



DM536

Introduction to Programming

Peter Schneider-Kamp

petersk@imada.sdu.dk

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

Python & Linux Install Party

- next week (Tuesday 14-17)
- NEW Fredagsbar (“Nedenunder”)
- 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”

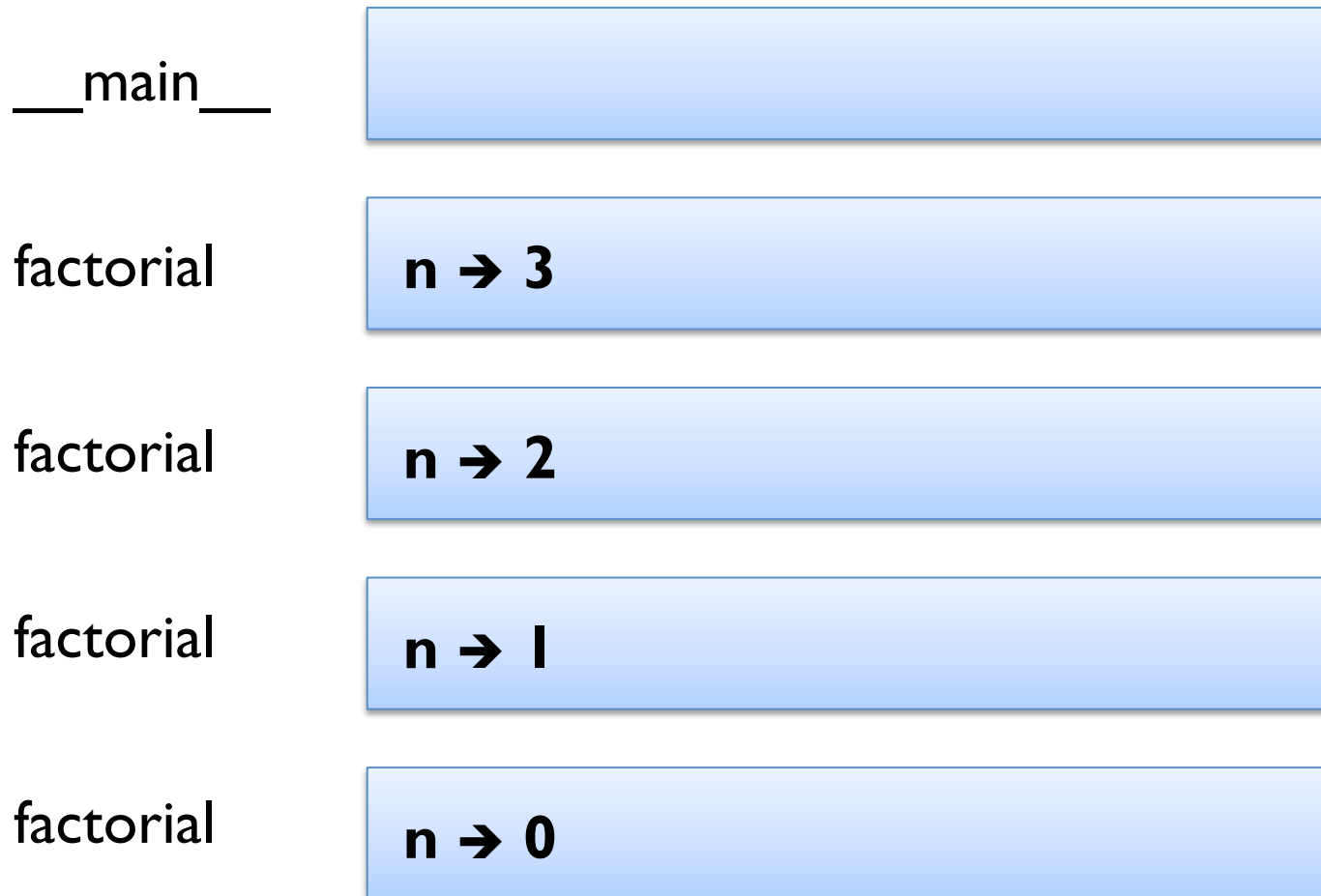
- so far we know:
 - values of type integer, float, string
 - arithmetic expressions
 - (recursive) function definitions
 - (recursive) function calls
 - conditional execution
 - input/output
- **ALL** possible programs can be written using these elements!
- we say that we have a “Turing complete” language

Factorial

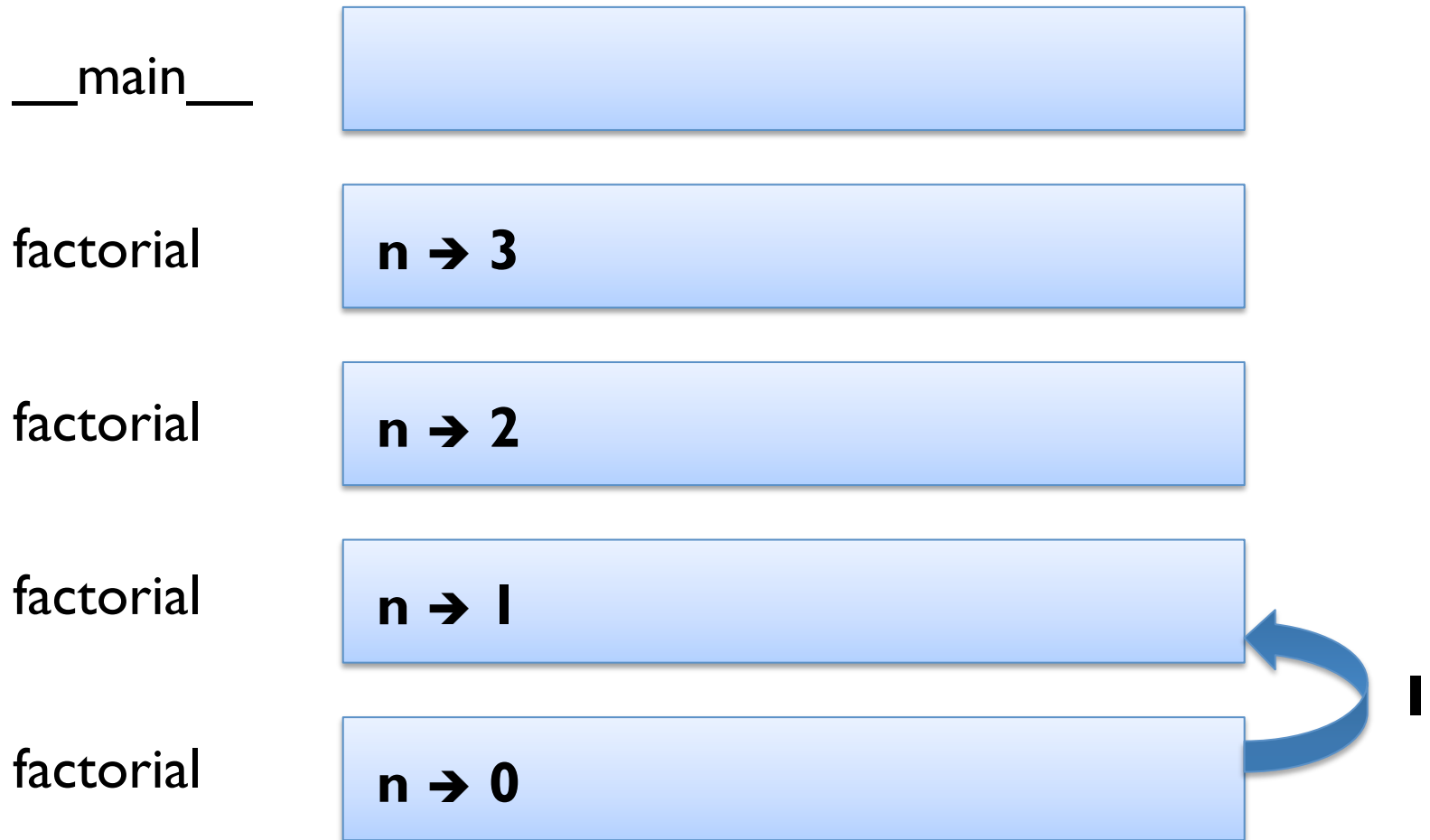
- in mathematics, the factorial function is defined by
 - $0! = 1$
 - $n! = n * (n-1)!$
- such *recursive* definitions can trivially be expressed in Python
- Example:

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

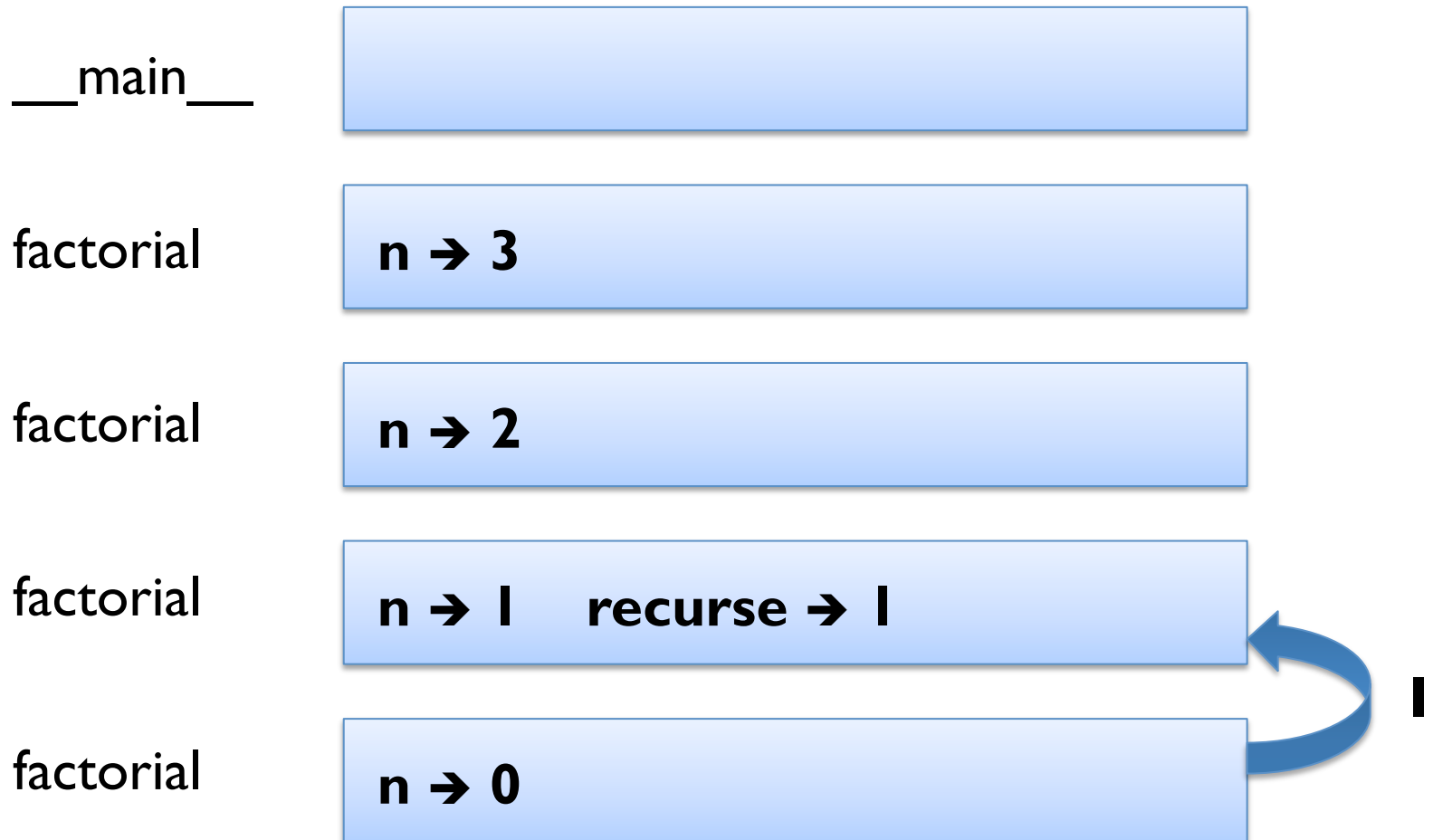
Stack Diagram for Factorial



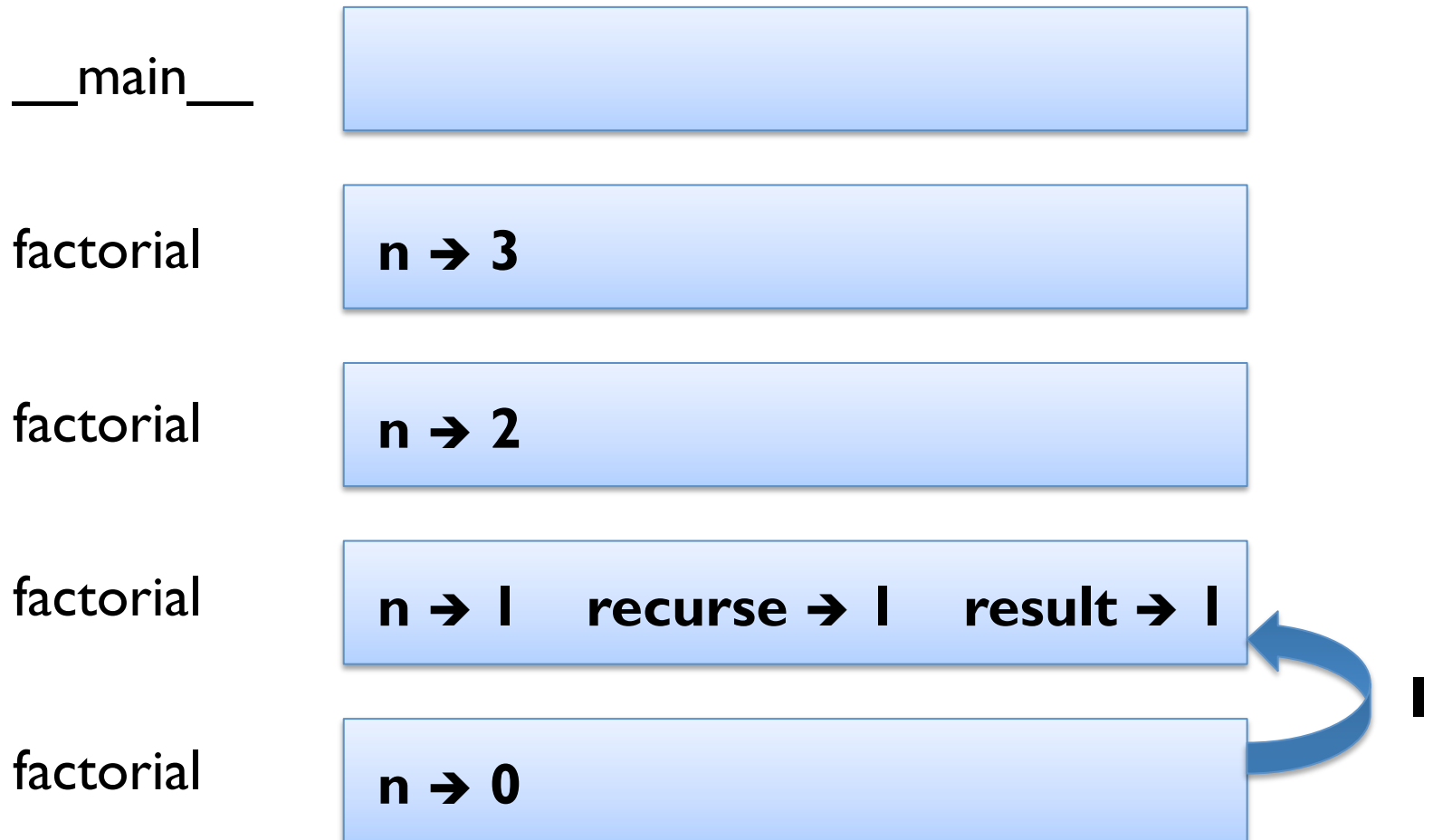
Stack Diagram for Factorial



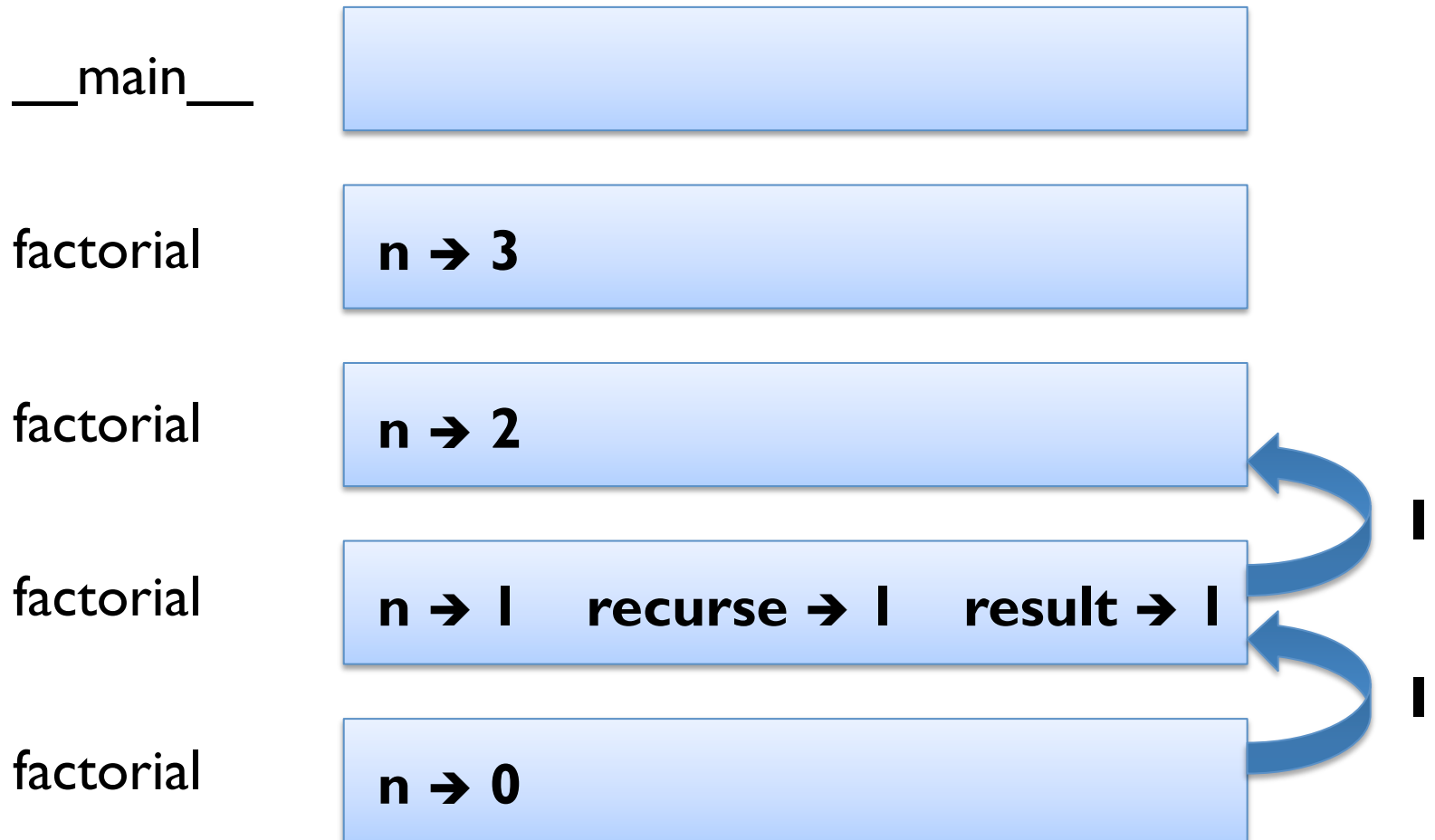
