MY470 Computer Programming

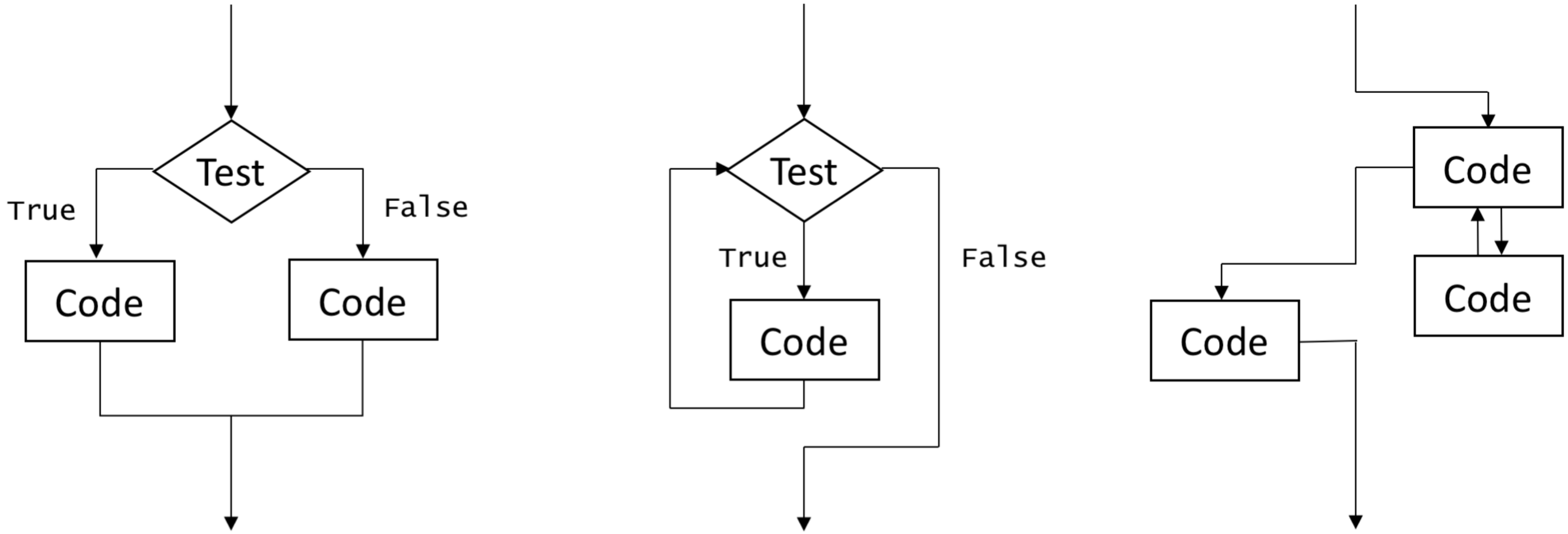
Functions in Python

Week 4 Lecture, MT 2017

Overview

- Decomposition and abstraction
- Defining and calling functions
- Variable scope
- Modules and packages
- Recursion

