BIOS6642

More on Functions

Fuyong Xing

Department of Biostatistics and Informatics

Colorado School of Public Health

University of Colorado Anschutz Medical Campus

Outline

- **Keyword arguments**
- Recursion
- Functions as data
- Lambda expression
- Packaging functions into modules

Keyword arguments

- The following function specifies formal parameters named *a, b*, and *c.*

Codes

```
def process(a, b, c):
    print('a =', a, ' b =', b, ' c =', c)

x = 14
process(2, x, 10)
```

Output

```
a = 2  b = 14  c = 10
```

Keyword arguments

- The following function specifies formal parameters named *a, b*, and *c.*

Codes

```
def process(a, b, c):
    print('a =', a, ' b =', b, ' c =', c)

x = 14
process(2, x, 10)
```

Output

```
a = 2  b = 14  c = 10
```

- By default, the association of actual parameter to formal parameter during a function invocation is strictly positional.

Keyword arguments

- The caller can pass its actual parameters in any order using **keyword arguments**.

Codes

```
def process(a, b, c):
    print('a =', a, ' b =', b, ' c =', c)



x = 14
process(1, 2, 3)
process(a=10, b=20, c=30)
process(b=200, c=300, a=100)
process(c=3000, a=1000, b=2000)
process(10000, c=30000, b=20000)
```

Keyword arguments

- The caller can pass its actual parameters in any order using **keyword arguments**.

Output

```
a = 1   b = 2   c = 3
a = 10   b = 20   c = 30
a = 100   b = 200   c = 300
a = 1000   b = 2000   c = 3000
a = 10000   b = 20000   c = 30000
```

Keyword arguments

- The caller can pass its actual parameters in any order using **keyword arguments.**
- Both keywords and non-keyword arguments may appear in the same call. In the mixed parameter calls, all non-keyword arguments must appear before any keyword arguments.

```
process(10000, c=30000, b=20000)
```

Outline

- Keyword arguments
- **Recursion**
- Functions as data
- Lambda expression
- Packaging functions into modules

Recursion

- One function can call another in its body, and it is also legal to call itself.
- **Recursion**: The process of calling the function that is currently executing.

```
def countdown(n):
    if n <= 0:
        print('Blastoff!')
    else:
        print(n)
        countdown(n-1)
```