Stack Diagram for Factorial



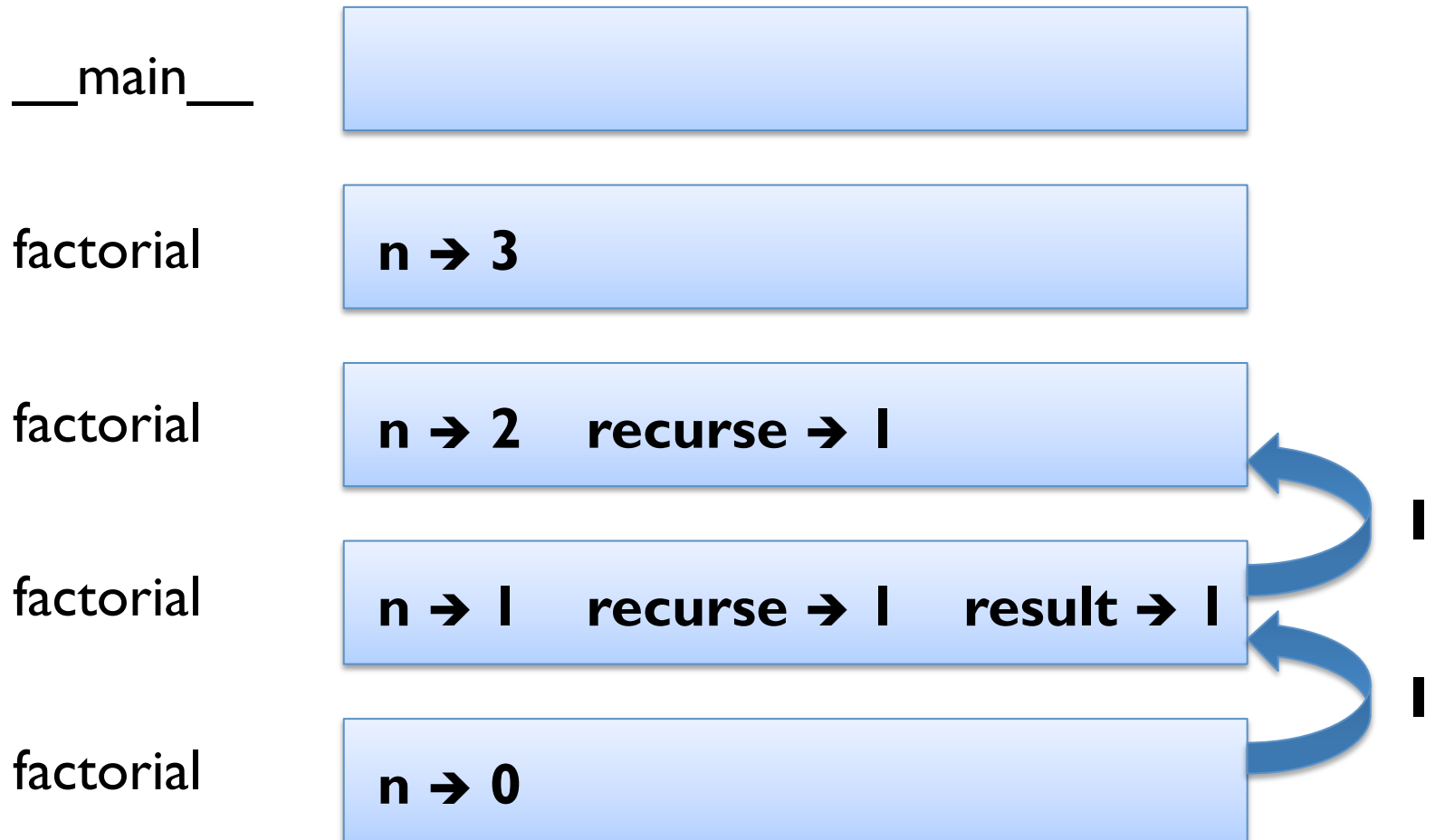
Stack Diagram for Factorial



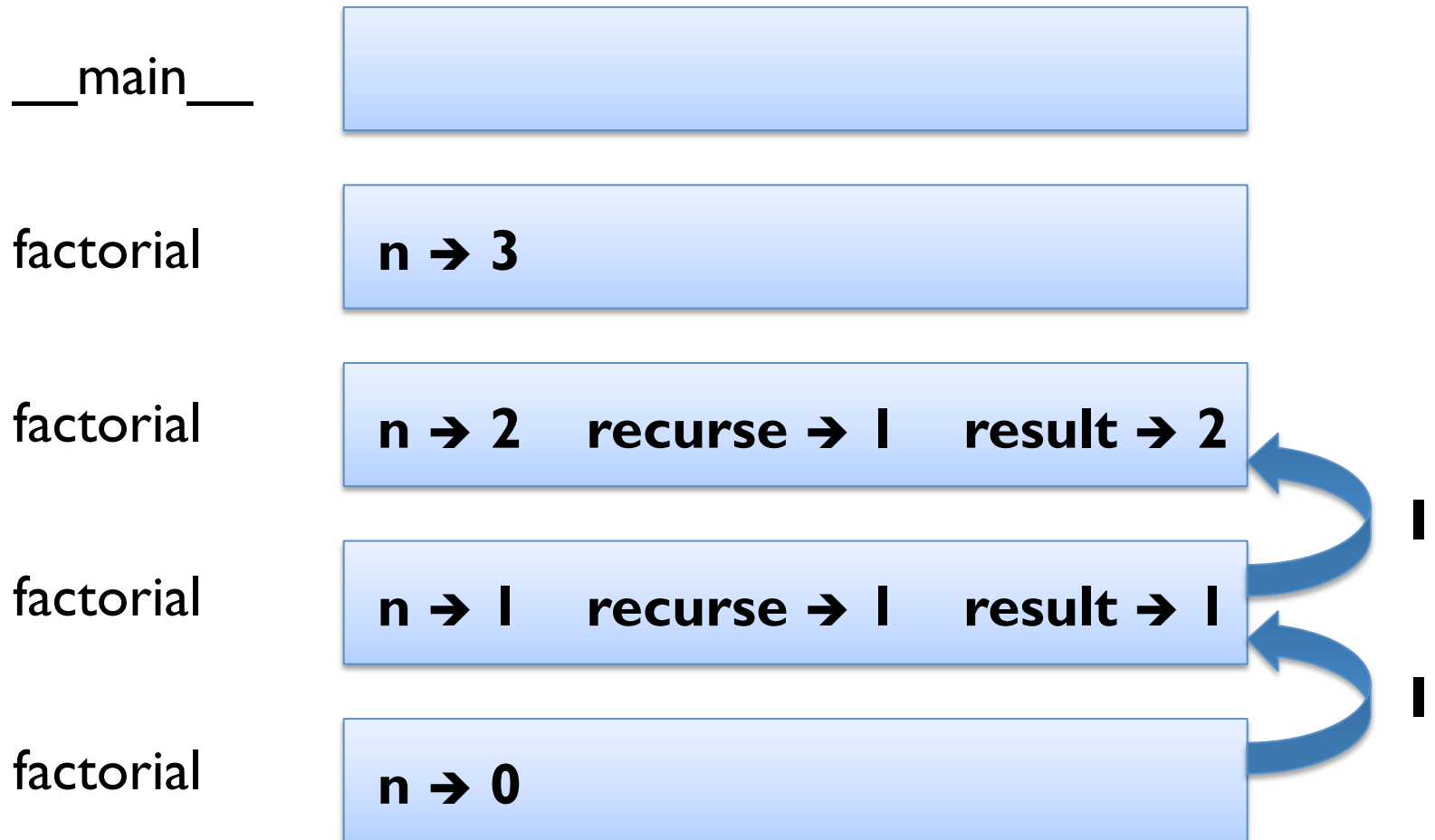
Stack Diagram for Factorial



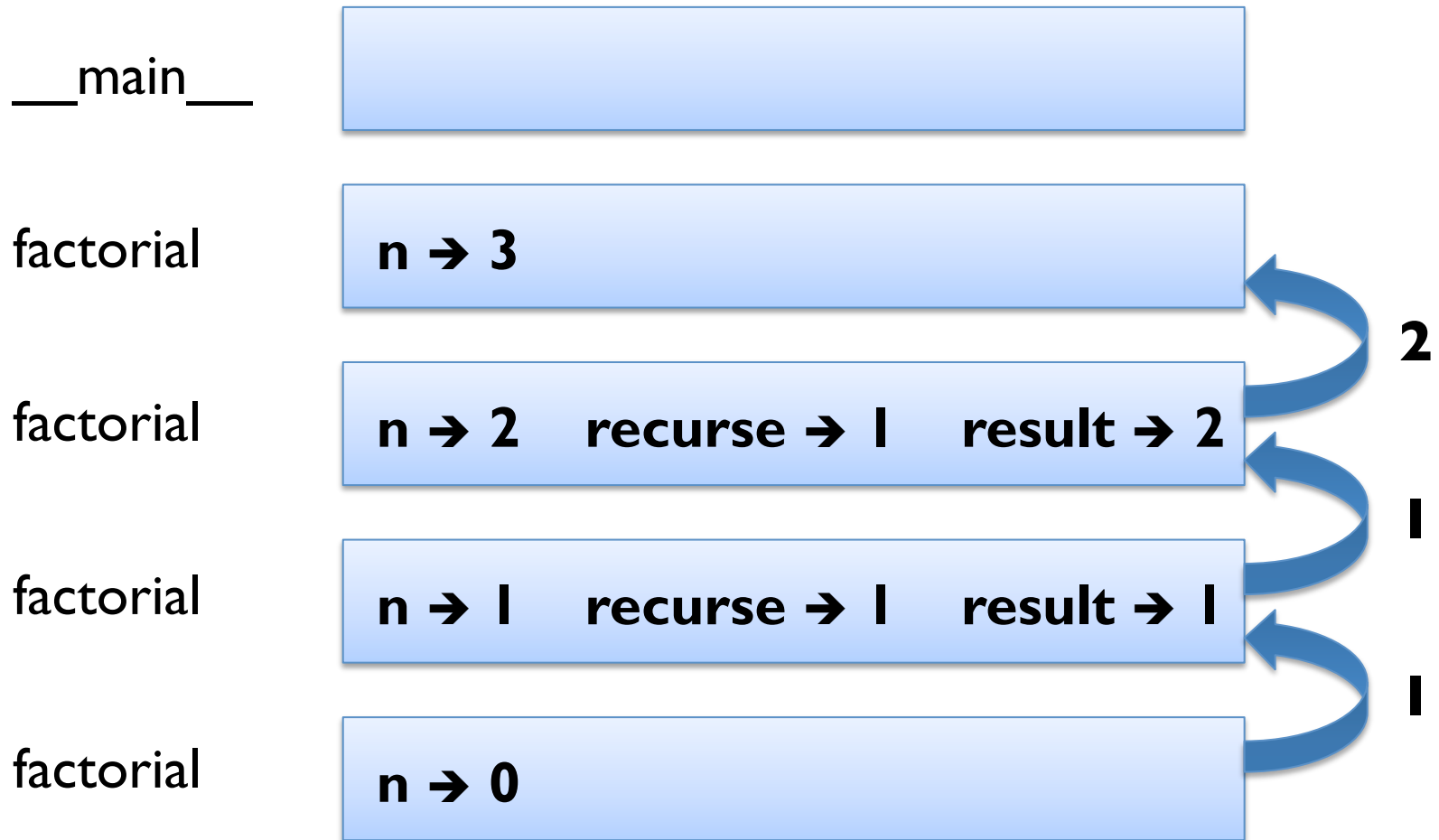
Stack Diagram for Factorial