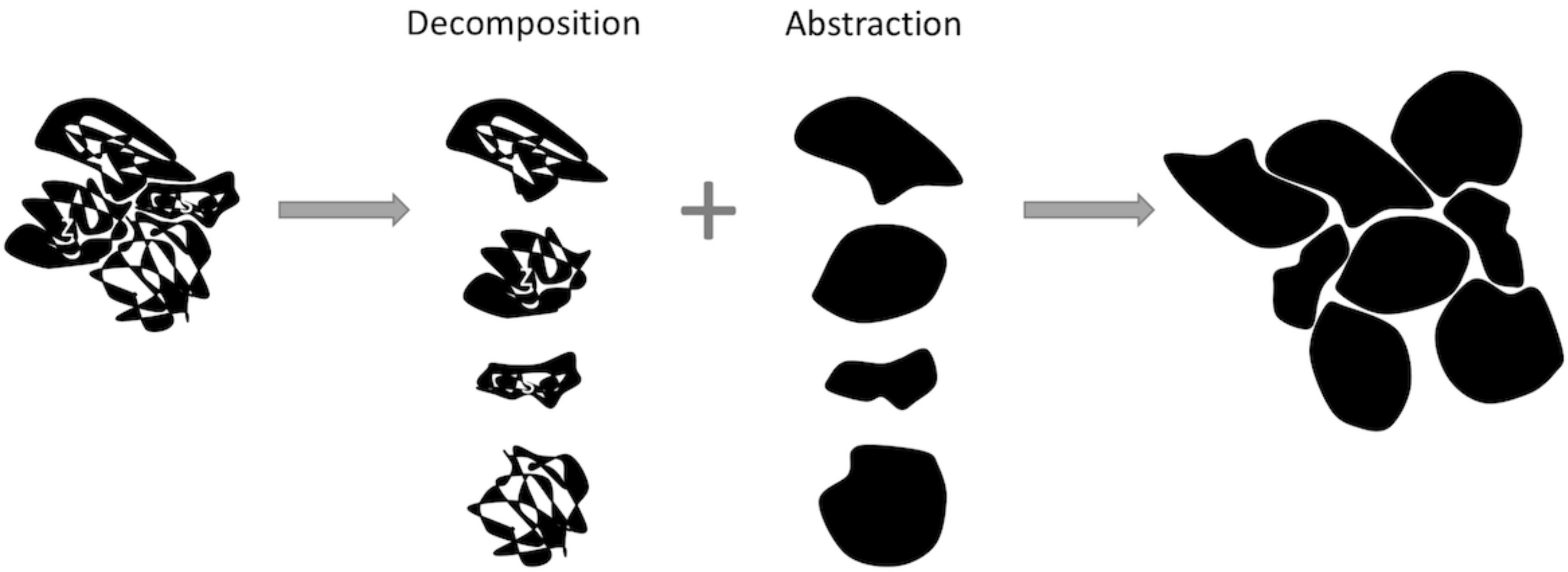
From Last Week: Control Flow



Functions

- Built-in
  - `len()`, `max()`, `range()`, `open()`, etc.
- User-defined
  - You
  - Collaborators
  - The open-source community

Decomposition and Abstraction



Defining and Calling Functions

Defining a function

```
def *function_name*(*list of parameters*):
    *body of function*
```

Calling a function

```
*function_name*(*arguments*)
```

When the Function is Used, the Parameters are Bound to the Arguments

```
def *function_name*(*list of parameters*):
    *body of function*

*function_name*(*arguments*)
```

In [10]:

```
def get_larger(x, y):
    '''Assumes x and y are of numeric type.
    Returns the larger of x and y.'''
    if x > y:
        # The execution of a `return` statement terminates the function call
        return x
    else:
        return y

m = get_larger(3, 4)
print(m)
```

```
4
```

A Function Call Always Returns a Value

- The execution of a `return` statement terminates the function call
- The function call also terminates when there are no more statements to execute
- If no expression follows `return` or there is no `return` statement, the function returns `None`

In [11]:

```
def get_larger(x, y):
    if x > y:
        return x
    if y > x:
        return y

ex1 = get_larger(3, 5)
ex2 = get_larger(6, 4)
ex3 = get_larger(3, 3)

print(ex1, ex2, ex3)
```

```
5 6 None
```

Functions with Multiple Returned Values

In [12]:

```
def double_one(a):
    return 2*a

def double_two(a, b):
    return 2*a, 2*b

x = double_two(5, 3)
print(x)

x1, x2 = double_two(5, 3)
print(x1)
print(x2)
```

```
(10, 6)
10
6
```

Positional vs. Keyword Arguments

In [13]:

```
def print_reverse(first, second, third):
    print(third, second, first)

print_reverse(1, 2, 3)
print_reverse(third = 3, second = 2, first = 1)
print_reverse(1, second = 2, third = 3)
# print_reverse(first = 1, 2, 3)  # Gives a syntax error
```

```
3 2 1
3 2 1
3 2 1
```

Keyword arguments cannot come before positional arguments!