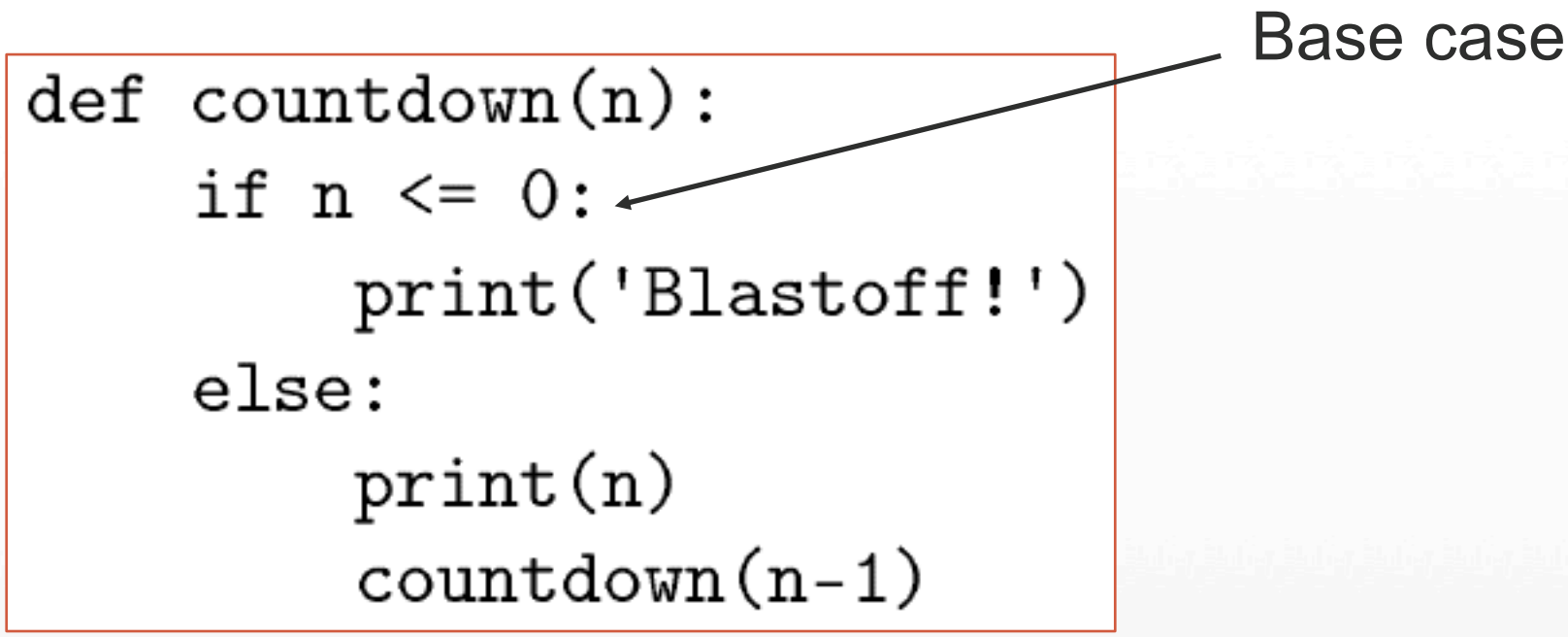
Codes

```
3
2
1
Blastoff!
```

Results of the call *countdown(3)*

Recursion

- **Base case**: a conditional branch in a recursive function that does not make a recursive call.
- The function's recursive execution should converge to the base case.



Recursion

- Factorial function: *n!*

$$n! = n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdots 3 \cdot 2 \cdot 1$$

- We can also define it as

$$n! = \begin{cases} 1, & \text{if } n = 0 \\ n \cdot (n-1)!, & \text{otherwise.} \end{cases}$$

- This definition is recursive.

Recursion

- Factorial function

Codes

```
def factorial(n):
    """
    Computes n!
    Returns the factorial of n.
    """
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
def main():
    """ Try out the factorial function """
    print(" 0! = ",  factorial(0))
    print(" 1! = ",  factorial(1))
    print(" 6! = ",  factorial(6))
    print("10! = ",  factorial(10))


main()
```