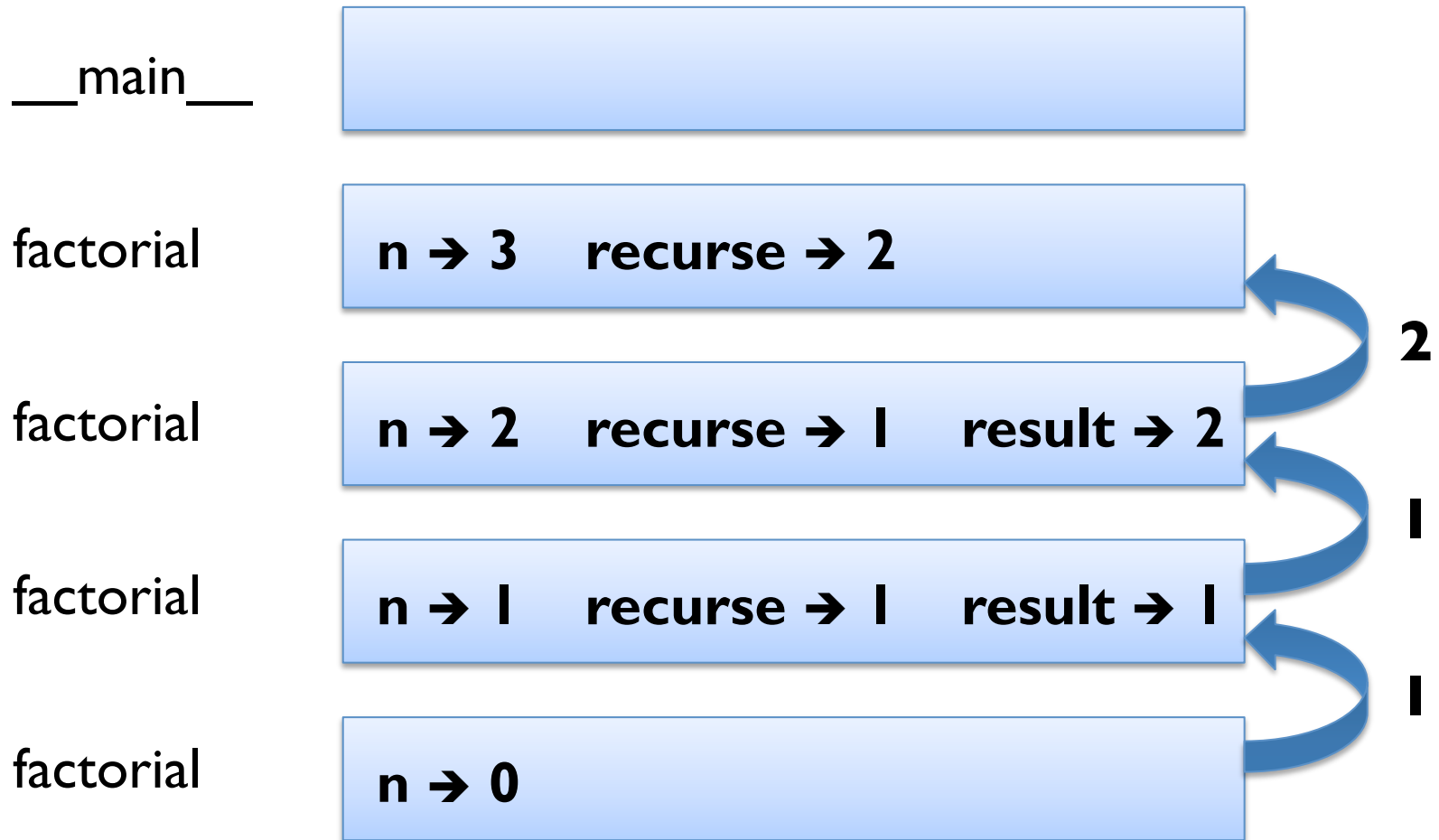
Stack Diagram for Factorial



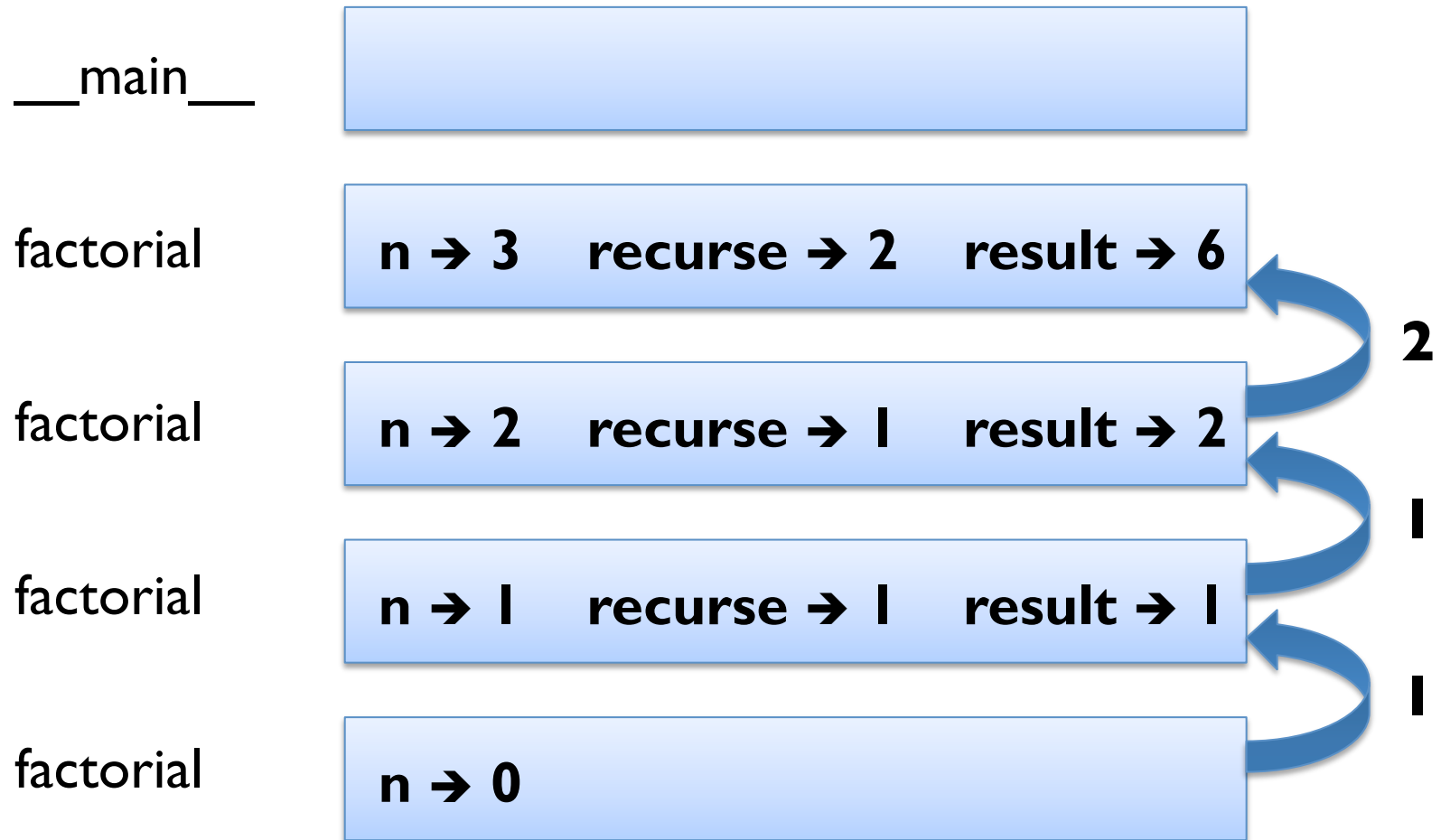
Stack Diagram for Factorial



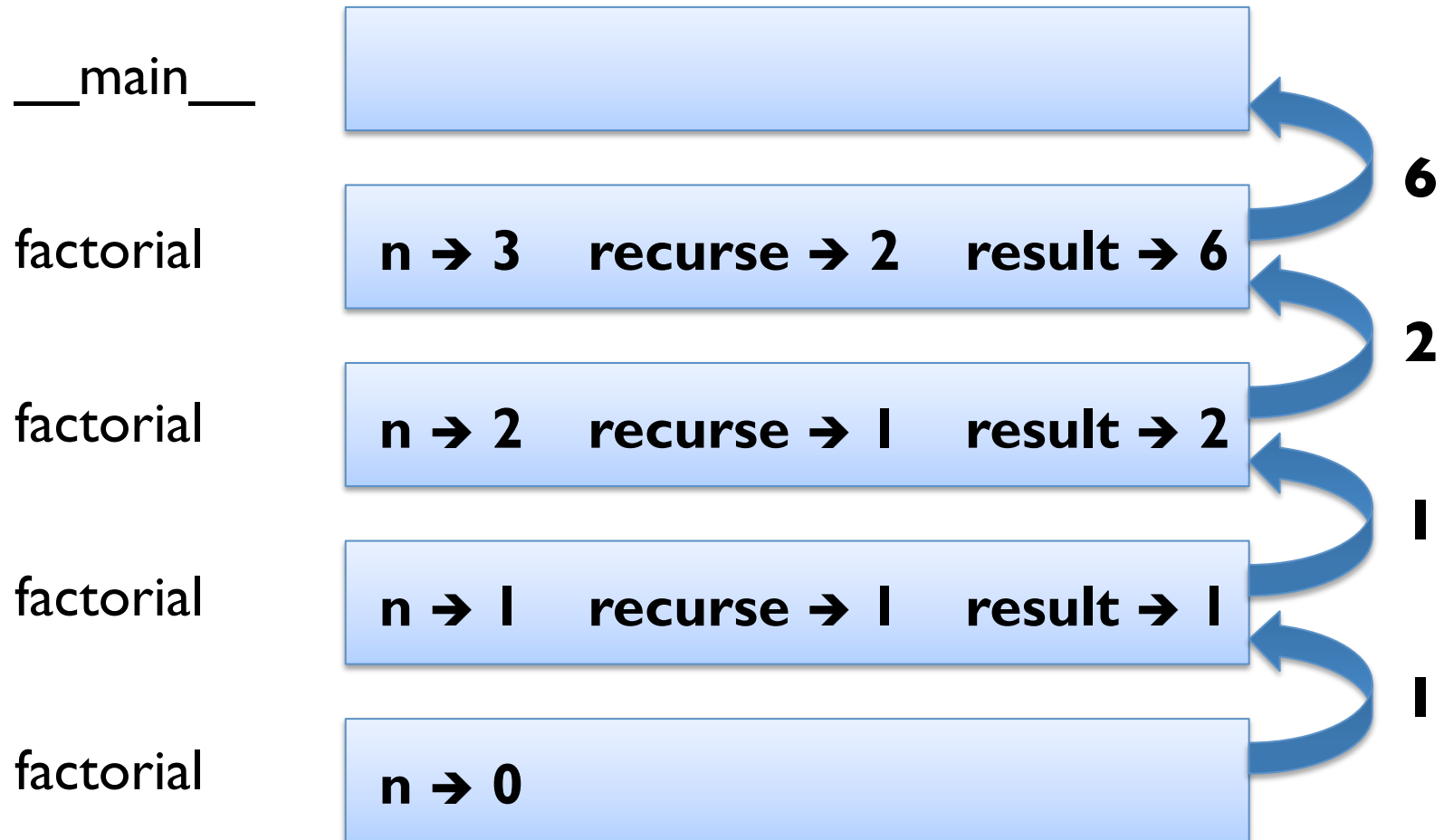
Stack Diagram for Factorial



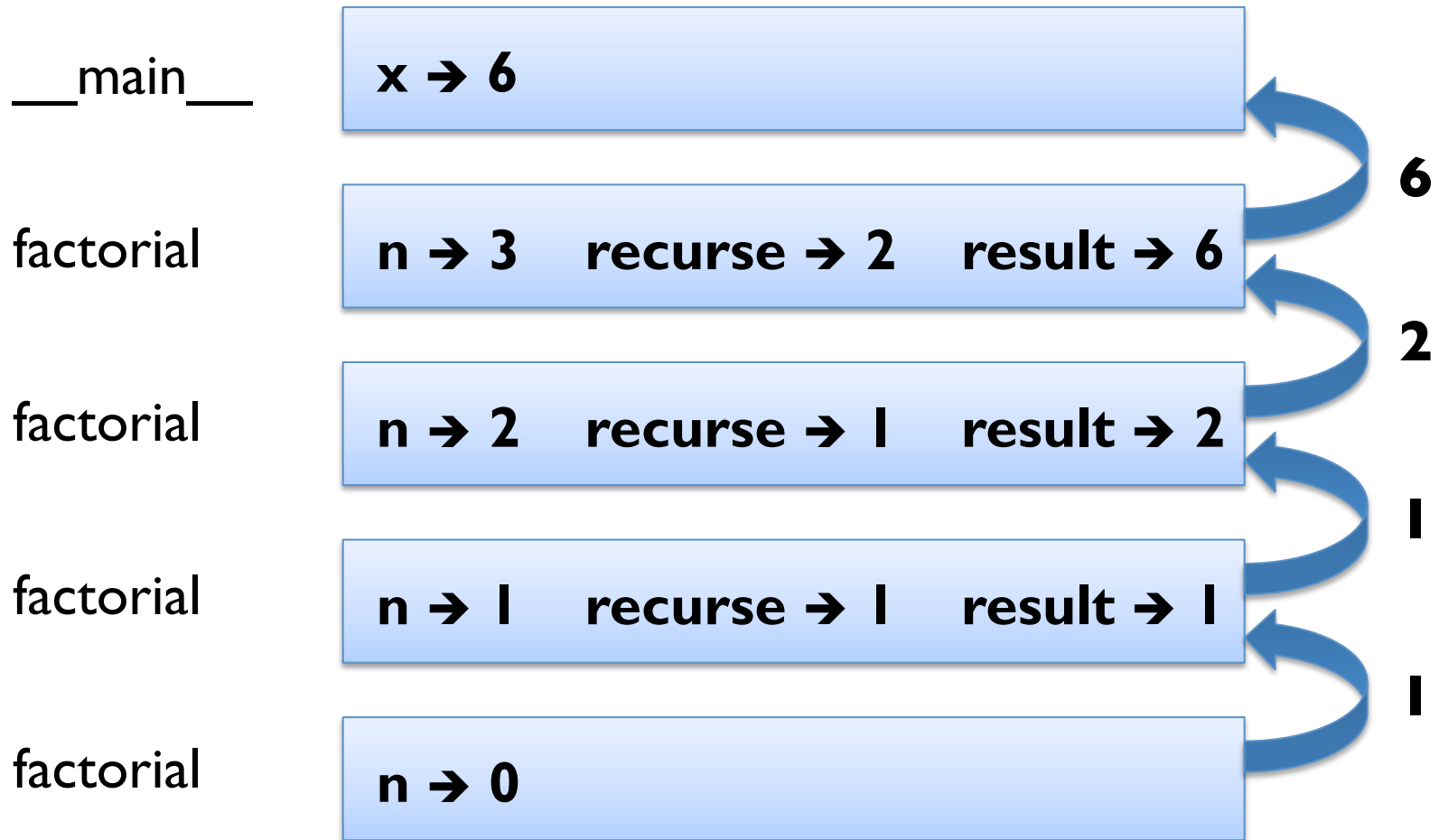