Default Parameter Values

- Default values allow to call a function with fewer arguments than specified
- Default arguments cannot come before non-default arguments!

In [14]:

```
def pretty_print(lst, sep, fullstop = True, capitalize = True):
    toprint = sep.join(lst)
    if fullstop:
        toprint+='.'
    if capitalize:
        toprint = toprint.capitalize()
    print(toprint)

wordlst = ['the', 'quick', 'brown', 'fox', 'jumps', 'over', 'the', 'lazy', 'dog']
 # an English pangram

pretty_print(wordlst, ' ', True, True)
pretty_print(wordlst, ' ')
pretty_print(wordlst, ' ', False)
```

```
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog.
The quick brown fox jumps over the lazy dog
```

Functions with Variable Number of Arguments

In [15]:

```
def double_many(*args):
    return [2*i for i in args]

print(double_many(1))
print(double_many(1, 2, 3))
print(double_many())
```

```
[2]
[2, 4, 6]
[]
```

A Function Defines a New Scope

- Scope = name space

In [16]:

```
def func(x, y):
    x += 1
    # x is a parameter, z is a local variable
    z = x + y   # z, x, and y exist only in the scope of the definition of func
    return z

x = 1
res = func(x, 5)

print(x)  # x has not changed
#print(z)  # Returns an error
```

```
1
```

This means you can reuse your favorite variable names in different functions!

The Global Scope

In [17]:

```
globvar = 3

def print_global():
    print(globvar)  # Since globvar is not defined in the function, it is treated
 as global

print_global()
```

```
3
```

Use CAPITALS to name global variables!

Modules

- For large programs, store different parts in `.py` files
- Get access using `import` statements

In [18]:
```
import module

module.my_func('Hello!')
```

```
She said: "Hello!"
```

In [19]:
```
import module as md

md.my_func('Hello there!')
```

```
She said: "Hello there!"
```

In [20]:
```
from module import *

my_func('HELLO! DO YOU HEAR ME?')
```

```
She said: "HELLO! DO YOU HEAR ME?"
```

Useful Python Modules

https://docs.python.org/3/library/ (https://docs.python.org/3/library/)

