DM502
Programming A

Peter Schneider-Kamp

petersk@imada.sdu.dk

http://imada.sdu.dk/~petersk/DM502/

Python & Linux Install Party

- Tomorrow (Tuesday, September 12) from 10 – 14
- Fredagsbar (area south of Kantine II)

- Participants are those
  - who want Python (& Swampy) on their computer,
  - who want Linux on their computer,
  - who want some study-related software on their computer,
  - who have problems with some study related software, or
  - who just like to hang out and help other people!

- drinks and some snacks will be provided by IMADA!

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

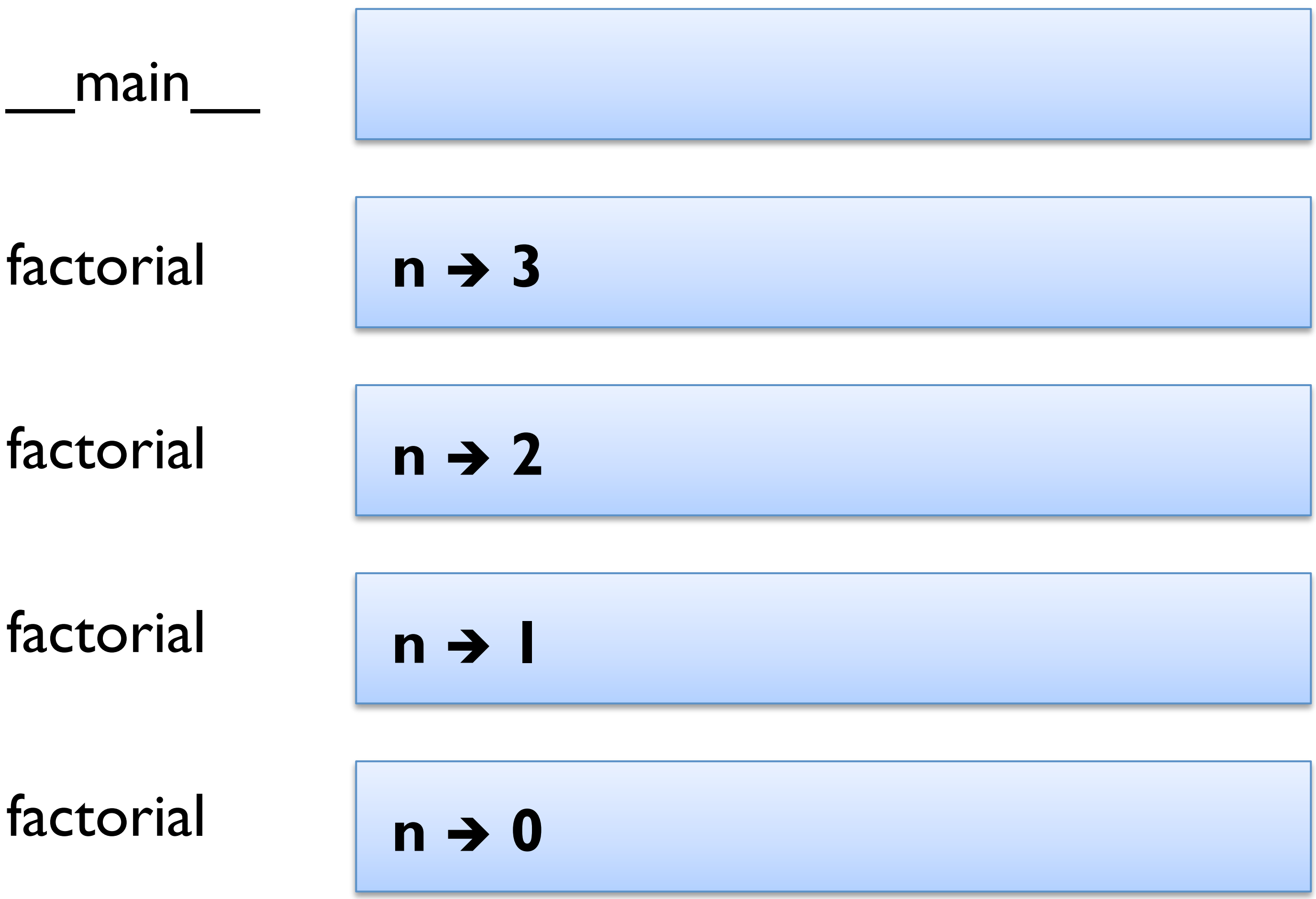
- ALL possible programs can be written using these elements!
- we say that we have a "Turing complete" language

Factorial

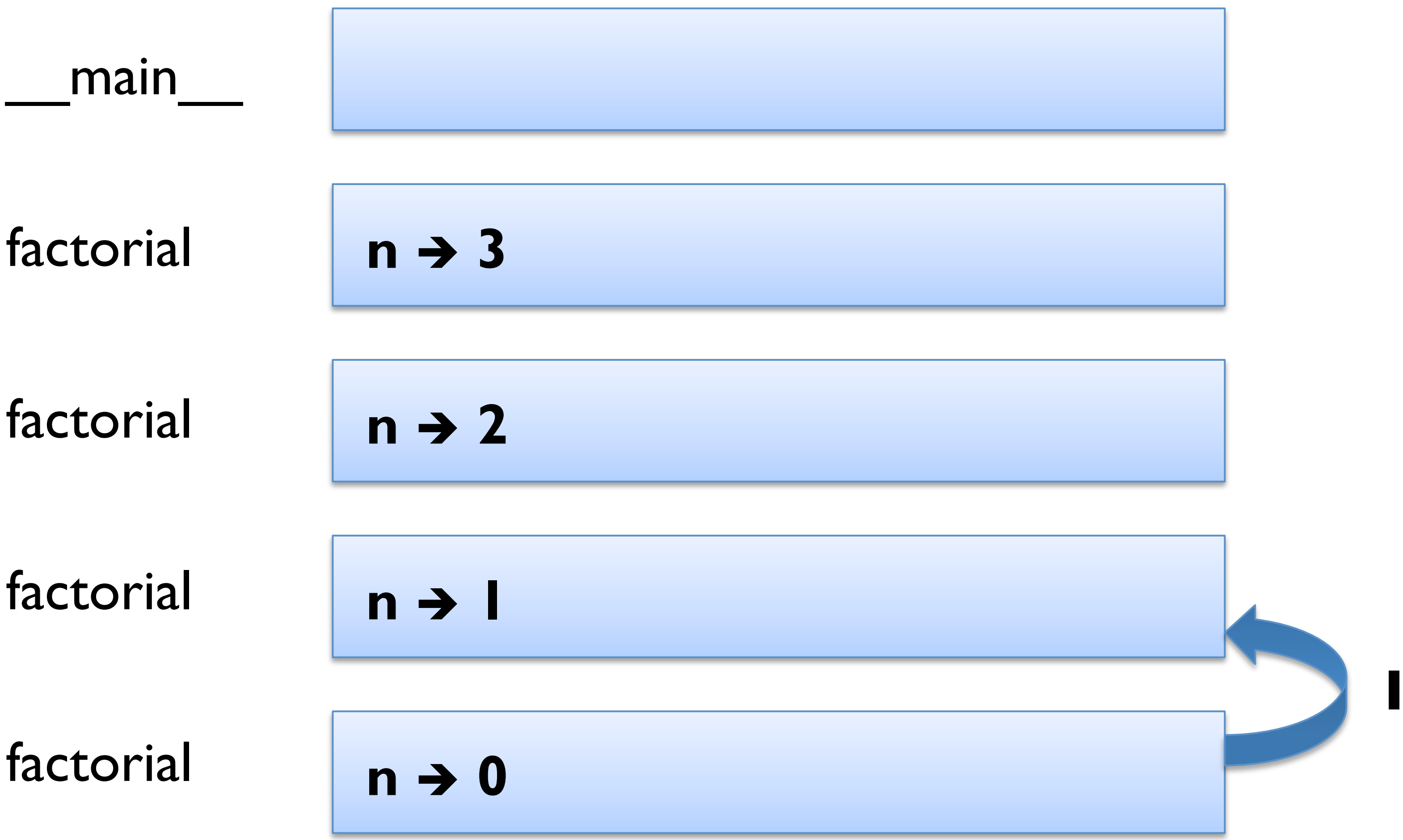
- in mathematics, the factorial function is defined by
  - 0! = 1
  - n! = n * (n-1)!
- such *recursive* definitions can trivially be expressed in Python
- Example:

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

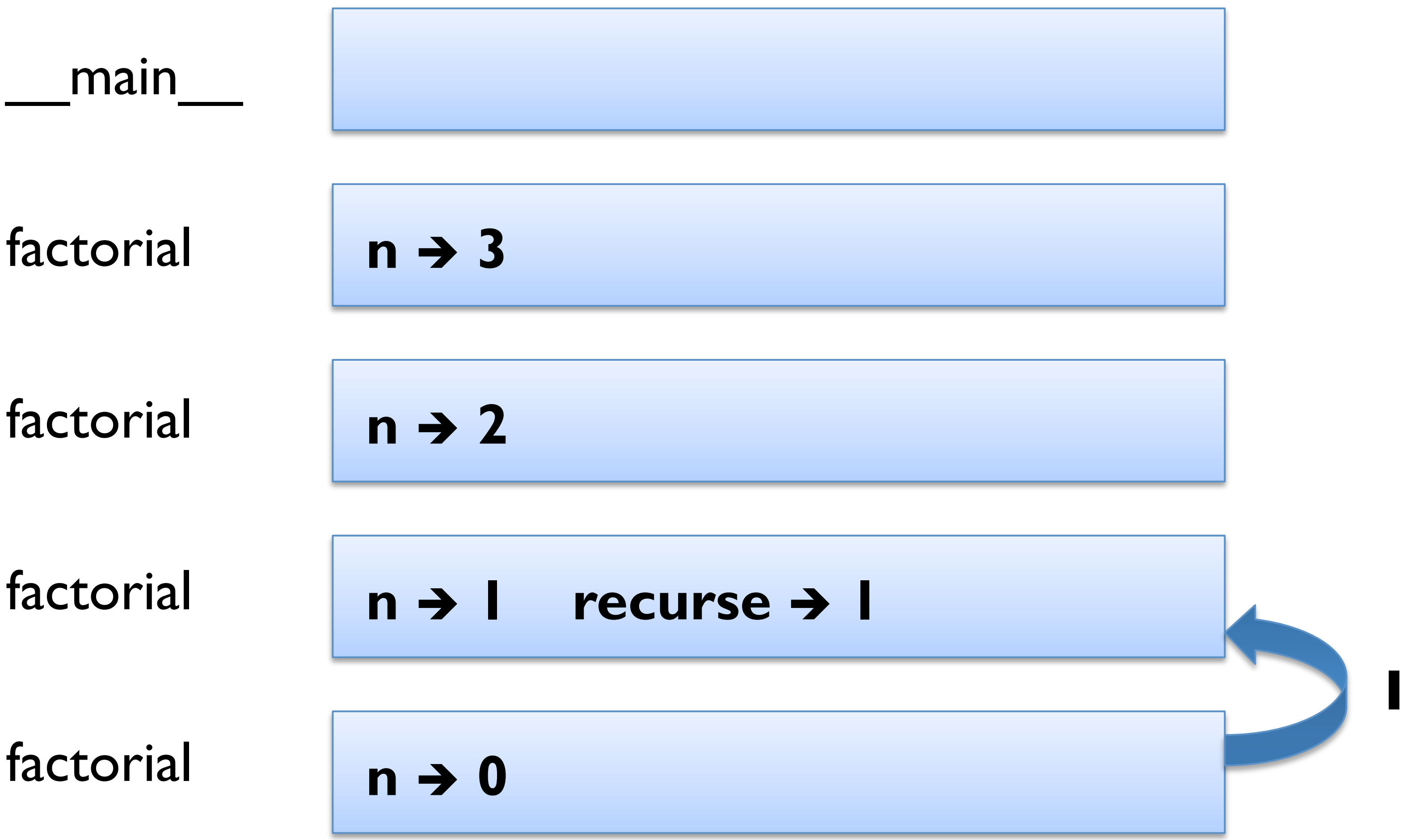
Stack Diagram for Factorial



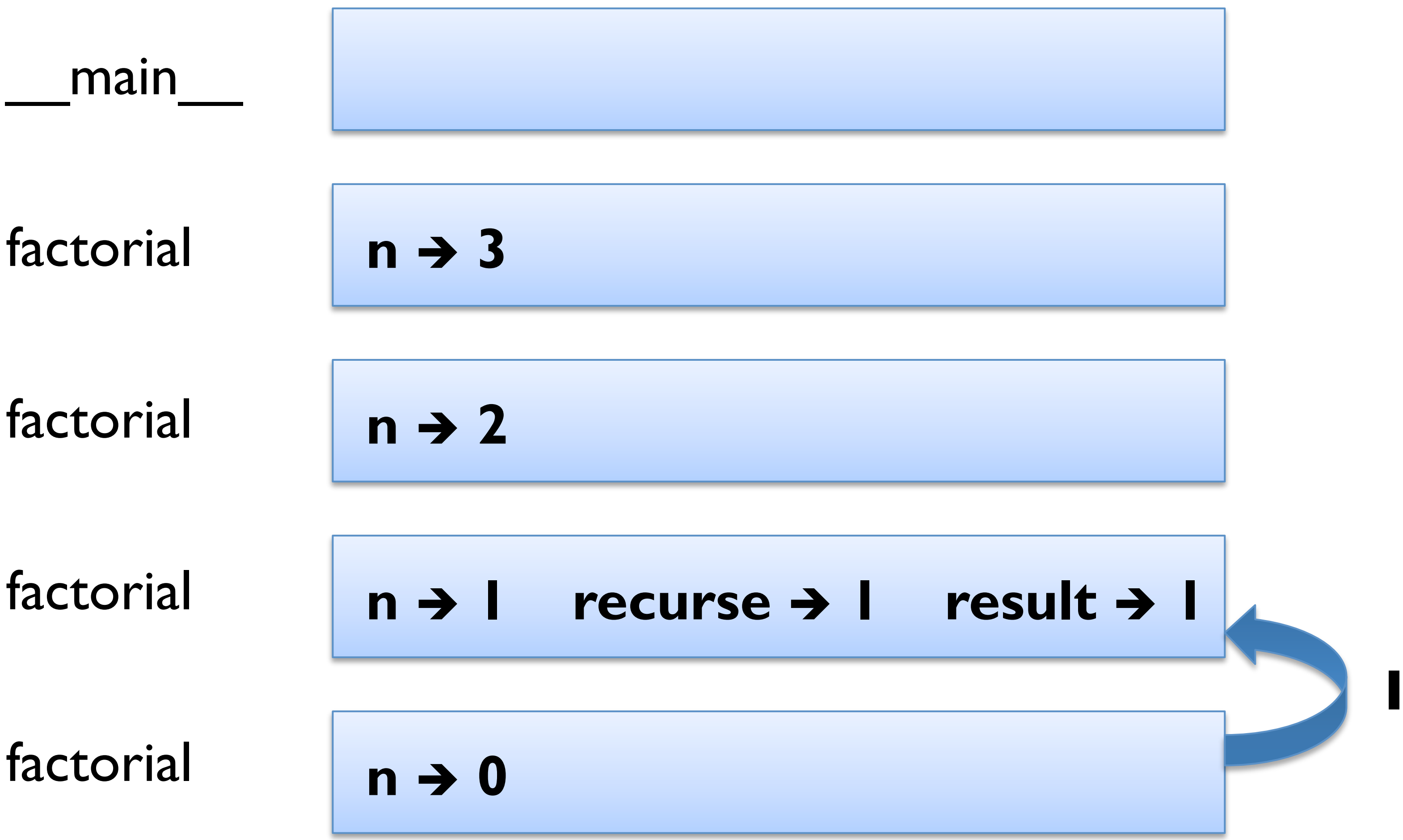
Stack Diagram for Factorial



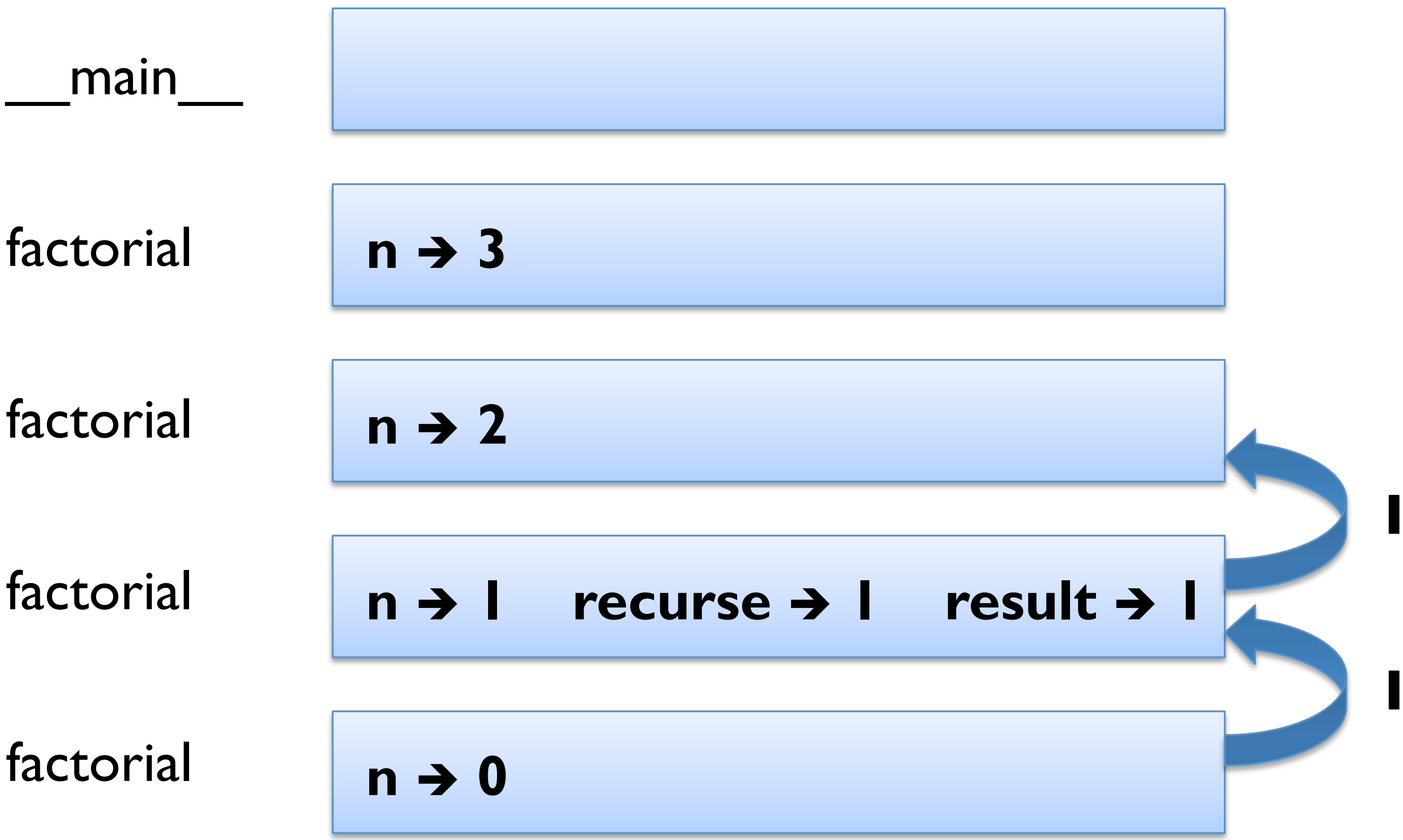
Stack Diagram for Factorial



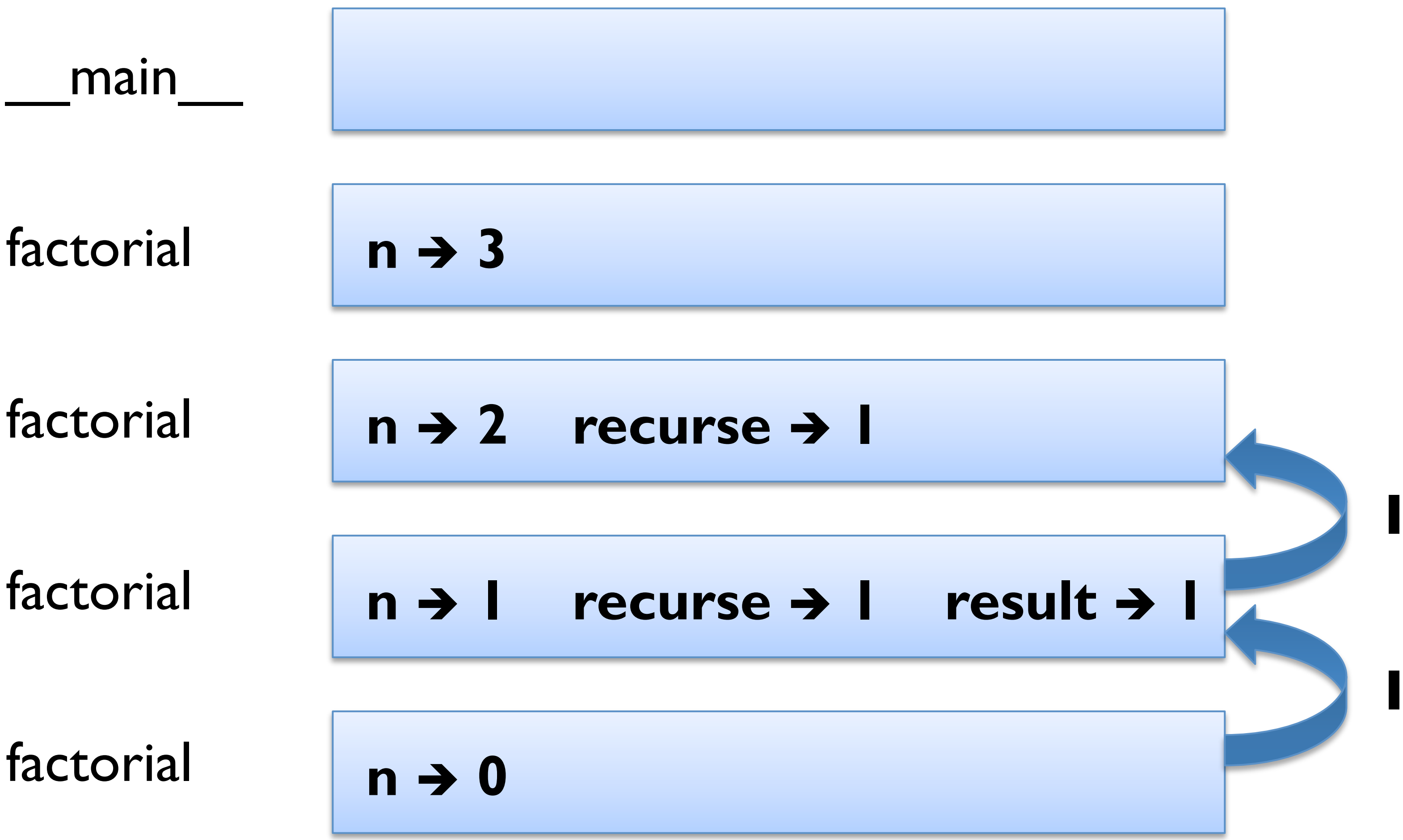