Stack Diagram for Factorial



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)
```

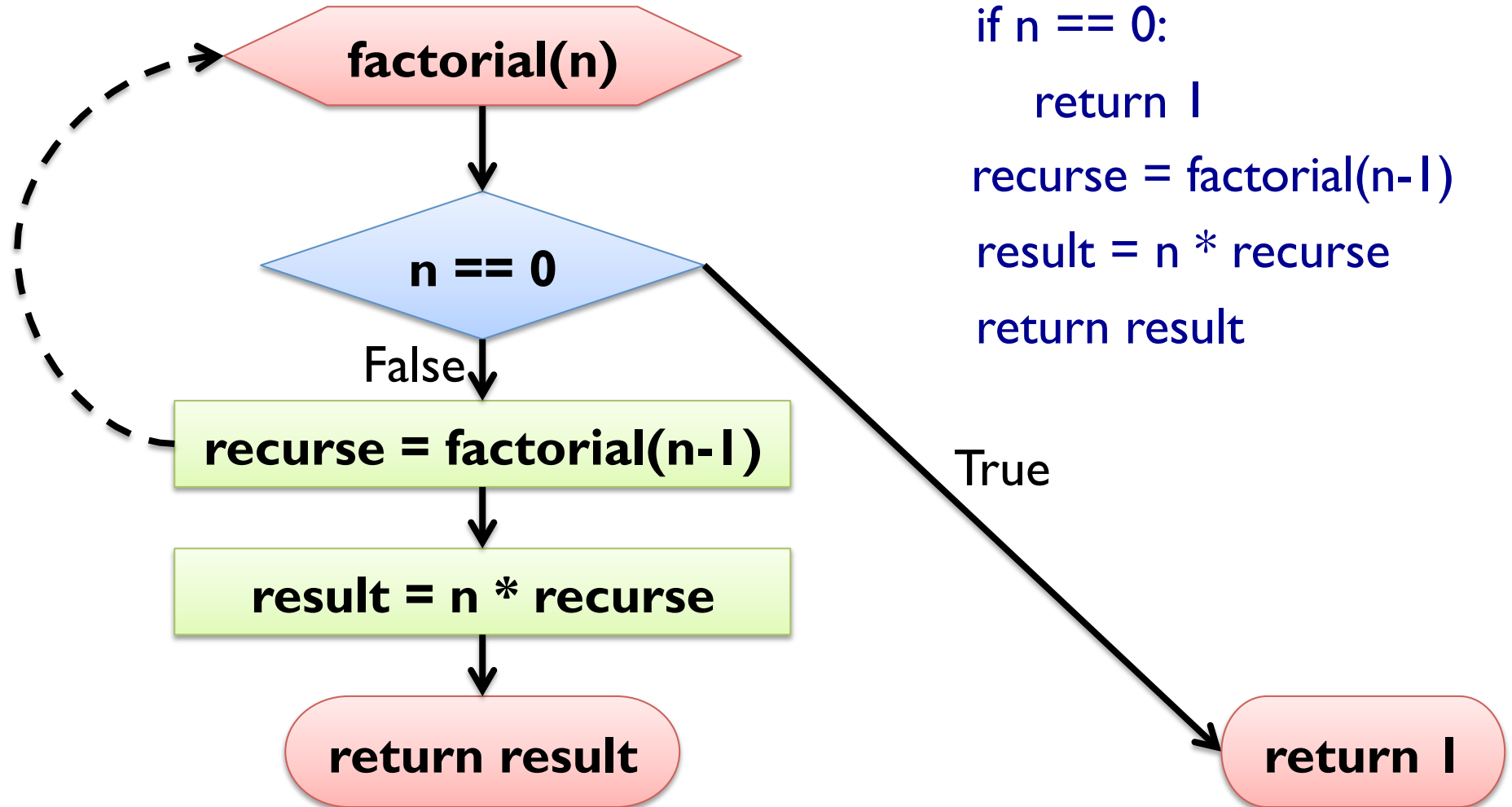
**check the
base case**

**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:
 - $\text{fib}(0) = 