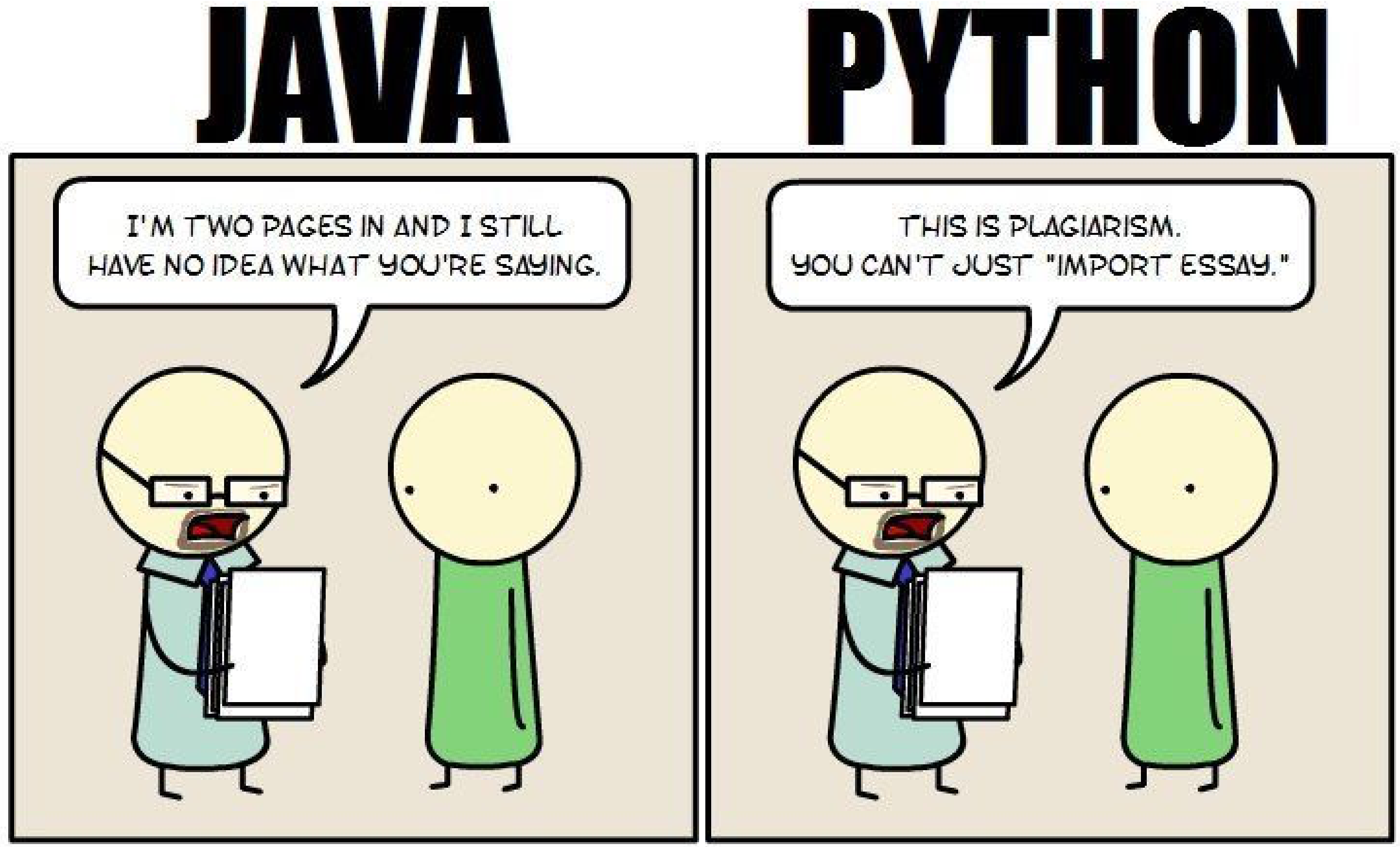
- `re` – Regular expression operations
- `datetime` – Basic date and time types
- `math` – Mathematical functions
- `random` – Generate pseudo-random numbers
- `os.path` – Common pathname manipulations
- `pickle` — Python object serialization
- `csv` — CSV file reading and writing
- `json` — JSON encoder and decoder
- ...

Useful Python Packages

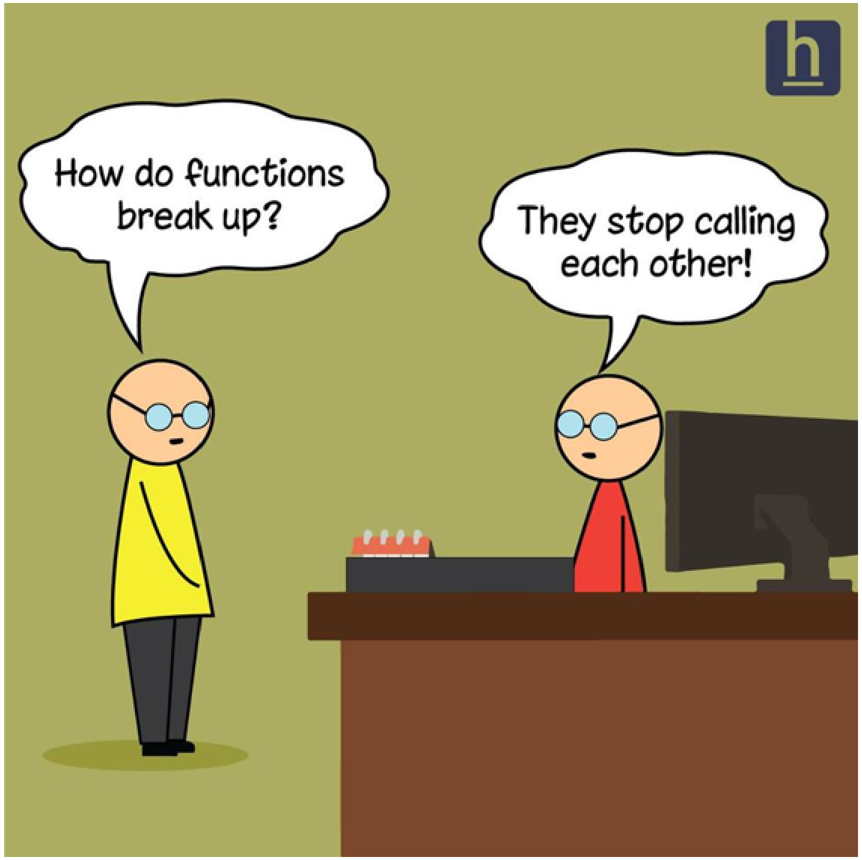
- `numpy` – Scientific computing with multi-dimensional arrays
- `pandas` – Data anlysis with table-like structures (R, pretty much)
- `statsmodels` – Statistical data analysis with linear models
- `scikit-learn` – Data mining and machine learning
- `networkx` – Network analysis
- `matplotlib` – Plotting
- ...