Stack Diagram for Factorial



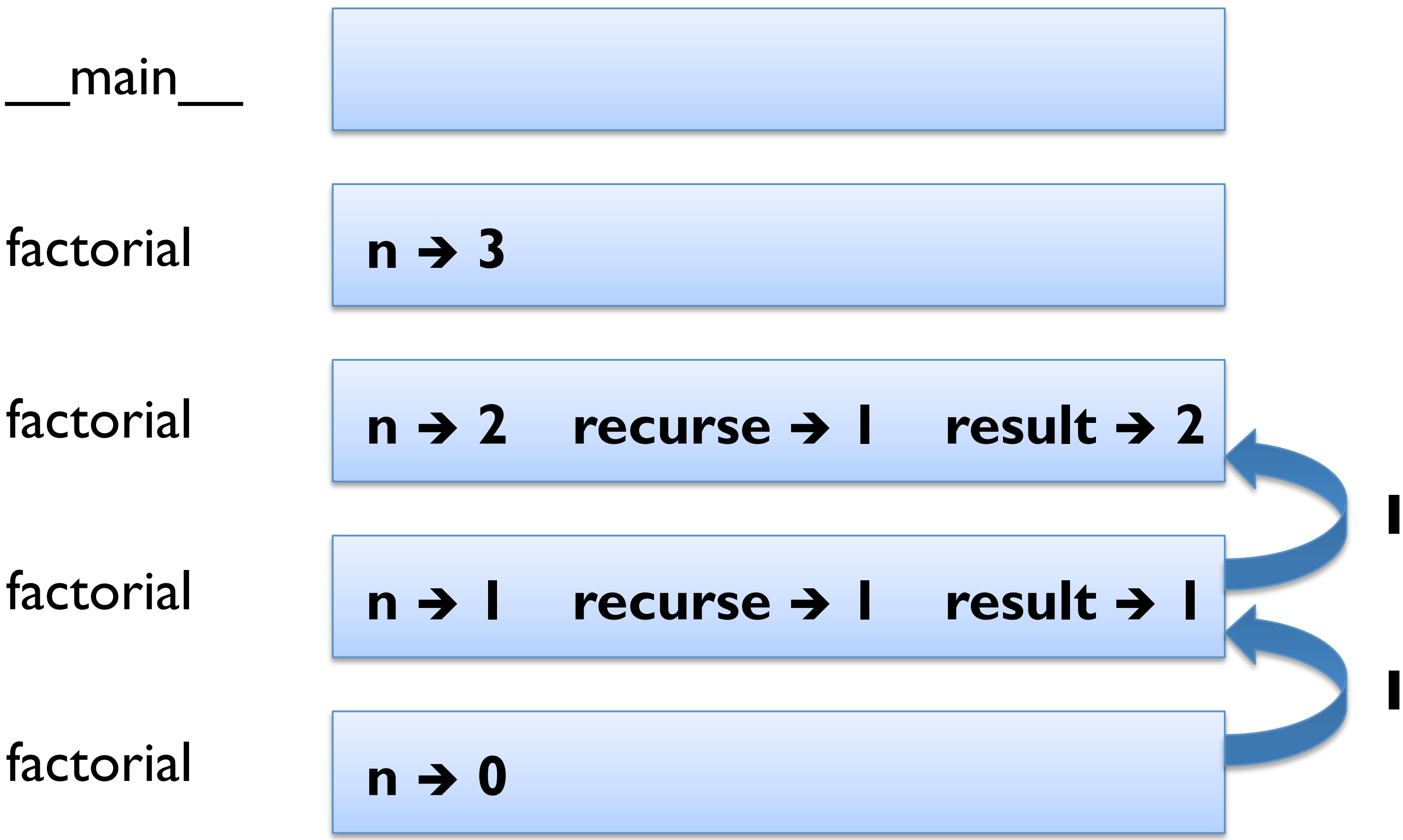
Stack Diagram for Factorial



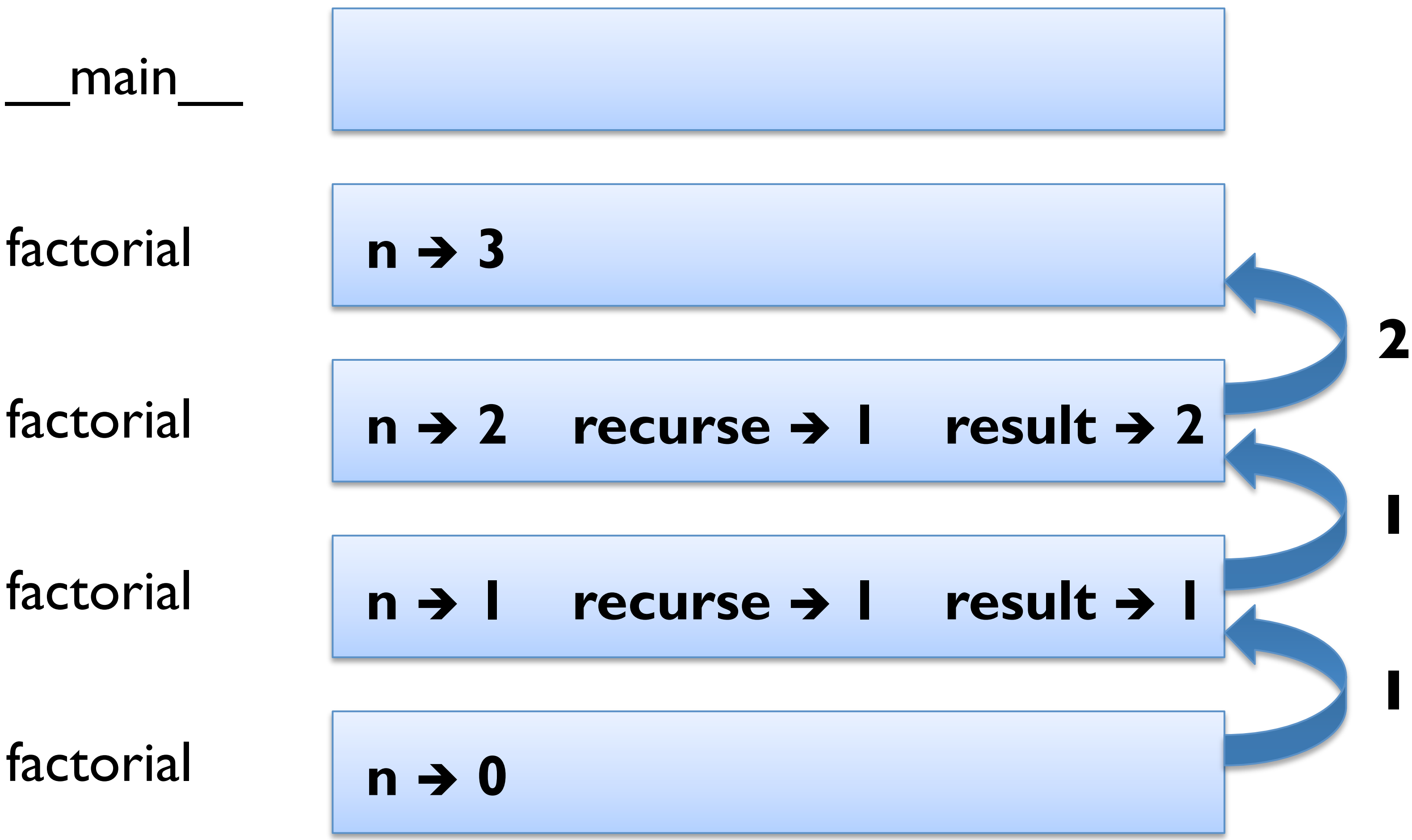
Stack Diagram for Factorial



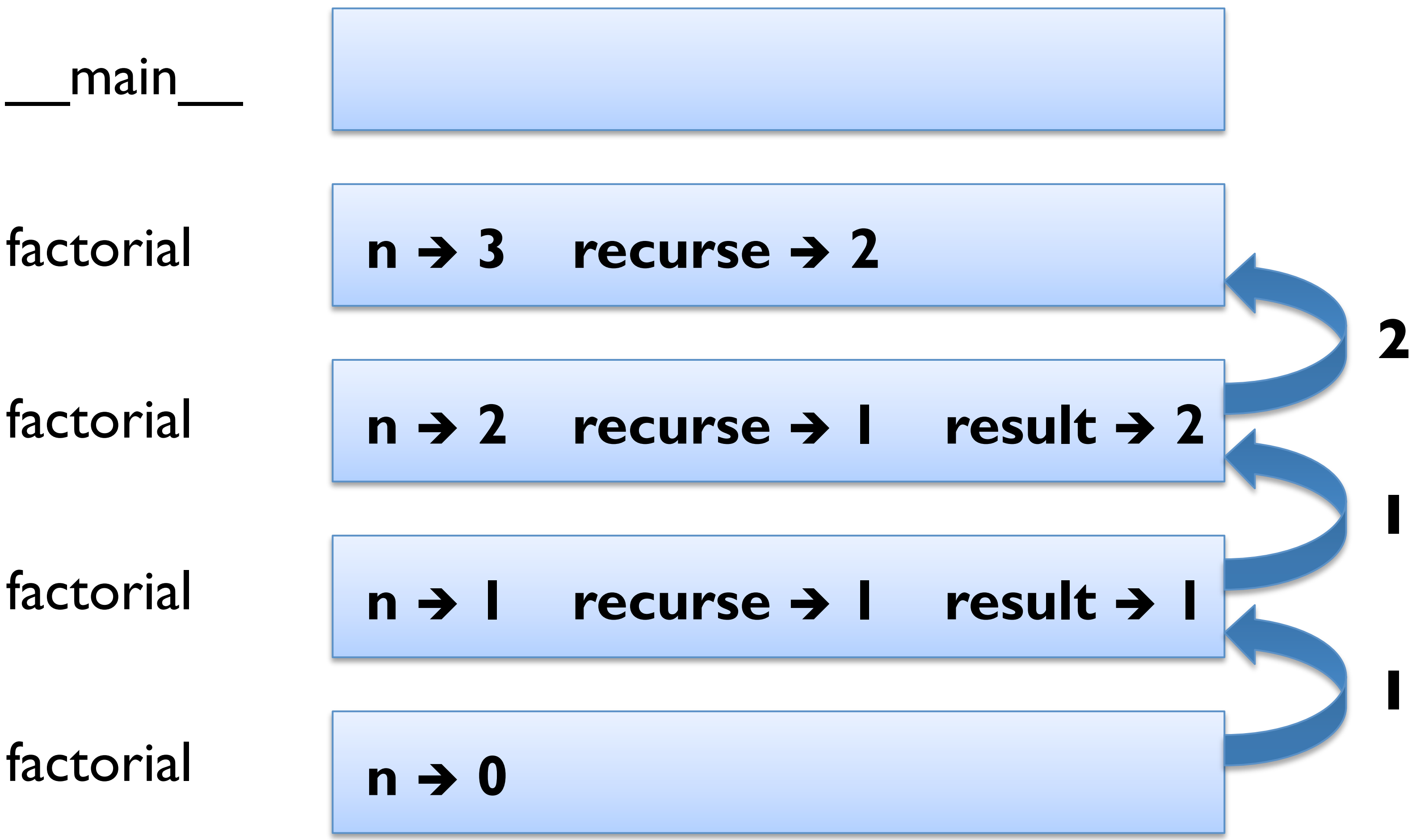
Stack Diagram for Factorial



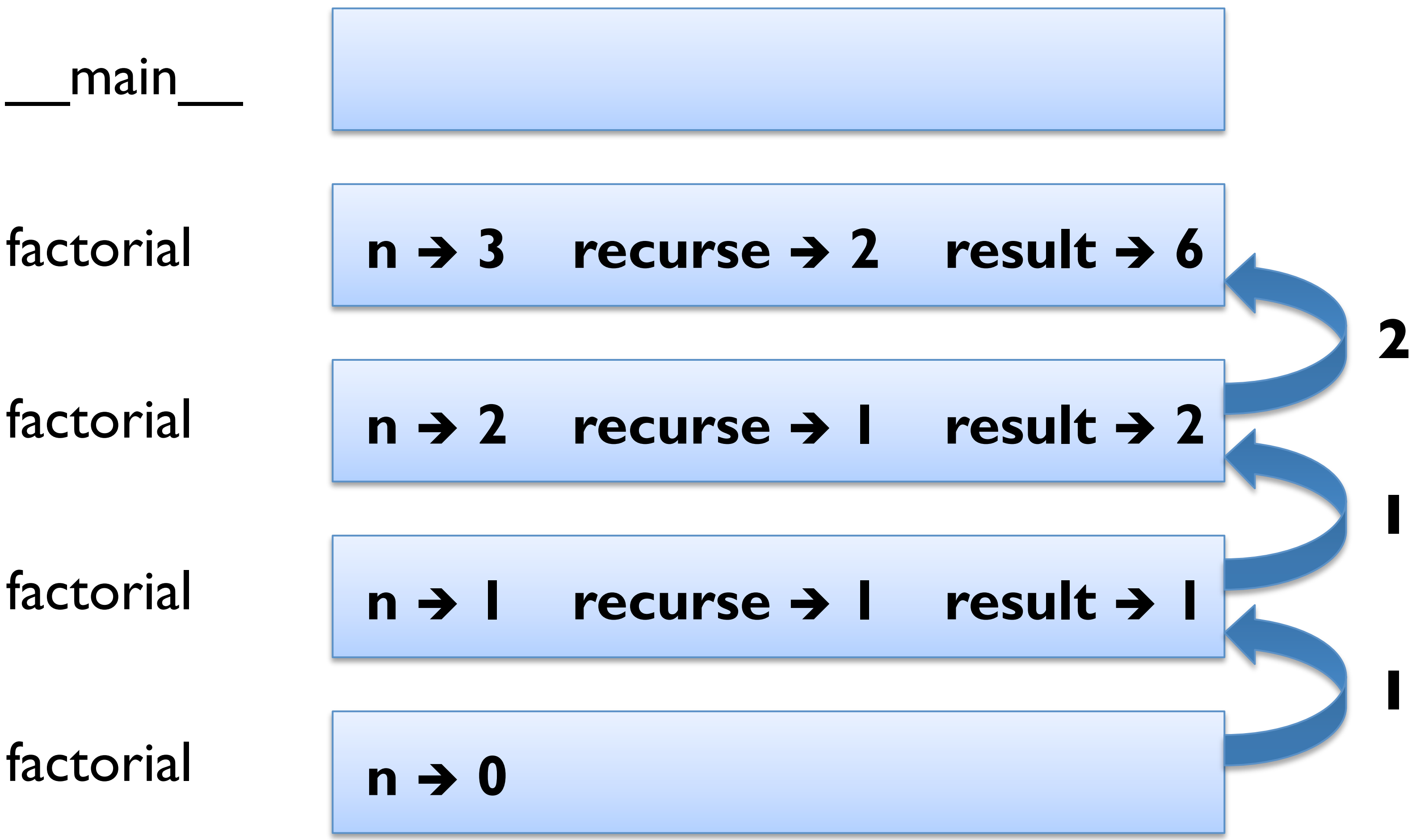