Recursion

- Factorial function

Output

```
 0! =  1
 1! =  1
 6! =  720
10! =  3628800
```

Recursion

- The call *factorial(6)* is computed as follows

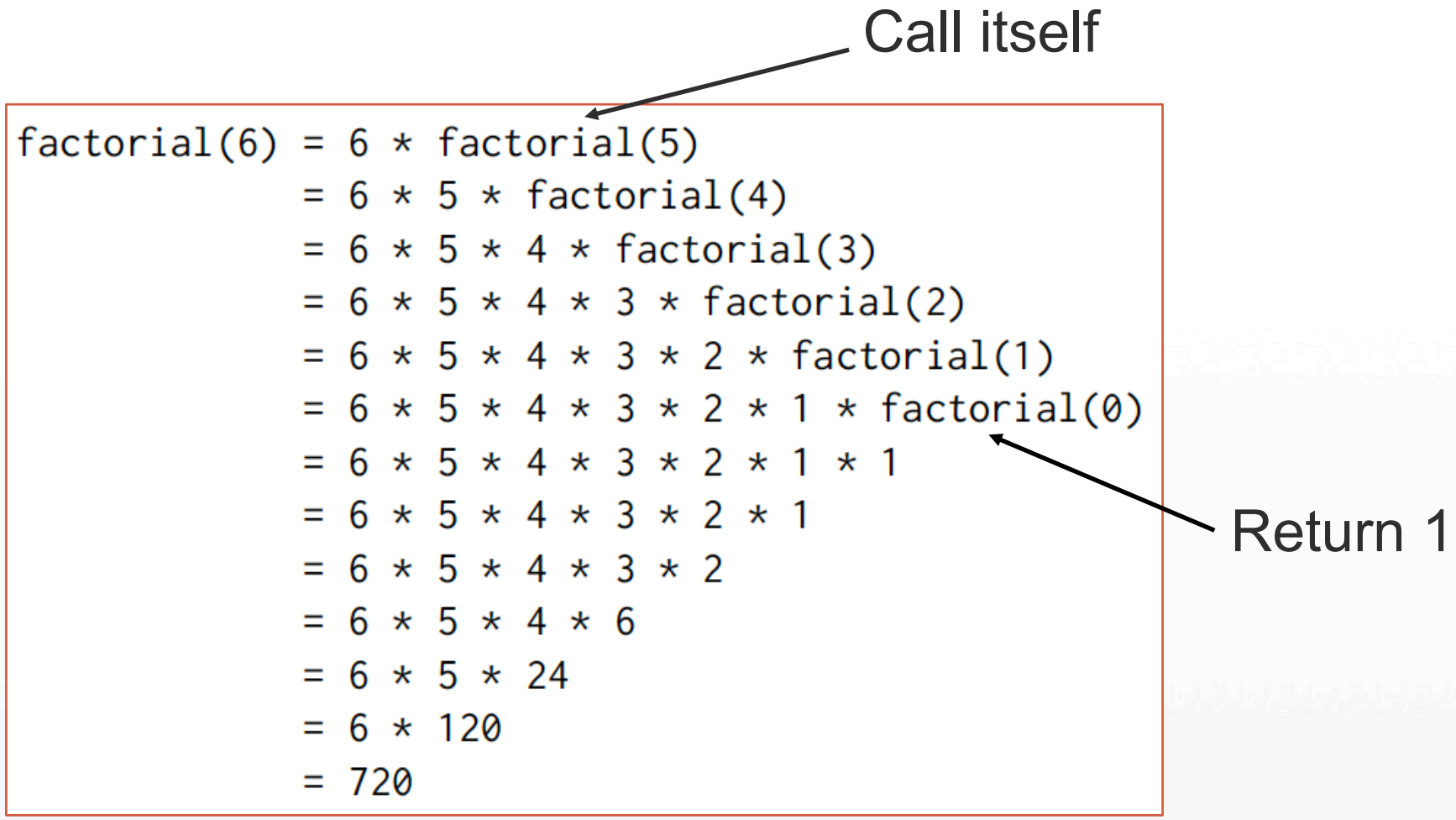
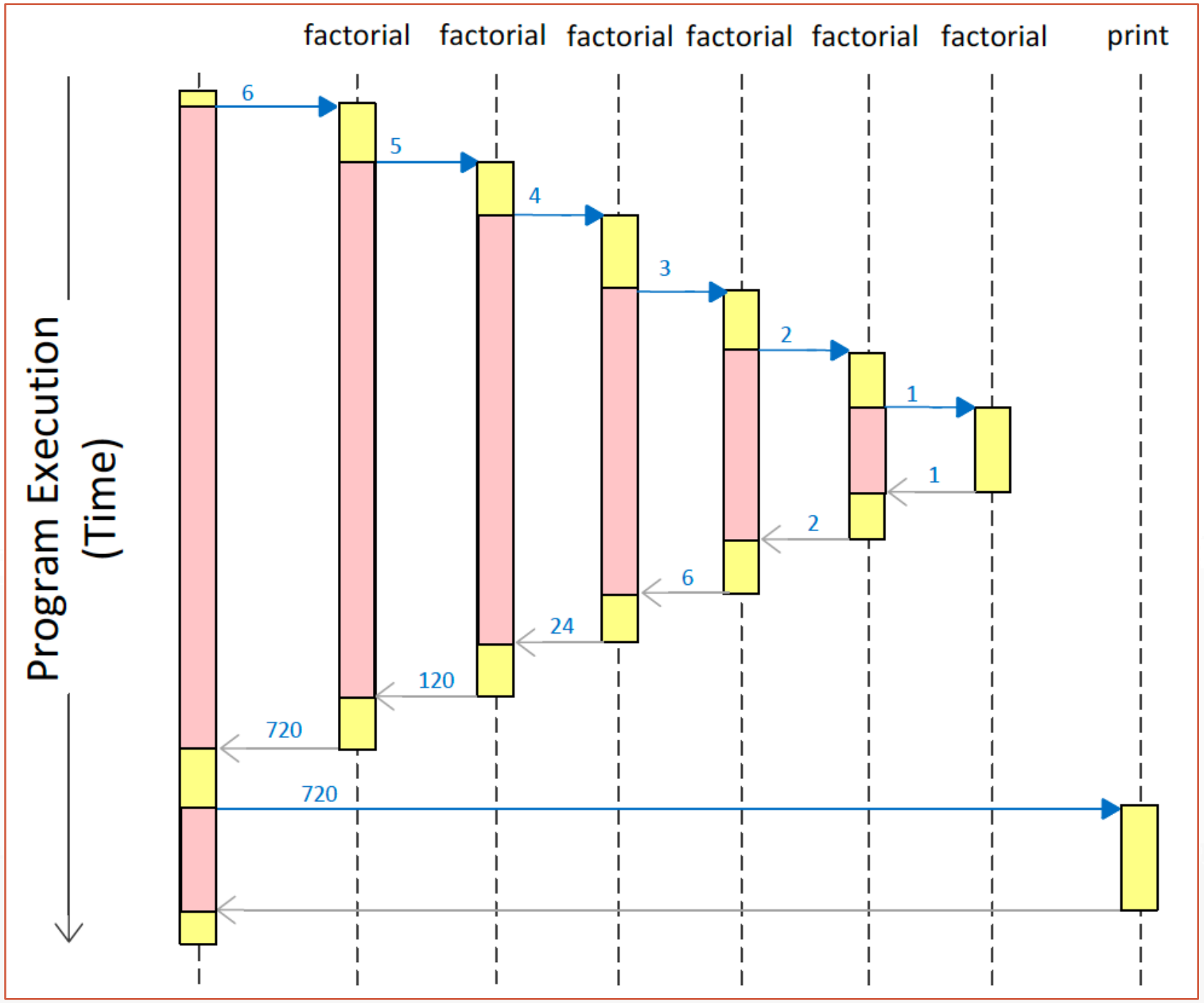