Using Modules and Packages in Python

How does a programmer write an essay?



Functions Calling Other Functions



Functions Calling Themselves, A.K.A. Recursion

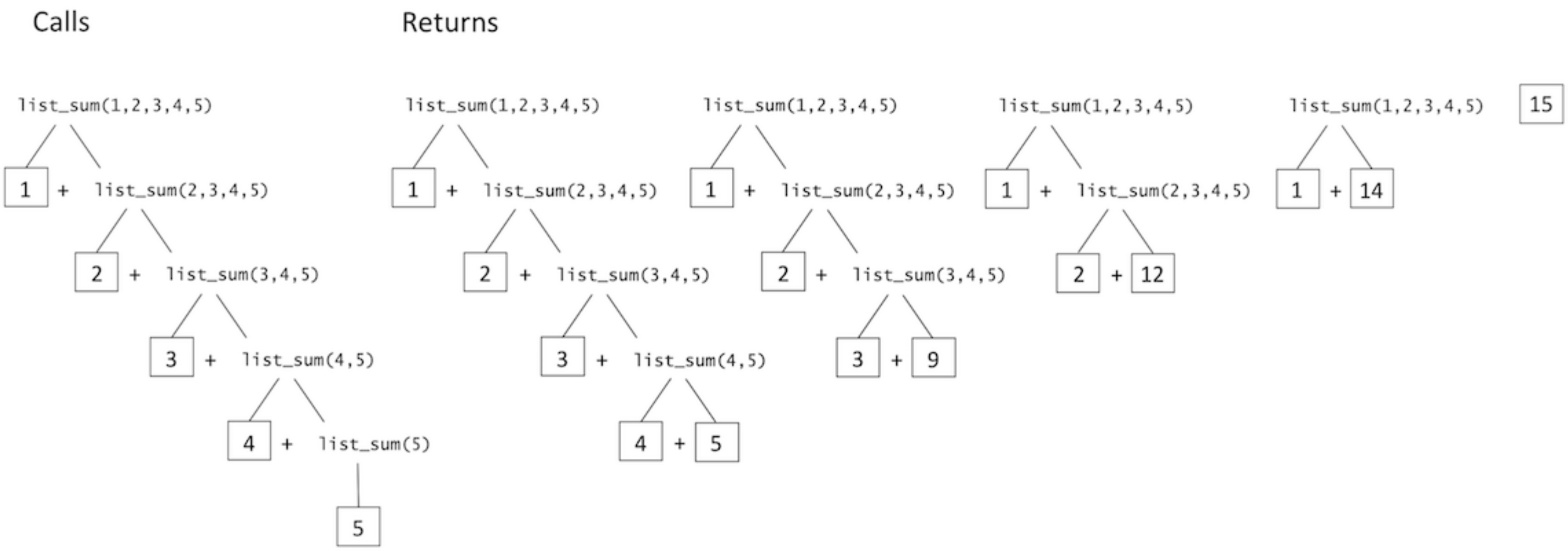
Recursion is a problem-solving method where the solution to a problem depends on solutions to smaller instances of the same problem.