Stack Diagram for Factorial



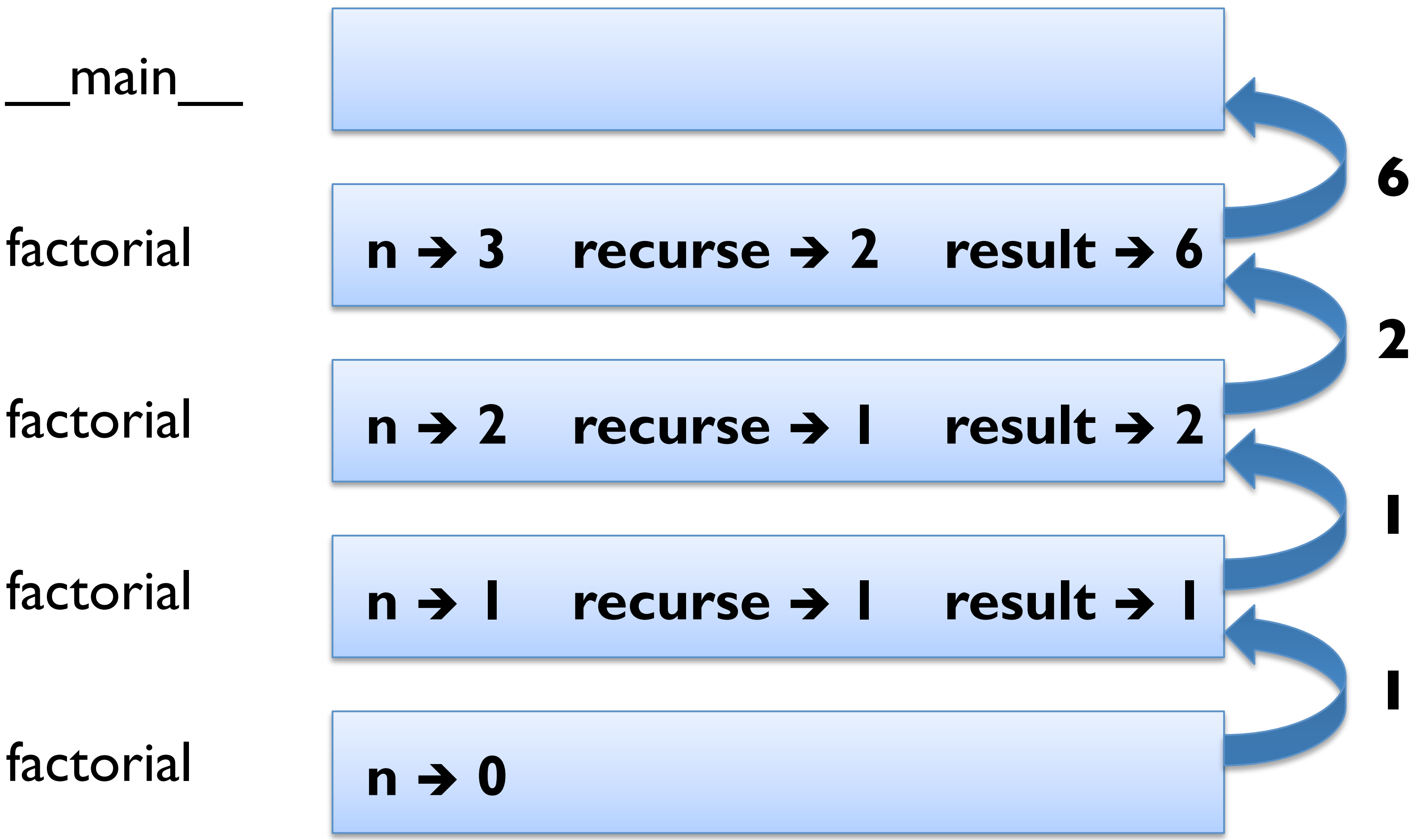
Stack Diagram for Factorial



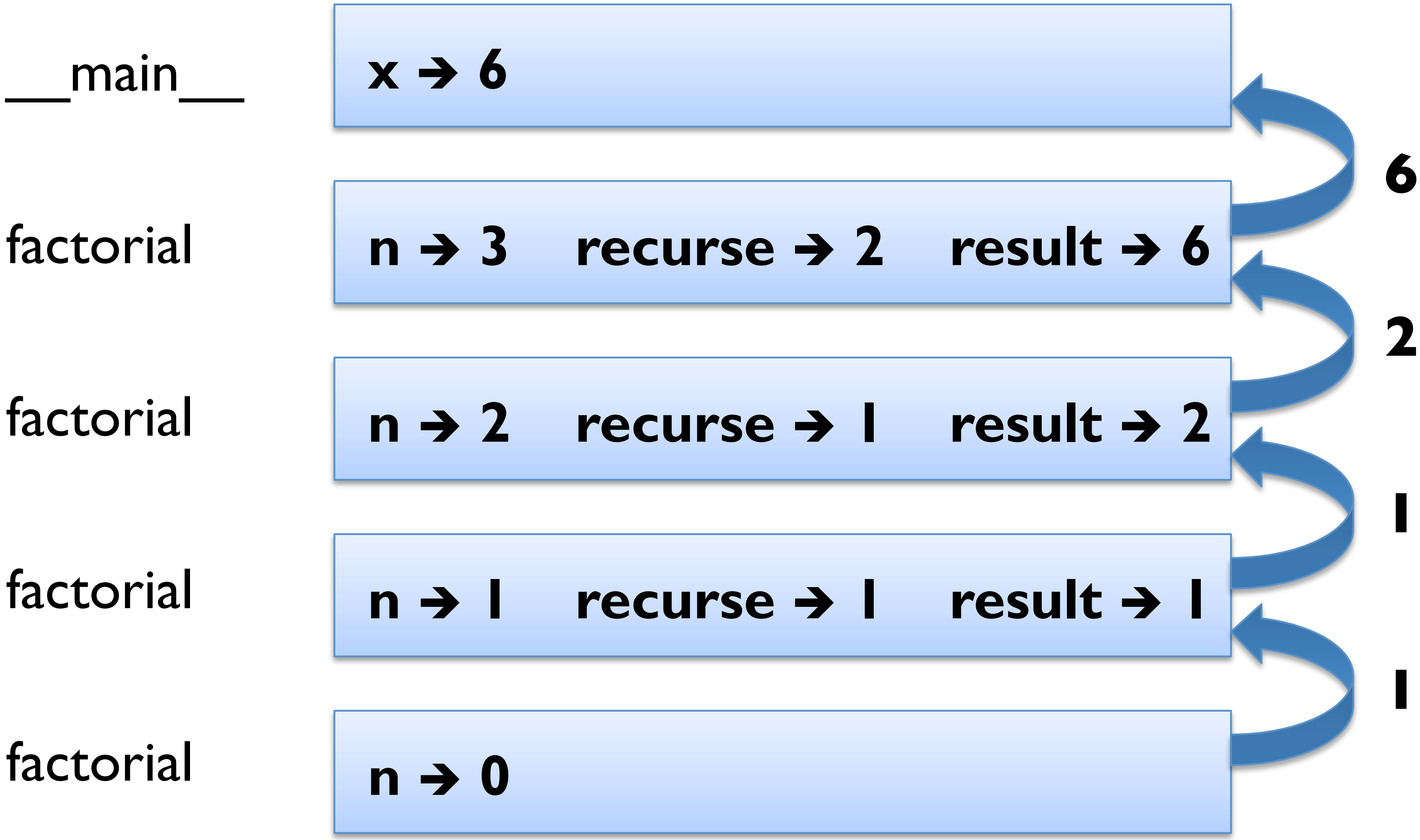
Stack Diagram for Factorial



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)
```

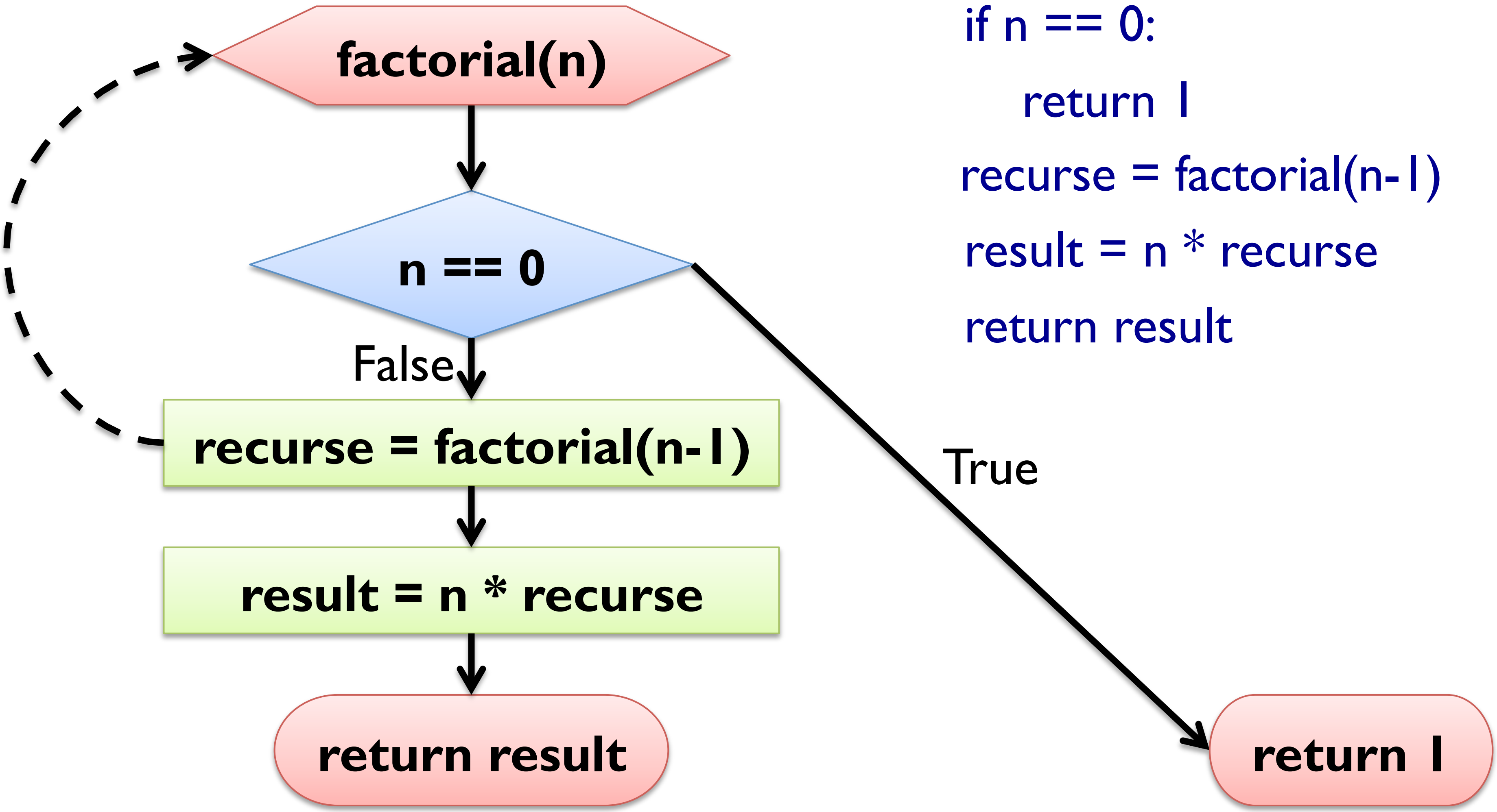
check the base case