Diagram for the call *factorial(6)*



Recursion

- What if the call *factorial(1.5)?*

Recursion

- What if the call *factorial(1.5)?*
  - It will lead to an infinite recursion because it never reaches the base case.
  - One solution: check the type of argument

```
def factorial(n):
    if not isinstance(n, int):
        print('Factorial is only defined for integers.')
        return None
    elif n < 0:
        print('Factorial is not defined for negative integers.')
        return None
    elif n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Recursion

- Non-recursive factorial function
- Recursive versus non-recursive
  - Which one is better?

```
def factorial(n):
    """
    Computes n!
    Returns the factorial of n.
    """
    product = 1
    while n:
        product *= n
        n -= 1
    return product


def main():
    """ Try out the factorial function """
    print(" 0! = ",  factorial(0))
    print(" 1! = ",  factorial(1))
    print(" 6! = ",  factorial(6))
    print("10! = ",  factorial(10))


main()
```

Recursion

- A recursive function can call itself within its definition multiple times.
- Fibonacci sequence: each number is the sum of the two preceding ones, starting from 0 and 1.

$$F_0 = 1 \qquad F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2}, \ \textit{for} \ \ n > 1$$

https://en.wikipedia.org/wiki/Fibonacci_number

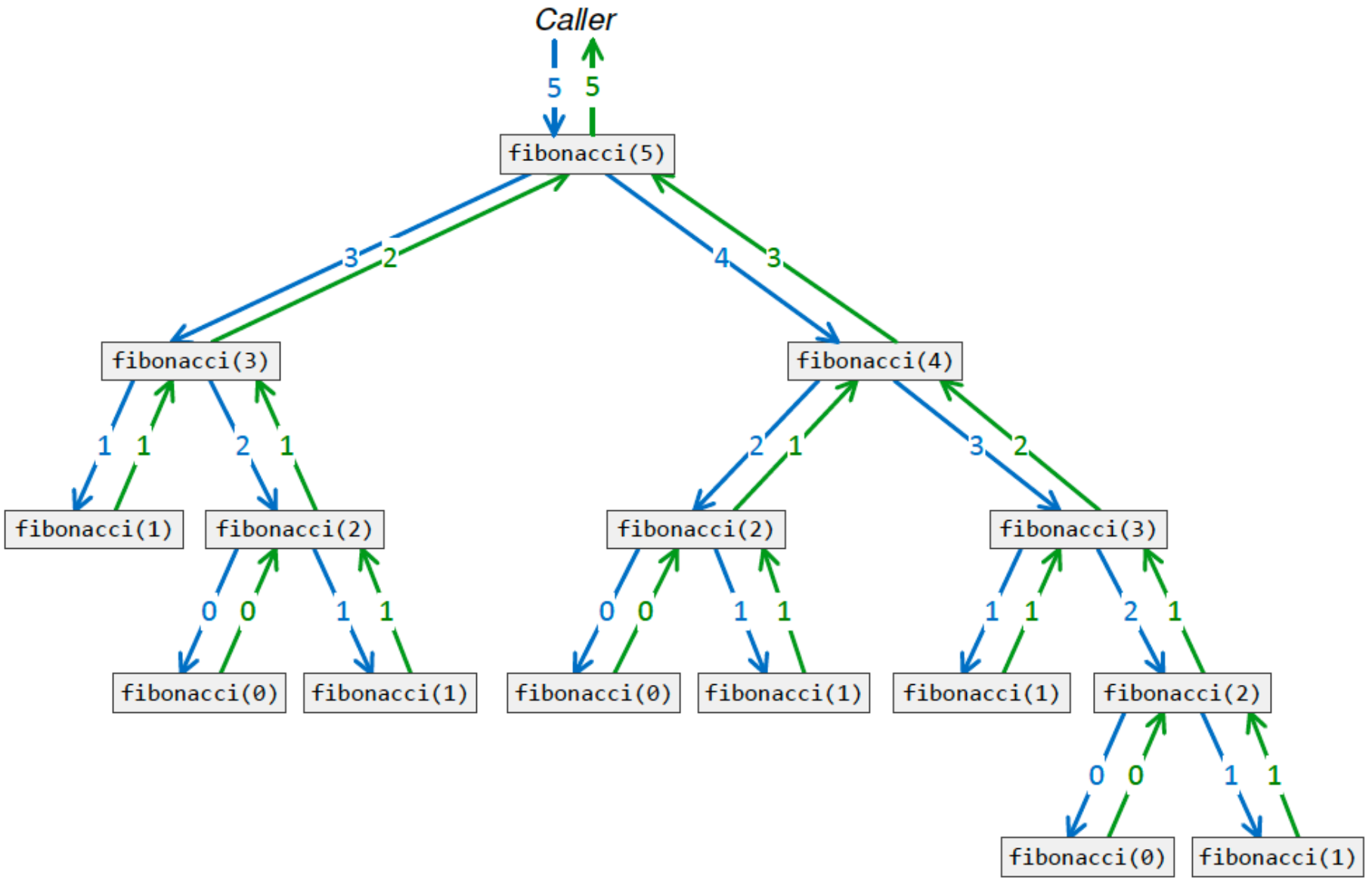
Recursion

- A recursive Python function to compute the *n*-th Fibonacci number