In [21]:
```
# Consider the sum of elements in a list

lst = [1, 2, 3, 4, 5]
# We want (1 + 2 + 3 + 4 + 5), which is equivalent to (1 + (2 + (3 + (4 + 5)))).
# This suggests that we can reduce the problem to the problem of adding two number
s.

def list_sum(lst):
    '''Assumes lst is a sequence of numeric types.
    Returns the sum of all elements in lst.'''
    if len(lst) == 1:
        return lst[0]   # base case
    else:
        return lst[0] + list_sum(lst[1:])   # recursive case

print(list_sum(lst))
```

```
15
```

How Recursion Works

In [22]:
```
def list_sum(lst):
    if len(lst) == 1:
        return lst[0]
    else:
        return lst[0] + list_sum(lst[1:])
```



Writing Recursive Procedures

1. Define the general case
2. Define the base case
3. Ensure the base case is reached after a finite number of recursive calls

Recursion Example: The Factorial Function `n!`*

* The product of all positive integers less than or equal to n

Recursion Example: What is `n!`?

1. General case: `n! = n * (n-1)!`
2. Base case: `1! = 1`
3. Base case reached when `n==1`

In [23]:

```
def factorial(n):
    '''Assumes n is a positive integer.
    Estimates n!.'''
    if n==1:
        print('base case', n)
        answer = 1
    else:
        print('before', n)
        answer = n * factorial(n-1)
        print('after', n, answer)
    return answer

print(factorial(5))
```

```
before 5
before 4
before 3
before 2
base case 1
after 2 2
after 3 6
after 4 24
after 5 120
120
```

From Last Week: Approximation with Bisection Search

In [24]:

```
def bisec_search(x, epsilon):
    '''Assumes x and epsilon are numeric.
    Finds an approximation to the square root
    of a number x using bisection search.'''

    # Define interval for search
    low = 0
    high = max(1, x)

    # Start in the middle
    guess = (low + high) / 2

    # Narrow down search interval until guess close enough
    while abs(guess**2 - x) >= epsilon:
        if guess**2 < x:
            low = guess
        else:
            high = guess
        guess = (low + high) / 2

    return guess

print(bisec_search(25, 0.01))
```

```
5.00030517578125
```

Recursion Example: Approximation with Bisection Search

1. General case:

```
guess = (low + high) / 2
if guess**2 < x: low = guess
else: high = guess
```

- Base case: `return guess`
- The base case is reached when `abs(guess**2 - x) < epsilon`

In [25]:

```python
def bisec_search_rec(low, high, x, epsilon):
    # Start in the middle of the interval
    guess = (low + high) / 2

    if abs(guess**2 - x) < epsilon:
        return guess
    else:
        if guess**2 < x:
            return bisec_search_rec(guess, high, x, epsilon)
        else:
            return bisec_search_rec(low, guess, x, epsilon)

print(bisec_search_rec(0, 25, 25, 0.01))
```

```
5.00030517578125
```

Recursion Example: The Fibonacci Numbers*

* An integer sequence where every number after the first two is the sum of the two preceding ones: `0, 1, 1, 2, 3, 5, 8, 13, ...` (modern version)

Recursion Example: What Is the n-th Fibonacci Number?

1. General case: `fib(n) = fib(n - 1) + fib(n - 2)`
2. Base case: `fib(0) = 0`, `fib(1) = 1`
3. The base case is reached when `n==0` or `n==1`

In [26]:
```
def fib(n):
    if n==0:
        return 0
    elif n==1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)
    return answer

for i in range(10):
    print(fib(i))
```

```
0
1
1
2
3
5
8
13
21
34
```

Recursion Could be Quite Inefficient

Even if you can formulate a problem in recursive terms, it does not mean that recursion is the most efficient way to solve it.

In [27]:

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


def fib_ite(n):
    first = 0
    second = 1
    for i in range(n):
        old_first = first
        first = second
        second = old_first + second
    return first
```

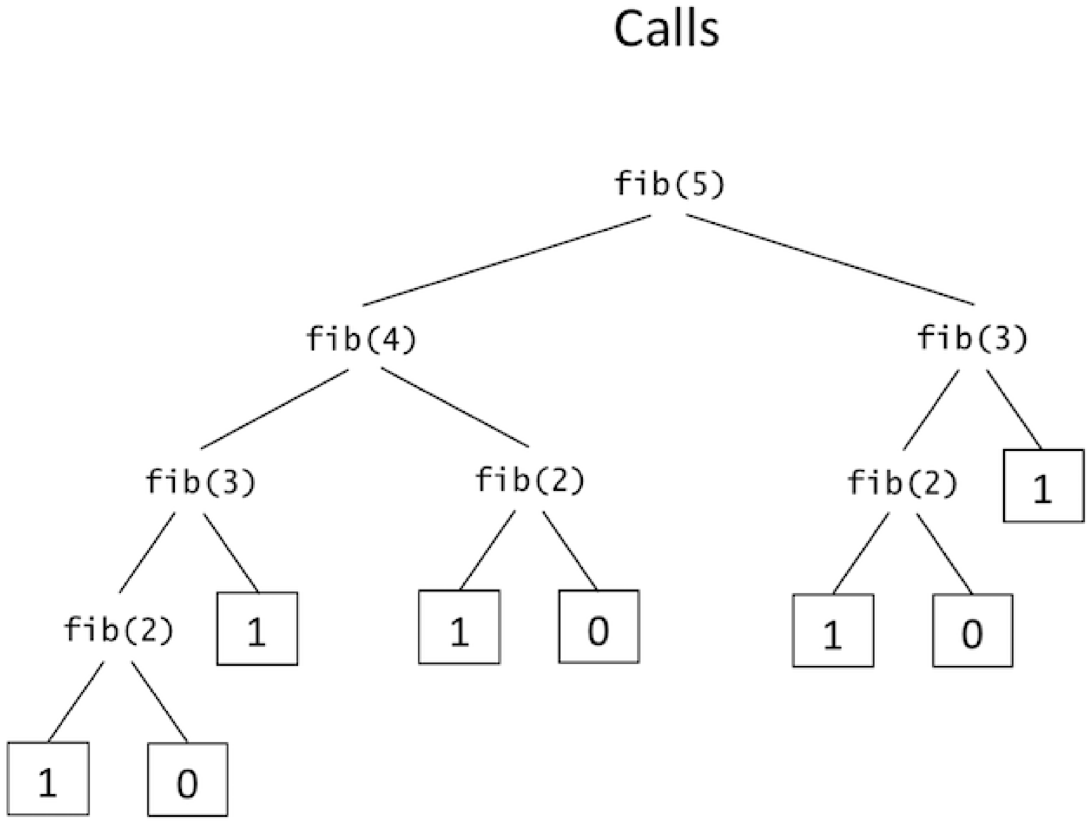
Comparing the Execution Time

In [28]:

```
import time

start_time = time.time()
fib33_rec = fib_rec(33)
print('Recursion:', (time.time() - start_time), 'seconds')

start_time = time.time()
fib33_ite = fib_ite(33)
print('Iteration:', (time.time() - start_time), 'seconds')

print(fib33_rec==fib33_ite)
```

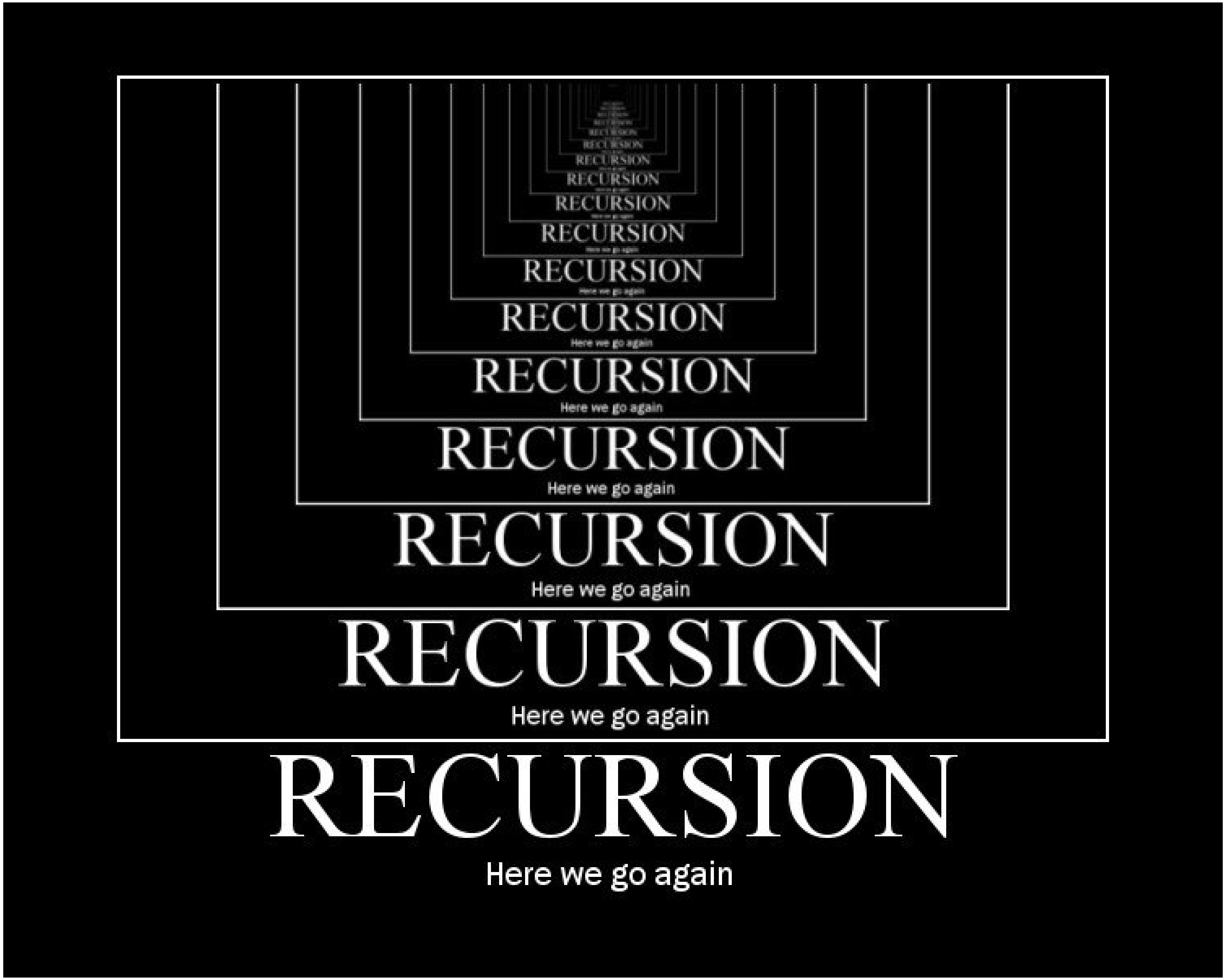
```
Recursion: 1.4864697456359863 seconds
Iteration: 5.078315734863281e-05 seconds
True
```

The Fibonacci Numbers: The Number of Recursive Calls Increases Exponentially with `n`



The calls form a binary tree because there are two recursive self-calls.

Any recursive algorithm can be transformed into an iterative one!



Functions

Functions provide **abstraction** and **decomposition**:

- Break large problems into smaller ones
- Hide the gory implementation details
- Present elementary building blocks that can be recombined to solve new problems
- Improve code legibility
- Enable collaboration

- **Lab**: Writing and calling functions in Python
- **Next week**: Classes in Python