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
```

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:

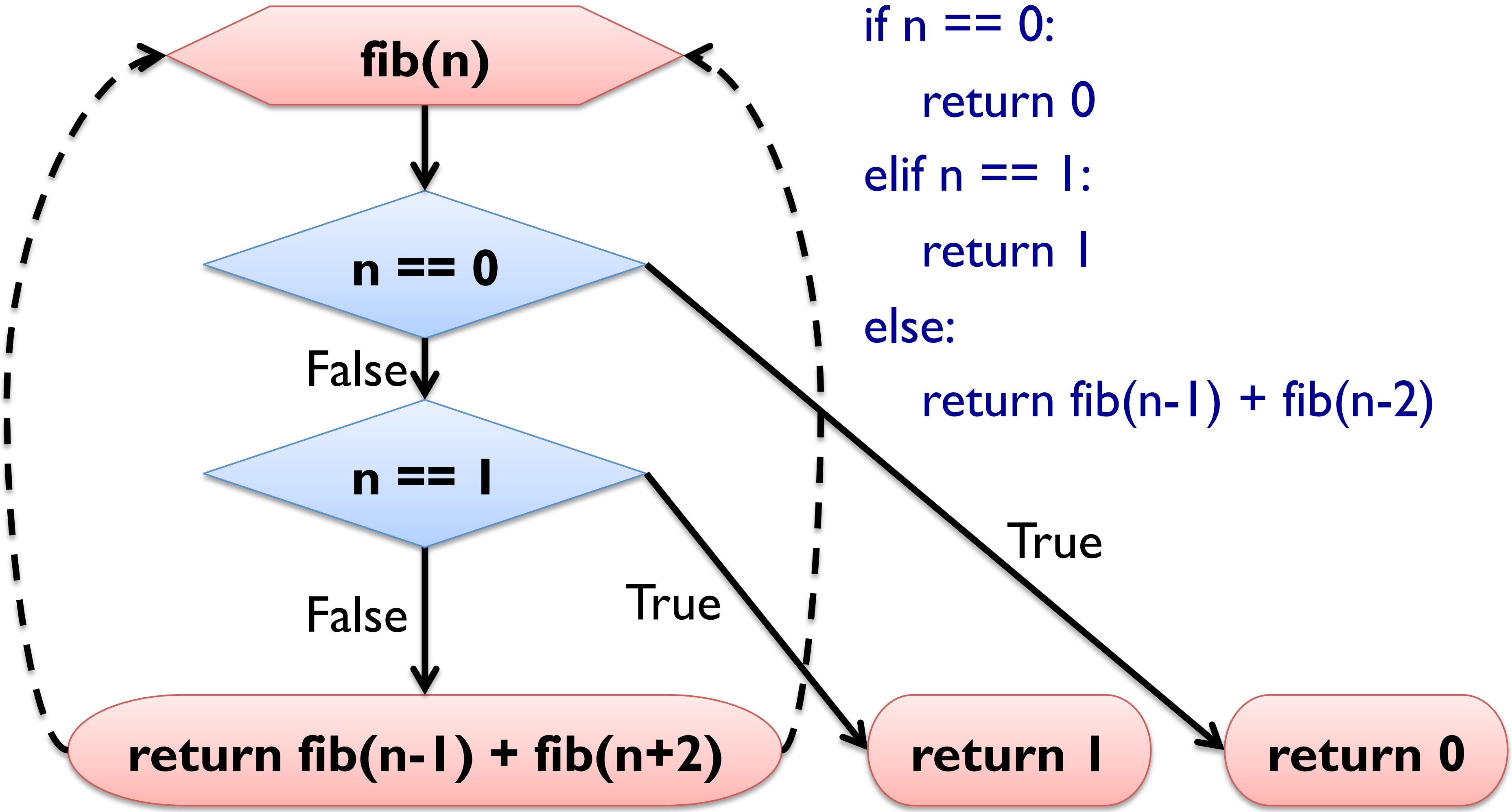
```
def fib(n):
    if n == 0:    return 0
    elif n == 1:  return 1
    else:         return fib(n-1) + fib(n-2)
```

- "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 2nd part
3. if intermediate output is wrong, apply recursively to 1st part

STRINGS

Strings as Sequences

- strings can be viewed as 0-indexed sequences
- Examples:

  "Slartibartfast"[0] == "S"

  "Slartibartfast"[1] == "l"

  "Slartibartfast"[2] == "Slartibartfast"[7]

  "Phartiphukborlz"[-1] == "z"

- grammar rule for expressions:

  <expr> => … | <expr$_1$>[<expr$_2$>]

- <expr$_1$> = expression with value of type string
- index <expr$_2$> = expression with value of type integer
- negative index counting from the back

Length of Strings

- length of a string computed by built-in function len(object)

- Example:

```
name = "Slartibartfast"
length = len(name)
print name[length-4]
```

- Note: name[length] gives runtime error

- identical to write name[len(name)-1] and name[-1]
- more general, name[len(name)-a] identical to name[-a]