0$, $\text{fib}(1) = 1$, $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$
- 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:
```

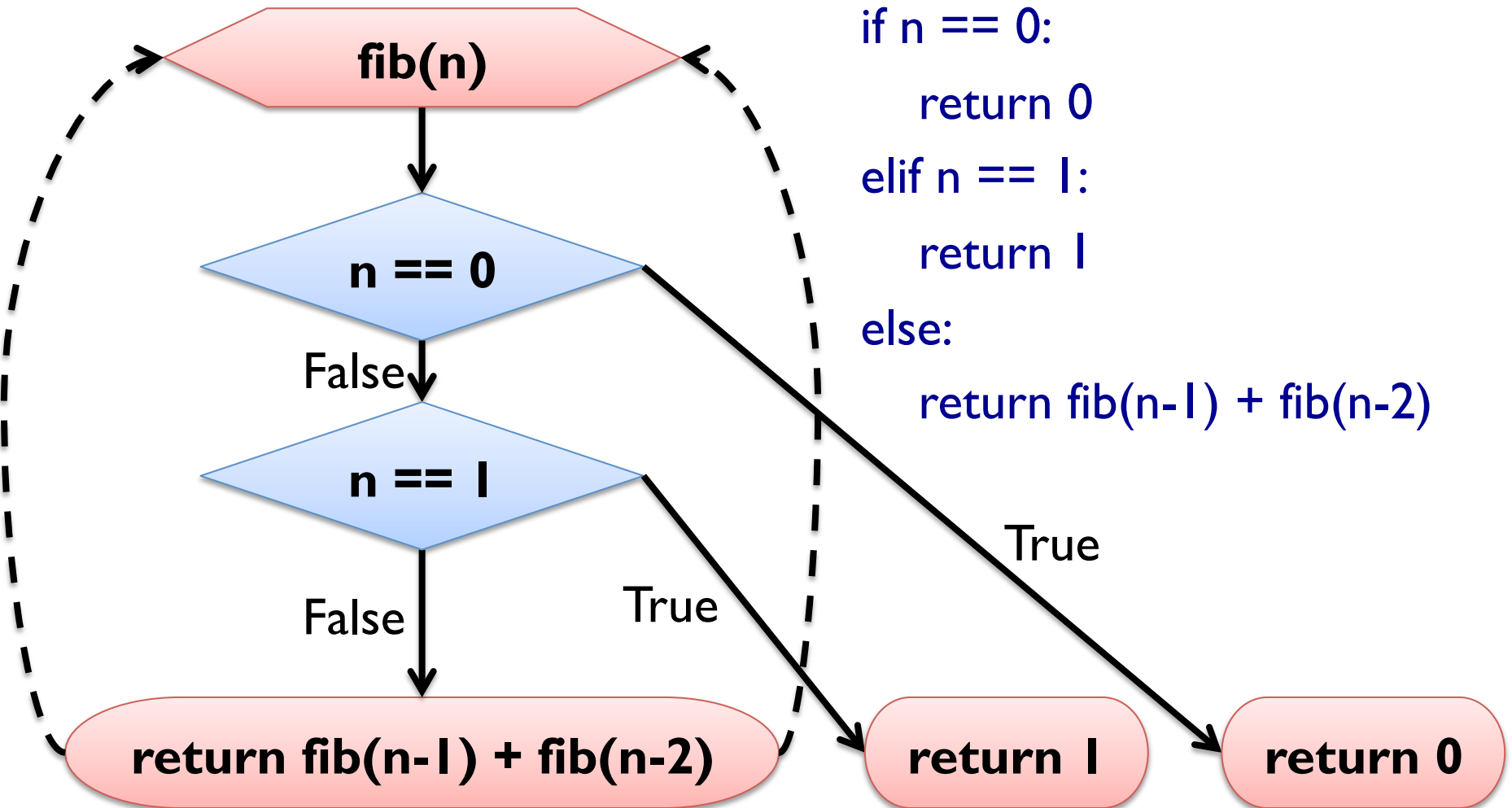
```
    return 0
```

```
  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:

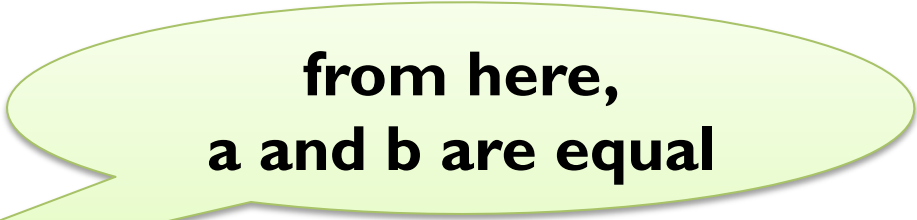
$a = 42$

...


$b = a$

...

$a = 23$



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>:`
`<instr1>; <instr2>; <instr3>`

- 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')
```

