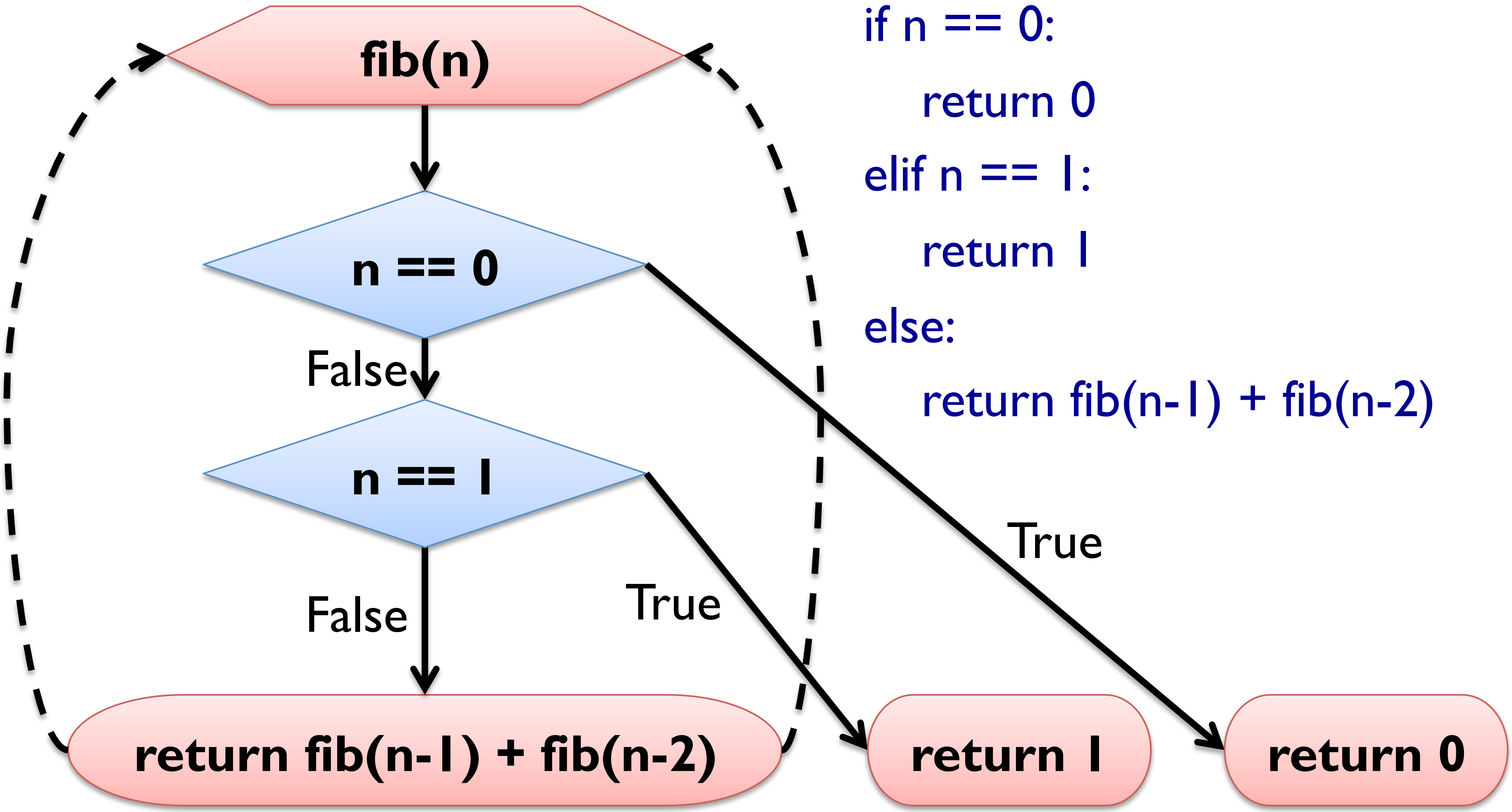
```
def fib(n):
    if n == 0:    return 0
    elif n == 1:  return 1
    else:         return fib(n-1) + fib(n-2)
```

- “leap of faith” required even for small n!

Control Flow Diagram

- Example:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```



Types and Base Cases

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

- **Problem:** factorial(1.5) exceeds recursion limit
- factorial(0.5)
- factorial(-0.5)
- factorial(-1.5)
- …

Types and Base Cases

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

- **Idea:** check type at beginning of function

Types and Base Cases

```
def factorial(n):
    if not isinstance(n, int):
        print "Integer required";  return None
    if n == 0:
        return 1
    recurse = factorial(n-1)
    result = n * recurse
    return result
```

- **Idea:** check type at beginning of function

Types and Base Cases

```
def factorial(n):
    if not isinstance(n, int):
        print "Integer required";  return None
    if n < 0:
        print "Non-negative number expected";  return None
    if n == 0:
        return 1
    recurse = factorial(n-1)
    result = n * recurse
    return result
```

- **Idea:** check type at beginning of function

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

Debugging (Recursive) Functions

- to check pre-conditions:
  - print values & types of parameters at beginning of function
  - insert check at beginning of function (*pre assertion*)

- to check post-conditions:
  - print values before return statements
  - insert check before return statements (*post assertion*)

- side-effect: visualize flow of execution

ITERATION

Multiple Assignment Revisited

- as seen before, variables can be assigned multiple times
- assignment is NOT the same as equality
- it is not symmetric, and changes with time

- Example:

```
a = 42
…
b = a
…
a = 23
```

from here, a and b are equal

from here, a and b are different

Updating Variables

- most common form of multiple assignment is *updating*
- a variable is assigned to an expression containing that variable

- Example:

```
x = 23
for i in range(19):
    x = x + 1
```

- adding one is called *incrementing*

- expression evaluated BEFORE assignment takes place
- thus, variable needs to have been *initialized* earlier!

Iterating with While Loops

- iteration = repetition of code blocks
- can be implemented using recursion (countdown, polyline)

- while statement:

  <while-loop> => while <cond>:

  <instr$_1$>; <instr$_2$>; <instr$_3$>

- Example:

```
def countdown(n):
    while n > 0:
        print n, "seconds left!"
        n = n - 1
    print "Ka-Boom!"
countdown(3)
```

n == 0

False

Termination

- Termination = the condition is eventually False
- loop in countdown obviously terminates:

```
while n > 0:    n = n - 1
```

- difficult for other loops:

```
def collatz(n):
    while n != 1:
        print n,
        if n % 2 == 0:        # n is even
            n = n / 2
        else:                 # n is odd
            n = 3 * n + 1
```

Termination

- Termination = the condition is eventually False
- loop in countdown obviously terminates:

```
while n > 0:      n = n - 1
```

- can also be difficult for recursion:

```
def collatz(n):
    if n != 1:
        print n,
        if n % 2 == 0:          # n is even
            collatz(n / 2)
        else:                   # n is odd
            collatz(3 * n + 1)
```

Breaking a Loop

- sometimes you want to *force* termination
- Example:

```
while True:
    num = raw_input('enter a number (or "exit"):\n')
    if num == "exit":
        break
    n = int(num)
    print "Square of", n, "is:", n**2
print "Thanks a lot!"
```

Approximating Square Roots

- Newton's method for finding root of a function f:
  1. start with some value x_0
  2. refine this value using $x_{n+1} = x_n - f(x_n) / f'(x_n)$
- for square root of a: $f(x) = x^2 - a$ $f'(x) = 2x$
- simplifying for this special case: $x_{n+1} = (x_n + a / x_n) / 2$
- Example 1:

```
while True:
    print xn
    xnp1 = (xn + a / xn) / 2
    if xnp1 == xn:
        break
    xn = xnp1
```

Approximating Square Roots

- Newton's method for finding root of a function f:
  1. start with some value x_0
  2. refine this value using $x_{n+1} = x_n - f(x_n) / f'(x_n)$

- Example 2:

```
def f(x):       return x**3 - math.cos(x)
def f1(x):      return 3*x**2 + math.sin(x)
while True:
    print xn
    xnp1 = xn - f(xn) / f1(xn)
    if xnp1 == xn:
        break
    xn = xnp1
```

Approximating Square Roots

- Newton's method for finding root of a function f:
  1. start with some value x_0
  2. refine this value using $x_{n+1} = x_n - f(x_n) / f'(x_n)$

- Example 2:

```
def f(x):       return x**3 - math.cos(x)
def f1(x):      return 3*x**2 + math.sin(x)
while True:
    print xn
    xnp1 = xn - f(xn) / f1(xn)
    if math.abs(xnp1 - xn) < epsilon:
        break
    xn = xnp1
```

Algorithms

- algorithm = mechanical problem-solving process
- usually given as a step-by-step procedure for computation

- Newton's method is an example of an algorithm
- other examples:
  - addition with carrying
  - subtraction with borrowing
  - long multiplication
  - long division

- directly using Pythagora's formula is not an algorithm

Divide et Impera

- latin, means “divide and conquer” (courtesy of Julius Caesar)
- **Idea:** break down a problem and recursively work on parts
- Example: guessing a number by bisection

```
def guess(low, high):
    if low == high:
        print "Got you! You thought of: ", low
    else:
        mid = (low+high) / 2
        ans = raw_input("Is "+str(mid)+" correct (>, =, <)?")
        if ans == ">":      guess(mid,high)
        elif ans == "<":    guess(low,mid)
        else:               print "Yeehah! Got you!"
```

Debugging Larger Programs

- assume you have large function computing wrong return value
- going step-by-step very time consuming

- **Idea:** use bisection, i.e., half the search space in each step

1. insert intermediate output (e.g. using print) at mid-point
2. if intermediate output is correct, apply recursively to 2^{nd} part
3. if intermediate output is wrong, apply recursively to 1^{st} part