```
def fibonacci(n):
    """ Returns the nth Fibonacci number. """
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n - 2) + fibonacci(n - 1)
```

Recursion

Execution of the call *fibonacci(5)*



Outline

- Keyword arguments
- Recursion
- **Functions as data**
- Lambda expression
- Packaging functions into modules

Functions as data

- Passing a function as a parameter to another function

```
def evaluate(f, x, y):
    """
    Calls the function f with parameters x and y:
    f(x, y)
    """
    return f(x, y)
```

- The first parameter of function *evaluate* is a function *f*. Here the function *f* must be a function that can be called with two arguments.

Functions as data

- Passing a function as a parameter to another function

Codes

```
def add(x, y):
    """
    Adds the parameters x and y and returns the result
    """
    return x + y


def multiply(x, y):
    """
    Multiplies the parameters x and y and returns the result
    """
    return x * y
```

Codes

```
def evaluate(f, x, y):
    """
    Calls the function f with parameters x and y:
    f(x, y)
    """
    return f(x, y)


def main():
    """
    Tests the add, multiply, and evaluate functions
    """
    print(add(2, 3))
    print(multiply(2, 3))
    print(evaluate(add, 2, 3))
    print(evaluate(multiply, 2, 3))


main()  # Call main
```

Functions as data

- The results are:

Output

```
5
6
5
6
```

- The expression, *evaluate(add, 2, 3)*, passes the *add* function and the literal values 2 and 3 to *evaluate*. The *evaluate* function then invokes the *add* function with arguments 2 and 3.

Functions as data

- The results are:

Output

- The expression, *evaluate(add, 2, 3)*, passes the *add* function and the literal values 2 and 3 to *evaluate*. The *evaluate* function then invokes the *add* function with arguments 2 and 3.
- How about the statement, *print(evaluate(add, '2', '3'))*?

Outline

- Keyword arguments
- Recursion
- Functions as data
- **Lambda expression**
- Packaging functions into modules

Lambda expression

- *lambda* expression: define simple, anonymous functions

- *lambda* is a reserved word that introduces a lambda expression.
- *parameterlist* is a comma-separated list of parameters (notice the lack of parentheses).
- *expression* is a single Python expression.

Lambda expression

- *lambda* expression: define simple, anonymous functions

```
def multiply(x, y):
    """
    Multiplies the parameters x and y and returns the result
    """
    return x * y
```

```
multiply = lambda x, y: x * y
```

Lambda expression

- *lambda* expression

```
>>> def evaluate(f, x, y):
...     return f(x, y)
...
>>> evaluate(lambda x, y: 3*x + y, 10, 2)
32
>>> evaluate(lambda x, y: print(x, y), 10, 2)
10 2
>>> evaluate(lambda x, y: 10 if x == y else 2, 5, 5)
10
>>> evaluate(lambda x, y: 10 if x == y else 2, 5, 3)
2
```

Lambda expression

- Example: calculation of a derivative

$$f'(a) = \lim_{h \to 0} \frac{f(a+h) - f(a)}{h}$$

- A Python function to calculate the derivative:

```
def derivative(f, h):
    return lambda x: (f(x + h) - f(x)) / h
```

Lambda expression

- Example: calculation of a derivative for a particular function

$$f(x) = 3x^2 + 5$$

- The derivative of the function above:

$$f'(x) = 6x$$

Lambda expression

- Example: calculation of a derivative for a particular function

$$f(x) = 3x^2 + 5$$

Codes

```
def derivative(f, h):
    """  Approximates the derivative of function f
         given an h value.  The closer h is to zero,
         the better the estimate.  """
    return lambda x: (f(x + h) - f(x)) / h


def fun(x):     # The function we wish to differentiate
    return 3*x**2 + 5


def ans(x):     # The known derivative to function fun
    return 6*x
```

Codes

```
#  Difference: Approximation better as h -> 0
h = 0.0000001

#  Compute the function representing an approximation
#  of the derivative of function fun
der = derivative(fun, h)

#  Compare the computed derivative to the exact derivative
#  derived symbolically
x = 5.0
print('-------------------------------------------------------')
print('                                      Approx.    Actual')
print('   x        f(x)          h          f\'(x)      f\'(x)')
print('-------------------------------------------------------')
while x < 5.1:
    print('{:.5f}   {:.5f}   {:.8f}   {:.5f}   {:.5f}'.format(x, fun(x), h, der(x), ans(x)))
    x += 0.01
```

Lambda expression

Output

x	f(x)	h	Approx. f'(x)	Actual f'(x)
5.00000	80.00000	0.00000010	30.00000	30.00000
5.01000	80.30030	0.00000010	30.06000	30.06000
5.02000	80.60120	0.00000010	30.12000	30.12000
5.03000	80.90270	0.00000010	30.18000	30.18000
5.04000	81.20480	0.00000010	30.24000	30.24000
5.05000	81.50750	0.00000010	30.30000	30.30000
5.06000	81.81080	0.00000010	30.36000	30.36000
5.07000	82.11470	0.00000010	30.42000	30.42000
5.08000	82.41920	0.00000010	30.48000	30.48000
5.09000	82.72430	0.00000010	30.54000	30.54000
5.10000	83.03000	0.00000010	30.60000	30.60000

Outline

- Keyword arguments
- Recursion
- Functions as data
- Lambda expression
- **Packaging functions into modules**

Packaging functions into modules

- What if we want to reuse our function definitions in other programs? Do we need to copy the source code to every program?

Packaging functions into modules

- What if we want to reuse our function definitions in other programs? Do we need to copy the source code to every program?
  - The copy can be incomplete or incorrect.
  - Code duplication is a waste of space.
  - What if a bug is discovered in the function that all the other programs are built around?
- We can package our functions into modules. Building modules is independent from the programs that use them.

Packaging functions into modules

- Suppose we package the function determining prime numbers, ***is_prime***, into a file, **primecode.py,** as follows.

```
"""  Contains the definition of the is_prime function """
from math import sqrt

def is_prime(n):
    """ Returns True if nonnegative integer n is prime; otherwise, returns false """
    trial_factor = 2
    root = sqrt(n)

    while trial_factor <= root:
        if n % trial_factor == 0:    # Is trial factor a factor?
            return False             # Yes, return right away
        trial_factor += 1            # Consider next potential factor

    return True                      # Tried them all, must be prime
```

Packaging functions into modules

- Suppose we package the function determining prime numbers, ***is_prime***, into a file, **primecode.py**.
- We can import the ***is_prime*** function into another program that we work on.

```
from primecode import is_prime

num = int(input("Enter an integer: "))
if is_prime(num):
    print(num, "is prime")
else:
    print(num, "is NOT prime")
```

Readings

- Think Python, chapters 5.8 ~ 5.10 and 6.5 ~ 6.9

References

- *Think Python: How to Think Like a Computer Scientist*, 2nd edition, 2015, by Allen Downey
- *Python Cookbook*, 3rd edition, by David Beazley and Brian K. Jones, 2013
- *Python for Data Analysis*, 2nd edition, by Wes McKinney, 2018
- *Fundamentals of Python Programming,* by Richard L. Halterman