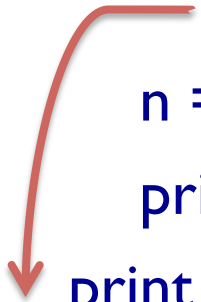
```
    if num == "exit":
```

```
        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 fl(x):     return 3*x**2 + math.sin(x)
while True:
    print xn
    xnp1 = xn - f(xn) / fl(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 fl(x):     return 3*x**2 + math.sin(x)
while True:
    print xn
    xnp1 = xn - f(xn) / fl(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

PROJECT PART I

Organizational Details

- 2 possible projects, each consisting of 2 parts
- for 1st part, you have to pick ONE
- for 2nd part, you can stay or you may switch
- projects must be done individually, so no co-operation
- you may talk about the problem and ideas how to solve them
- deliverables:
 - written 4 page report as specified in project description
 - handed in electronically as a single PDF with appendix
 - deadline: October 3, 11:00
- ENOUGH - now for the FUN part ...

Fractals and the Beauty of Nature

- geometric objects similar to themselves at different scales

- many structures in nature are fractals:

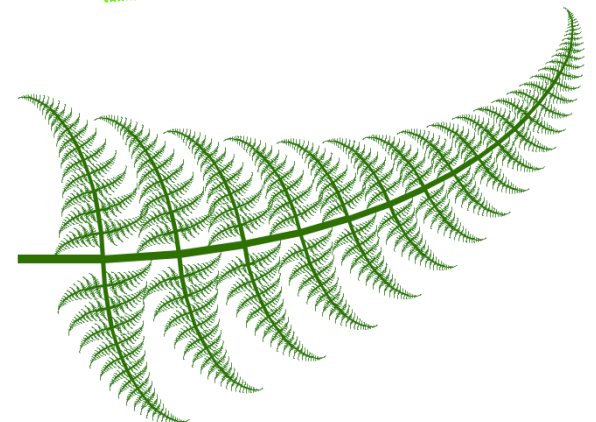
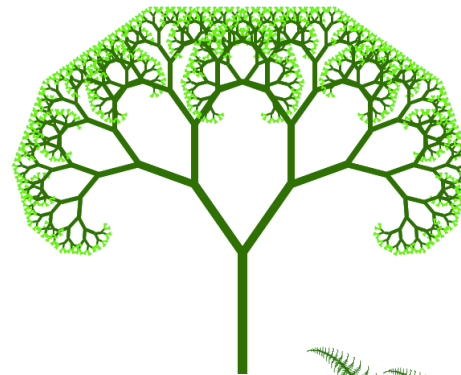
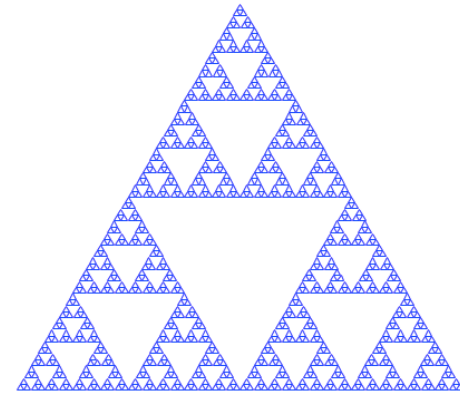
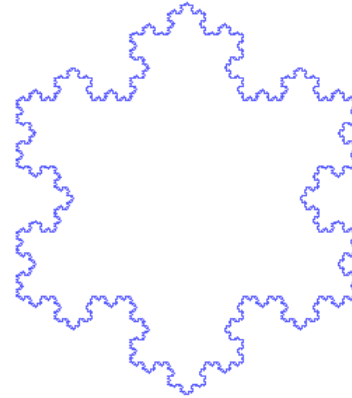
- snowflakes
- lightning
- ferns



- **Goal:** generate fractals using Swampy
- **Challenges:** Recursion, Tuning, Library Use

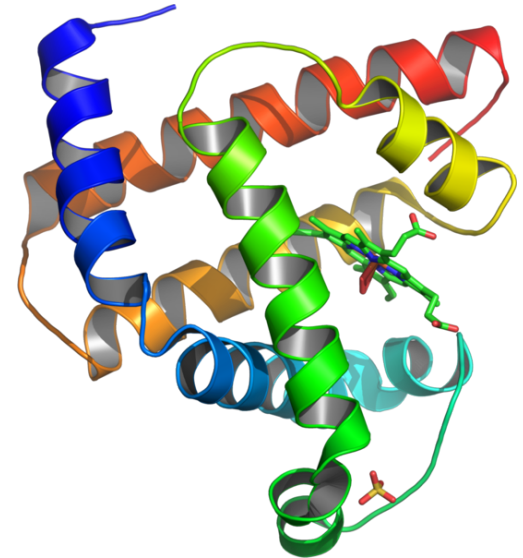
Fractals and the Beauty of Nature

- Task 0: Preparation
 - understand implementation of Koch snowflake
- Task 1: Sierpinski Triangle
 - draw fractal triangle of fixed depth
- Task 2: Binary Tree
 - draw binary trees of fixed depth
- Task 3 (optional): Fern Time
 - draw beautiful fern leaves with fixed detail



From DNA to Proteins

- proteins encoded by DNA base sequence using A, C, G, and T
- Background:
 - proteins are sequences of amino acids
 - amino acids encoded using three bases
 - chromosomes given as base sequences
- **Goal:** assemble and analyze sequences from files
- **Challenges:** File Handling, String and List Methods, Iteration



From DNA to Proteins

- Task 0: Preparation
 - download human DNA sequence and take a look at it
- Task 1: Assembling the Sequence
 - clean up the sequence and assemble it into one string
- Task 2: Finding Starting Points
 - find positions in string where ATG closely follows TATAAA
- Task 3: Finding End Points
 - find one of the potential end markers (TAG, TAA, TGA)
- Task 4 (optional): Potential Proteins without TATA Boxes
 - analysis of overlaps in encoded proteins

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:

$\langle \text{expr} \rangle \Rightarrow \dots \mid \langle \text{expr}_1 \rangle [\langle \text{expr}_2 \rangle]$

- $\langle \text{expr}_1 \rangle$ = expression with value of type string
- index $\langle \text{expr}_2 \rangle$ = 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]`

Traversing with While Loop

- many operations go through string one character at a time
- this can be accomplished using
 - a while loop,
 - an integer variable, and
 - index access to the string
- Example:

```
index = 0
```

```
while index < len(name):
```

```
    letter = name[index]
```

```
    print letter
```

```
    index = index + 1
```

Traversing with For Loop

- many operations go through string one character at a time
- this can be accomplished *easier* using
 - a for loop and
 - a string variable
- Example:
 - for letter in name:
 - print letter

Generating Duck Names

- What does the following code do?

```
prefix = "R"  
infixes = "iau"  
suffix = "p"  
for infix in infixes:  
    print prefix + infix + suffix
```

- ... and greetings from Andebyen!

String Slices

- slice = part of a string

- Example 1:

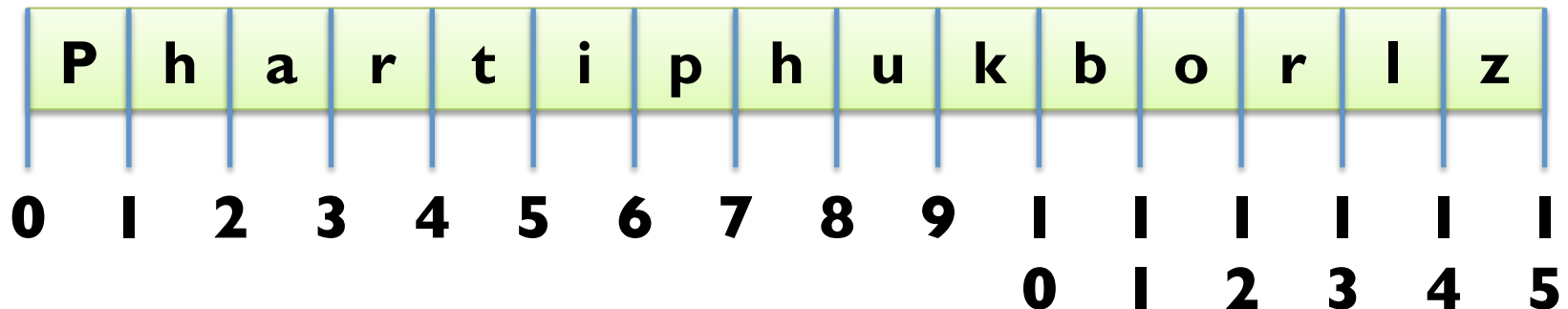
```
name = "Phartiphukborlz"
```

```
print name[6:10]
```

- one can use negative indices:

```
name[6:-5] == name[6:len(name)-5]
```

- view string with indices before letters:



String Slices

- slice = part of a string

- Example 2:

```
name = "Phartiphukborlz"
```

```
print name[6:6]      # empty string has length 0
```

```
print name[:6]      # no left index = 0
```

```
print name[6:]      # no right index = len(name)
```

```
print name[:]      # guess ;)
```

- view string with indices before letters:

