

17.1 Recursive functions



This section has been set as optional by your instructor.

©zyBooks 09/27/22 12:27 469702

Steven Cameron

WGUC859v4

A function may call other functions, including calling itself. A function that calls itself is known as a **recursive function**. The following program illustrates.

PARTICIPATION
ACTIVITY

17.1.1: A recursive function example.



Animation captions:

1. count_down is called and count = 2.
2. count_down is recursively called and count = 1.
3. count_down is recursively called and count = 0.

The function is mostly useful for demonstrating recursion; counting down is easily done instead of using a loop. Each call to count_down creates a new namespace for the local scope of the function. The script makes the first call to count_down(), creating a namespace with the count argument bound to the integer value 2. That first function call prints 2, and calls count_down() with an argument of 1. A new namespace is created again for the local variables in count_down()'s local scope with the count argument bound to the integer value 1. That second function call prints 1, and calls count_down() with an argument of 0. That third function call prints GO!, and then because count == 0 is true, returns. The second function call is then done so it returns. The first function call is then done so it returns. Finally, the script finishes.

PARTICIPATION
ACTIVITY

17.1.2: Recursive functions.



- 1) How many times is count_down() called if the script calls count_down(5)?

©zyBooks 09/27/22 12:27 469702
Steven Cameron
WGUC859v4

Check

[Show answer](#)

- 2) How many times is count_down() called if the script



calls count_down(0)?

[Show answer](#)

- 3) Is there a difference in how we define the parameters of a recursive versus non-recursive function? Answer yes or no.

[Check](#)[Show answer](#)

©zyBooks 09/27/22 12:27 469702
Steven Cameron
WGUC859v4

CHALLENGE ACTIVITY

17.1.1: Calling a recursive function.



Write a statement that calls the recursive function backwards_alphabet() with input starting_letter.

Sample output with input: 'f'

f
e
d
c
b
a

334598.939404.qx3zqy7

```
1 def backwards_alphabet(curr_letter):  
2     if curr_letter == 'a':  
3         print(curr_letter)  
4     else:  
5         print(curr_letter)  
6         prev_letter = chr(ord(curr_letter) - 1)  
7         backwards_alphabet(prev_letter)  
8  
9 starting_letter = input()  
10  
11 ''' Your solution goes here '''  
12
```

©zyBooks 09/27/22 12:27 469702
Steven Cameron
WGUC859v4

Run

©zyBooks 09/27/22 12:27 469702

Steven Cameron
WGUC859v4

17.2 Recursive algorithm: Search



This section has been set as optional by your instructor.

An algorithm is a sequence of steps for solving a problem. For example, an algorithm for making lemonade is:

- Make lemonade
 - Add sugar to pitcher
 - Add lemon juice
 - Add water
 - Stir

Each step is distinct. Alternatively, an algorithm, for mowing the lawn is:

- Mow the lawn
 - Mow the frontyard
 - Mow the left front
 - Mow the right front
 - Mow the backyard
 - Mow the left back
 - Mow the right back

The mowing algorithm is defined *recursively*, i.e., the mowing algorithm's steps themselves consist of mowing, but of a smaller region.

Consider a guessing game program where a friend thinks of a number from 0-100 and you try to guess the number, with the friend telling you to guess higher or lower until you guess correctly. What algorithm would you use to minimize the number of guesses? An algorithm that simply guesses in increments of 1 -- Is it 0? Is it 1? Is it 2? -- requires too many guesses (50 on average). An algorithm that guesses by 10s and then by 1s -- Is it 10? Higher: Is it 20? Higher: Is it 30? Lower: Is it 21? 22? 23? -- does better but still requires about 10 guesses on average (5 to find the correct tens digit and 5 to guess the correct ones digit). An even better algorithm uses a binary search approach, guessing the midpoint of the range and halving the range after each guess -- Is it 50 (the

middle of 0-100)? Lower: Is it 25 (the middle of 0-50)? Higher: Is it 38 (the middle of 26-50)? Lower: Is it 32 (the middle of 26-38). After each guess, the binary search algorithm is applied again, just on a smaller range, i.e., the algorithm is recursive. The following animation illustrates.

PARTICIPATION ACTIVITY**17.2.1: Binary search: A well-known recursive algorithm.****Animation content:**

©zyBooks 09/27/22 12:27 469702
Steven Cameron
WGUC859v4

undefined

Animation captions:

1. The midpoint of 0 and 100 is 50.
2. 32 is lower than 50, so the window is halved and the midpoint of 0 and 50 is found.
3. 32 is greater than 25, so the window is halved and the midpoint of 26 and 50 is found.
4. 32 is less than 38, so the window is halved and the midpoint of 26 and 38, which is 32, is found.

A recursive function is a natural match for the recursive binary search algorithm. We can define a function `find(low, high)` whose parameters indicate the low and high sides of the guessing range. The function guesses at the midpoint of the range. If the user says lower, the function calls `find(low, mid)`. If the user says higher, the function calls `find(mid+1, high)`^{Note_mid}. The following program illustrates.

©zyBooks 09/27/22 12:27 469702
Steven Cameron
WGUC859v4

Figure 17.2.1: A recursive function `find()` carrying out a binary search algorithm.

```
def find(low, high):
    mid = (high + low) // 2 # Midpoint of low..high
    answer = input('Is it {}? (l/h/y):'.format(mid))

    if (answer != 'l') and (answer != 'h'): # Base case
        print('Got it!')
    else:
        if answer == 'l':
            find(low, mid)
        else:
            find(mid+1, high)

print('Choose a number from 0 to 100.')
print('Answer with:')
print('  l (your num is lower)')
print('  h (your num is higher)')
print(' any other key (guess is right).')

find(0, 100)
```

©zyBooks 09/27/22 12:27 469702
Steven Cameron

WGUC859v4
Choose a number from 0 to 100.
Answer with:

l (your num is lower)
h (your num is higher)
any other key (guess is right).
Is it 50? (l/h/y): l
Is it 25? (l/h/y): h
Is it 38? (l/h/y): h
Is it 44? (l/h/y): l
Is it 41? (l/h/y): y
Got it!

The recursive function has an if-else statement, where the if branch is the end of the recursion, known as the **base case**. The else part has the recursive calls. Such an if-else pattern is quite common in recursive functions.

Consider the following program, in which a recursive algorithm is used to find an item in a sorted list. This example is for demonstration purposes only, a programmer would be better off using the `list.index()` or "in" operator to find a specific list element. Consider having a list of attendees at a conference, whose names have been stored in alphabetical order in a list. The following program determines whether a particular person is in attendance.

©zyBooks 09/27/22 12:27 469702
Steven Cameron
WGUC859v4

Figure 17.2.2: Recursively searching a sorted list.

```
def find(lst, item, low, high):
    """
        Finds index of string in list of strings,
    else -1.
        Searches only the index range low to high
        Note: Upper/Lower case characters matter
    """
    range_size = (high - low) + 1
    mid = (high + low) // 2

    if item == lst[mid]: # Base case 1: Found at
        pos = mid
    elif range_size == 1: # Base case 2: Not
        found
        pos = -1
    else: # Recursive search: Search lower or
        upper half
        if item < lst[mid]: # Search lower half
            pos = find(lst, item, low, mid)
        else: # Search upper half
            pos = find(lst, item, mid+1, high)

    return pos

attendees = []

attendees.append('Adams, Mary')
attendees.append('Carver, Michael')
attendees.append('Domer, Hugo')
attendees.append('Fredericks, Carlo')
attendees.append('Li, Jie')

name = input("Enter person's name: Last, First:
")
pos = find(attendees, name, 0, len(attendees)-1)

if pos >= 0:
    print('Found at position {}.'.format(pos))
else:
    print('Not found.')
```

©zyBooks 09/27/22 12:27 469702
Steven Cameron
WGUC859v4

```
Enter person's name: Last, First:
Simpson, Homer
Not found.
...
Enter person's name: Last, First:
Domer, Hugo
Found at position 2.
```

The `find()` function restricts its search to elements within the range "low" to "high". The script passes a range encompassing the entire list, namely 0 to (`list length - 1`). `find()` compares to the middle element, returning that element's position if matching. If not matching, then `find()` checks if the window's size is just one element, returning -1 in that case to indicate the item was not found because there is nothing left to search in the window. If neither of those two base cases are satisfied, then `find()` uses recursive binary search, recursively searching either the lower or upper half of the range as appropriate. Use the below tool to step through execution of the above program.

In general, any recursive solution can also be done using loops. However, in some cases using a

recursive algorithm may make a solution more clear, concise, and understandable. Candidates for recursion are problems that can be reduced into smaller and identical problems, and then solved. Above, the binary search algorithms iteratively reduced the problem by half, eventually reached a base case where the problem could be solved (i.e., the desired element was located).

PARTICIPATION ACTIVITY**17.2.2: Recursive search algorithm.**

©zyBooks 09/27/22 12:27 469702

Steven Cameron
WGUC859v4

- 1) If a sorted list has elements numbers 0 to 50 and the item being searched for happens to be at location 6, how many times will the find() function be called?

Check**Show answer**

- 2) If an alphabetically sorted list (ascending) has elements numbered 0 to 50, and the item at element 0 is "Bananas", how many recursive calls to find() will be made during the failed search for "Apples"?

Check**Show answer**

- 3) A list of 5 elements is: A B D E F. A is element 0 and F is element 4. find(0,4) is called to search for item C. Write the last call to find() that would occur when searching for item C.

Check**Show answer**©zyBooks 09/27/22 12:27 469702
Steven Cameron
WGUC859v4

(*Note_mid) Because mid has already been checked, it need not be part of the new window, so mid+1 rather than mid can be used for the window's new low side, or mid-1 for the window's new high side. But the mid-1 can have the drawback of a non-intuitive base case (i.e., mid < low, because if the current window is say 4..5, mid is 4, so the new window would be 4..4-1, or 4..3). We believe range==1 is more intuitive, and thus use mid rather than mid-1. However, we still have to use mid+1 when searching higher, due to integer rounding. In particular, for window 99..100, mid is 99 ((99+100)//2=99.5, rounded to 99 due to truncation of the fraction). So the next window would again be 99..100, and the algorithm would repeat with this window forever. mid+1 prevents the problem, and doesn't miss any numbers because mid was checked and thus need not be part of the window.

17.3 Adding output statements for debugging



This section has been set as optional by your instructor.

Recursive functions can be particularly challenging to debug. Adding output statements can be helpful. Furthermore, an additional trick is to indent the print statements to show the current depth of recursion. The following program adds a parameter indent to a find() function that searches a sorted list for an item. All of the find() function's print statements start with "print indent, ...". The indent variable is typically some number of spaces. The script sets indent to three spaces " ". Each recursive call adds three more spaces. Note how the output now clearly shows the recursion depth.

Figure 17.3.1: Output statements can help debug recursive functions, especially if indented based on recursion depth.

```
def find(lst, item, low, high, indent):
    """
    Finds index of string in list of strings, else
    -1.
    Searches only the index range low to high
    Note: Upper/Lower case characters matter
    """
    print(indent, 'find() range', low, high)
    range_size = (high - low) + 1
    mid = (high + low) // 2

    if item == lst[mid]: # Base case 1: Found at
        pos = mid
        print(indent, 'Found person.')
    elif range_size == 1: # Base case 2: Not found
        print(indent, 'Person not found.')
        pos = -1
    else: # Recursive search: Search lower or upper
        half
        if item < lst[mid]: # Search lower half
            print(indent, 'Searching lower half.')
            pos = find(lst, item, low, mid, indent +
            ' ')
        else: # Search upper half
            print(indent, 'Searching upper half.')
            pos = find(lst, item, mid+1, high,
            indent + ' ')

    print(indent, 'Returning pos = {}'.format(pos))
    return pos

attendees = []

attendees.append('Adams, Mary')
attendees.append('Carver, Michael')
attendees.append('Domer, Hugo')
attendees.append('Fredericks, Carlo')
attendees.append('Li, Jie')

name = input("Enter person's name: Last, First: ")
pos = find(attendees, name, 0, len(attendees)-1,
    ' ')

if pos >= 0:
    print('Found at position {}'.format(pos))
else:
    print('Not found.'
```

©zyBooks 09/27/22 12:27 469702
Steven Cameron
WGUC859v4

Enter person's name: Last, First:
Meeks, Stan
 find() range 0 4
 Searching upper half.
 find() range 3 4
 Searching upper half.
 find() range 4 4
 Person not found.
 Returning pos = -1.
 Returning pos = -1.
 Returning pos = -1.
Not found.
...
Enter person's name: Last, First:
Adams, Mary
 find() range 0 4
 Searching lower half.
 find() range 0 2
 Searching lower half.
 find() range 0 1
 Found person.
 Returning pos = 0.
 Returning pos = 0.
 Returning pos = 0.
Found at position 0.

©zyBooks 09/27/22 12:27 469702
Steven Cameron
WGUC859v4

Some programmers like to leave the output statements in the code, commenting them out with "#" when not in use. The statements actually serve as a form of comment. More advanced techniques for handling debug output exist too, such as the *logging* Python standard library (beyond this

section's scope).

zyDE 17.3.1: Output statements in a recursive function.

Run the recursive find program having the output statements for debugging, searching "Aaron, Joe", and observe the correct output indicating the person is not found. Next, an error in the algorithm by changing "pos = -1" to "pos = 0" in the base case where the person is found. Run the program again and notice how the indented print statements help isolate the error; in particular, note how the "Person not found" output is followed by "pos = 0", which may lead one to realize the wrong value is being returned. Try instead introducing different errors and seeing how the indented print statements might help.

[Load default template...](#)

```
1  def find(lst, item, low, high, indent):
2      """
3          Finds index of string in list of
4          Searches only the index range low
5          Note: Upper/Lower case characters
6          """
7
8      print(indent, 'find() range', low)
9      range_size = (high - low) + 1
10     mid = (high + low) // 2
11     if item == lst[mid]: # Base case
12         print(indent, 'Found person.')
13         pos = mid
14     elif range_size == 1: # Base case
15         print(indent, 'Person not found')
16         pos = -1
17     else: # Recursive search: Search
```

Pre-enter any input for program, then click Run.

Run

PARTICIPATION ACTIVITY

17.3.1: Recursive function debug statements.



- 1) The above debug approach requires an extra parameter be passed to indicate the amount of indentation.



- True
- False

- 2) Each recursive call should add a few spaces to the indent parameter.



- True
- False



- 3) The function should remove a few spaces from the indent parameter before returning.

- True
- False

©zyBooks 09/27/22 12:27 469702
Steven Cameron
WGUC859v4

17.4 Creating a recursive function



This section has been set as optional by your instructor.

Creating a recursive function can be accomplished in two steps.

- *Write base case* -- Every recursive function must have a case that returns a value without performing a recursive call. That case is called the **base case**. A programmer may write that part of the function first, and then test. There may be multiple base cases.
- *Write recursive case* -- The programmer then adds the recursive case to the function.

The following illustrates for a simple function that computes the factorial of N (N!). The base case is n=1 or 1!, which evaluates to 1. The recursive case is n*nfact(n-1), which is written and tested.

Note: Factorial is not necessarily a good candidate for a recursive function, because a non-recursive version using a loop is so simple; however, factorial makes a simple example for demonstrating recursion. Actually useful cases for recursion are rarer in Python than for other programming languages, since Python programmers tend to prefer more natural iterative loop structures. Typically, recursion is useful when dealing with data structures of unknown size and connectivity, properties most commonly associated with tree-shaped data structures.

PARTICIPATION
ACTIVITY

17.4.1: Writing a recursive function for factorial: First writing the base case, then adding the recursive case.



Animation captions:

©zyBooks 09/27/22 12:27 469702
Steven Cameron
WGUC859v4

1. The base case (non-recursive case) has to be written and tested.
2. The recursive case has to be added and tested.

Before writing a recursive function, a programmer should determine: (1) Whether the problem has a naturally recursive solution, *and* (2) whether that solution is better than a non-recursive solution. For example, computing $E = M*C*C$ doesn't seem to have a natural recursive solution. Computing $n!$ (n factorial) does have a natural recursive solution, but a recursive solution is not better than a

non-recursive solution that simply uses a loop, as in

`for i in range(n, 0, -1): result *= ifactorial` Binary search has a natural recursive solution, and that solution may be easier to understand than a non-recursive solution.

A common error is to not cover all possible base cases in a recursive function. Another common error is to write a recursive function that doesn't always reach a base case. Both errors may lead to infinite recursion, causing the program to fail.

Commonly, programmers will use two functions for recursion. An "outer" function is intended to be called from other parts of the program, like the function "factorial(n)". An "inner" function is intended only to be called from that outer function, like the function "_factorial(n)" (note the "_"). The outer function may check for a valid input value, e.g., ensuring n is not negative, and then calling the inner function. Commonly, the inner function has parameters that are mainly of use as part of the recursion, and need not be part of the outer function, thus keeping the outer function more intuitive.

PARTICIPATION ACTIVITY

17.4.2: Creating a recursive function.



- 1) A recursive function with parameter n counts up from any negative number to 0. An appropriate base case would be n==0.

- True
 False

- 2) A recursive function can have two base cases, such as n==0 returning 0, and n==1 returning 1.

- True
 False

- 3) n factorial (n!) is commonly implemented as a recursive function due to being easier to understand and executing faster than a loop implementation.

- True
 False

©zyBooks 09/27/22 12:27 469702
Steven Cameron
WGUC859v4

CHALLENGE ACTIVITY

17.4.1: Recursive function: Writing the base case.



Add an if branch to complete double_pennies()'s base case.

Sample output with inputs: 1 10

Number of pennies after 10 days: 1024

©zyBooks 09/27/22 12:27 469702
Steven Cameron
WGUC859v4

Note: If the submitted code has an infinite loop, the system will stop running the code after a few seconds, and report "Program end never reached." The system doesn't print the test case that caused the reported message.

334598.939404.qx3zqy7

```
1 # Returns number of pennies if pennies are doubled num_days times
2 def double_pennies(num_pennies, num_days):
3     total_pennies = 0
4
5     ''' Your solution goes here '''
6
7     else:
8         total_pennies = double_pennies((num_pennies * 2), (num_days - 1))
9
10    return total_pennies
11
12 # Program computes pennies if you have 1 penny today,
13 # 2 pennies after one day, 4 after two days, and so on
14 starting_pennies = int(input())
15 user_days = int(input())
16
17 print('Number of pennies after', user_days, 'days: ', end="")
```

Run

CHALLENGE
ACTIVITY

17.4.2: Recursive function: Writing the recursive case.



Write code to complete factorial_str()'s recursive case.

©zyBooks 09/27/22 12:27 469702
Steven Cameron
WGUC859v4

Sample output with input: 5

5! = 5 * 4 * 3 * 2 * 1 = 120

334598.939404.qx3zqy7

```
1 def factorial_str(fact counter, fact value):
```

©zyBooks 09/27/22 12:27 469702
Steven Cameron
WGUC859v4

Run

(*factorial) In this discussion, we ignore the fact that the math module has a very convenient math.factorial(n) function.

17.5 Recursive math functions



This section has been set as optional by your instructor.

Recursive functions can be used to solve certain math problems, such as computing the Fibonacci sequence. The Fibonacci sequence is 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, etc. The pattern is to compute the next number by adding the previous two numbers. The sequence starts with 0 and 1.

Below is a program that outputs the Fibonacci sequence step-by-step for a user-entered number of steps. The program starts after the first 0 and 1 of the Fibonacci sequence. The base case is that the program has output the requested number of steps. The recursive case computes the next step.

©zyBooks 09/27/22 12:27 469702
Steven Cameron
WGUC859v4

Figure 17.5.1: Fibonacci sequence step-by-step.

```
"""
Output the Fibonacci sequence step-by-step.
Fibonacci sequence starts as:
0 1 1 2 3 5 8 13 21 ... in which the first
two numbers are 0 and 1 and each additional
number is the sum of the previous two numbers
"""

def fibonacci(v1, v2, run_cnt):
    print(v1, '+', v2, '=', v1+v2)

    if run_cnt <= 1: # Base case:
        # Ran for user's number of
        steps
        pass # Do nothing
    else:           # Recursive case
        fibonacci(v2, v1+v2, run_cnt-1)

print ('This program outputs the\n'
       'Fibonacci sequence step-by-step,\n'
       'starting after the first 0 and 1.\n')

run_for = int(input('How many steps would you
like?'))

fibonacci(0, 1, run_for)
```

©zyBooks 09/27/22 12:27 469702
Steven Cameron
WGUC859v4

This program outputs the
Fibonacci sequence step-by-
step,
starting after the first 0 and
1.

How many steps would you
like?10

0 + 1 = 1
1 + 1 = 2
1 + 2 = 3
2 + 3 = 5
3 + 5 = 8
5 + 8 = 13
8 + 13 = 21
13 + 21 = 34
21 + 34 = 55
34 + 55 = 89

zyDE 17.5.1: Recursive Fibonacci.

Write a program that outputs the nth Fibonacci number, where n is a user-entered number. If the user enters 4, the program should output 3 (without outputting the intermediate values). Use a recursive function compute_nth_fib that takes n as a parameter and returns the nth Fibonacci number. The function has two base cases: input 0 returns 0, and input 1 returns 1.

©zyBooks 09/27/22 12:27 469702

Steven Cameron

WGUC859v4

Load default template...

Pre-enter any input for program, then click Run.

Run

```
1 def compute_nth_fib(num):
2     # if base case ...
3         # return base case value ...
4     # else ...
5         # recursively call compute_nth_fib(...)
```

Recursion can be used to solve the greatest common divisor (GCD) problem. The GCD is the largest number that divides evenly into two numbers, e.g. $\text{GCD}(12, 8) = 4$. A simple algorithm to compute the GCD subtracts the smaller number from the larger number until both numbers are equal. For example, $\text{GCD}(12, 8) = \text{GCD}(12-8=4, 8) = \text{GCD}(4, 8-4=4)$. The equal numbers are the GCD. Euclid described this algorithm around 300 BC.

The below program recursively computes the GCD of two numbers. The base case is that the two numbers are equal, so that number is returned. The recursive case subtracts the smaller number from the larger number and then calls GCD with the new pair of numbers.

©zyBooks 09/27/22 12:27 469702

Steven Cameron

WGUC859v4

Figure 17.5.2: Calculate greatest common divisor of two numbers.

```
"""
Determine the greatest common divisor
of two numbers, e.g., GCD(8, 12) = 4
"""

def gcd(n1, n2):
    if n1 % n2 == 0:                      # n2 is a
        common factor
        return n2
    else:
        return gcd(n2, n1%n2)

print ('This program outputs the greatest
       'common divisor of two
       numbers.\n')

num1 = int(input('Enter first number: '))
num2 = int(input('Enter second number:
')))

if (num1 < 1) or (num2 < 1):
    print('Note: Neither value can be
below 1.')
else:
    my_gcd = gcd(num1, num2)
    print('Greatest common divisor =',
my_gcd)
```

©zyBooks 09/27/22 12:27 469702
Steven Cameron
WGUC859v4

This program outputs the greatest common divisor of two numbers.

Enter first number:12
Enter second number:8
Greatest common divisor = 4

...
This program outputs the greatest common divisor of two numbers.

Enter first number:456
Enter second number:784
Greatest common divisor = 8

The **depth** of recursion is a measure of how many recursive calls of a function have been made, but have not yet returned. Each recursive call requires the Python interpreter to allocate more memory, and eventually all of the system memory could be used. Thus, a recursion depth limit exists, accessible using the function `sys.getrecursionlimit()`. The default recursion depth limit is typically 1000. The limit can be changed using `sys.setrecursionlimit()`. Exceeding the depth limit causes a `RuntimeError` to occur. Ex: The following program causes 1000 recursive calls.

©zyBooks 09/27/22 12:27 469702
Steven Cameron
WGUC859v4

Figure 17.5.3: Limit on recursion depth.

```
def rec_func(n):
    if n == 0:
        return 1
    return rec_func(n - 1)

num = int(input('Enter the number: '))
print(rec_func(num))
```

Enter the number: 1000

Traceback (most recent call last):
File "main.py", line 8, in <module>
 print(rec_func(num))
File "main.py", line 4, in rec_func
 return rec_func(n - 1)
File "main.py", line 4, in rec_func
 return rec_func(n - 1)
File "main.py", line 4, in rec_func
 return rec_func(n - 1)
[Previous line repeated 995 more times]
File "main.py", line 2, in rec_func
 if n == 0:
RecursionError: maximum recursion depth exceeded in comparison

©zyBooks 09/27/22 12:27 469702 Steven Cameron WGUC859v4

PARTICIPATION ACTIVITY

17.5.1: Recursive GCD.



- 1) How many calls are made to the gcd function for gcd(12, 8)?

Check**Show answer**

- 2) How many calls are made to the gcd function for gcd(5, 3)?

Check**Show answer**

Exploring further:

- More on the Fibonacci sequence from wikipedia.org
- More on the GCD algorithm from wikipedia.org

©zyBooks 09/27/22 12:27 469702
Steven Cameron
WGUC859v4

**CHALLENGE
ACTIVITY**

17.5.1: Writing a recursive math function.



Write code to complete `raise_to_power()`. Note: This example is for practicing recursion; a non-recursive function, or using the built-in function `math.pow()`, would be more common.

Sample output with inputs: 4 2

4² = 16

©zyBooks 09/27/22 12:27 469702
Steven Cameron
WGUC859v4

334598.939404.qx3zqy7

```
1 def raise_to_power(base_val, exponent_val):
2     if exponent_val == 0:
3         result_val = 1
4     else:
5         result_val = base_val * ''' Your solution goes here '''
6
7     return result_val
8
9 user_base = int(input())
10 user_exponent = int(input())
11
12 print('{0}^{1} = {2}'.format(user_base, user_exponent,
13     raise_to_power(user_base, user_exponent)))
```

Run

17.6 Recursive exploration of all possibilities



This section has been set as optional by your instructor.

©zyBooks 09/27/22 12:27 469702
Steven Cameron
WGUC859v4

Recursion is a powerful tool for exploring all possibilities, such as all possible reorderings of a word's letters, all possible subsets of items, all possible paths between cities, etc. This section provides several examples of using recursion for such exploration.

Consider the problem of printing all possible combinations (or "scramblings") of a word's letters. For example, the letters of "abc" can be scrambled in 6 ways: abc, acb, bac, bca, cab, cba. Those

possibilities can be obtained by thinking of three choices: Choosing the first letter ("a", "b", or "c"), then choosing the second letter (if "a" was the first choice, then second possible choices are "b" or "c"; if "b" was the first choice, then second possible choices are "a" and "c"; etc.), then choosing the third letter. The choices can be depicted using a tree. Each level represents a choice. Each node in the tree shows the unchosen letters on the left, and the chosen letters on the right, as in the animation figure below.

Such a tree forms the basis for a recursive exploration function to generate all possible combinations of a string's letters. The function will take two parameters, one for the unchosen letters, and one for the already chosen letters. The base case will be when no letters exist in the unchosen letters, in which case the chosen letters are printed. The recursive case will call the function once for each letter in the unchosen letters. The following animation depicts how such a recursive algorithm would traverse the tree. The leaves of the tree (the bottommost nodes) represent the base case.

PARTICIPATION ACTIVITY**17.6.1: Exploring all possibilities viewed as a tree of choices.****Animation captions:**

1. "a" is chosen from "abc", then "b" is chosen from "bc". Finally, "c" is chosen from "c".
2. "b" has already been chosen from "bc". "c" can also be chosen. "acb" is chosen from "b".
3. "b" is chosen from "abc".
4. "c" is chosen from "abc".

The program below receives a word from the user then jumbles all of its letters in to every possible ordering. The base case is that all letters have been used. In the recursive case, a remaining letter is moved to the scrambled letters, recursively explored, then put back. This is done for each remaining letter.

Figure 17.6.1: Scramble a word's letters in every possible way.

```
def scramble(r_letters, s_letters):
    """
        Output every possible combination of a word.
        Each recursive call moves a letter from
        r_letters (remaining letters) to
        s_letters (scrambled letters)
    """
    if len(r_letters) == 0:
        # Base case: All letters used
        print(s_letters)
    else:
        # Recursive case: For each call to scramble()
        # move a letter from remaining to scrambled
        for i in range(len(r_letters)):
            # The letter at index i will be scrambled
            scramble_letter = r_letters[i]

            # Remove letter to scramble from remaining
            # letters list
            remaining_letters = r_letters[:i] +
            r_letters[i+1:]

            # Scramble letter
            scramble(remaining_letters, s_letters +
            scramble_letter)

word = input('Enter a word to be scrambled: ')
scramble(word, '')
```

©zyBooks 09/27/22 12:27 469702
Steven Cameron
WGUC859v4

Enter a word to be
scrambled: cat
cat
cta
act
atc
tca
tac

Recursion is useful for finding all possible subsets of a set of items. The following example is a shopping spree in which you may select a 3-item subset from a larger set of items. The program should print all possible 3-item subsets given the larger set. The program also happens to print the total price value of those items.

The ShoppingBagCombinations() function has a parameter for the current bag contents, and a parameter for the remaining items from which to choose. The base case is that the current bag already has 3 items. The recursive case is to move one of the remaining items to the bag, recursively call the function, then move the item back from the bag to the remaining items.

©zyBooks 09/27/22 12:27 469702
Steven Cameron
WGUC859v4

Figure 17.6.2: Shopping spree in which you can fit 3 items in your shopping bag.

```
max_items_in_bag = 3

def shopping_bag_combinations(curr_bag, remaining_items):
    """
    Output every combination of items that fit
    in a shopping bag. Each recursive call moves
    one item into the shopping bag.
    """
    if len(curr_bag) == max_items_in_bag:
        # Base case: Shopping bag full
        bag_value = 0
        for item in curr_bag:
            bag_value += item['price']
            print(item['name'], ' ', end=' ')
        print('=', bag_value)
    else:
        # Recursive case: Move one of the remaining items
        # to the shopping bag.
        for index, item in enumerate(remaining_items):
            # Move item into bag
            curr_bag.append(item)
            remaining_items.pop(index)

            shopping_bag_combinations(curr_bag,
                                        remaining_items)

            # Take item out of bag
            remaining_items.insert(index, item)
            curr_bag.pop()

    items = [
        {
            'name': 'Milk',
            'price': 1.25
        },
        {
            'name': 'Belt',
            'price': 23.55
        },
        {
            'name': 'Toys',
            'price': 19.05
        },
        {
            'name': 'Cups',
            'price': 11.85
        }
    ]
    bag = []
    shopping_bag_combinations(bag, items)
```

Milk	Belt	Toys	=
43.85	Steven Cameron		
Milk	Belt	Cups	=
36.65		WGUC859v4	
Milk	Toys	Belt	=
43.85			
Milk	Toys	Cups	=
32.15			
Milk	Cups	Belt	=
36.65			
Milk	Cups	Toys	=
32.15			
Belt	Milk	Toys	=
43.85			
Belt	Milk	Cups	=
36.65			
Belt	Toys	Milk	=
43.85			
Belt	Toys	Cups	=
54.45			
Belt	Cups	Milk	=
36.65			
Belt	Cups	Toys	=
54.45			
Toys	Milk	Belt	=
43.85			
Toys	Milk	Cups	=
32.15			
Toys	Belt	Milk	=
43.85			
Toys	Belt	Cups	=
54.45			
Toys	Cups	Milk	=
32.15			
Toys	Cups	Belt	=
54.45			
Cups	Milk	Toys	=
36.65			
Cups	Milk	Belt	=
32.15			
Cups	Belt	Milk	=
36.65			
Cups	Belt	Toys	=
54.45			
Cups	Toys	Milk	=
32.15			
Cups	Toys	Belt	=
54.45			

Recursion is useful for finding all possible paths. In the following example, a salesman must travel to 3 cities: Boston, Chicago, and Los Angeles. The salesman wants to know all possible paths

among those three cities, starting from any city. A recursive exploration of all travel paths can be used. The base case is that the salesman has traveled to all cities. The recursive case is to travel to a new city, explore possibilities, then return to the previous city.

Figure 17.6.3: Find distance of traveling to 3 cities.

©zyBooks 09/27/22 12:27 469702

Steven Cameron
WGUC859v4

```
num_cities = 3
city_names = []
distances = []

def travel_paths(curr_path, need_to_visit):
    if len(curr_path) == num_cities: # Base case:
        Visited all cities
        total_distance = 0
        for i in range(len(curr_path)):
            print(city_names[curr_path[i]], ' ', end='')

        if i > 0:
            total_distance += distances[curr_path[i-1]][curr_path[i]]

        print('=', total_distance)
    else: # Recursive case: Travel to each city
        for i in range(len(need_to_visit)):
            # Visit city
            city = need_to_visit[i]
            need_to_visit.pop(i)
            curr_path.append(city)

            travel_paths(curr_path, need_to_visit)

            need_to_visit.insert(i, city)
            curr_path.pop()

distances.append([0])
distances[0].append(960) # Boston-Chicago
distances[0].append(2960) # Boston-Los Angeles
distances.append([960]) # Chicago-Boston
distances[1].append(0)
distances[1].append(2011) # Chicago-Los Angeles
distances.append([2960]) # Los Angeles-Boston
distances[2].append(2011) # Los Angeles-Chicago
distances[2].append(0)

city_names = ["Boston", "Chicago", "Los Angeles"]

path = []
need_to_visit = [0, 1, 2] # (Need to visit all 3 cities)
travel_paths(path, need_to_visit)
```

Boston	Chicago	Los
Angeles	= 2971	
Boston	Los Angeles	
Chicago	= 4971	
Chicago	Boston	Los
Angeles	= 3920	
Chicago	Los Angeles	
Boston	= 4971	
Los Angeles	Boston	
Chicago	= 3920	
Los Angeles	Chicago	
Boston	= 2971	

©zyBooks 09/27/22 12:27 469702
Steven Cameron
WGUC859v4

PARTICIPATION ACTIVITY

17.6.2: Recursive exploration.



- 1) What is the output of:



scramble("xy", "")? Determine your answer by manually tracing the code, not by running the program.

Check**Show answer**

©zyBooks 09/27/22 12:27 469702
Steven Cameron
WGUC859v4



- 2) You wish to generate all possible 3-letter subsets from the letters in an N-letter word ($N > 3$). Which of the above recursive functions is the closest (just enter the function's name)?

Check**Show answer****CHALLENGE ACTIVITY**

17.6.1: Enter the output of recursive exploration.



334598.939404.qx3zqy7

Start

Type the program's output

©zyBooks 09/27/22 12:27 469702
Steven Cameron
WGUC859v4

```
def scramble_nums(remain_nums, scram_nums):  
    if len(remain_nums) == 0:  
        print(scram_nums[0], scram_nums[1], scram_nums[2], sep=' ')  
    else:  
        for i in range(len(remain_nums)):  
            tmp_remain_nums = remain_nums[:] # Make a copy.  
            tmp_removed_num = tmp_remain_nums[i]  
            tmp_remain_nums.pop(i) # Remove element at i.
```

425
452

Exploring further:

- More on recursion trees from Wikipedia.org.

©zyBooks 09/27/22 12:27 469702
Steven Cameron
WGUC859v4

17.7 LAB: Fibonacci sequence (recursion)



This section has been set as optional by your instructor.

The Fibonacci sequence begins with 0 and then 1 follows. All subsequent values are the sum of the previous two, for example: 0, 1, 1, 2, 3, 5, 8, 13. Complete the fibonacci() function, which takes in an index, n, and returns the nth value in the sequence. Any negative index values should return -1.

Ex: If the input is:

7

the output is:

fibonacci(7) is 13

Note: Use recursion and DO NOT use any loops.

334598.939404.qx3zqy7

LAB
ACTIVITY

17.7.1: LAB: Fibonacci sequence (recursion)

©zyBooks 09/27/22 12:01 469702
Steven Cameron
WGUC859v4

main.py

1 Loading latest submission... |

©zyBooks 09/27/22 12:27 469702
Steven Cameron
WGUC859v4

Develop mode

Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above)



main.py
(Your program)



0

Program output displayed here

Coding trail of your work [What is this?](#)

Retrieving signature

17.8 LAB: All permutations of names

©zyBooks 09/27/22 12:27 469702
Steven Cameron
WGUC859v4



This section has been set as optional by your instructor.

Write a program that lists all ways people can line up for a photo (all permutations of a list of strings). The program will read a list of one word names, then use a recursive method to create and output all possible orderings of those names, one ordering per line.

When the input is:

Julia Lucas Mia

then the output is (must match the below ordering):

Julia Lucas Mia
Julia Mia Lucas
Lucas Julia Mia
Lucas Mia Julia
Mia Julia Lucas
Mia Lucas Julia

©zyBooks 09/27/22 12:27 469702
Steven Cameron
WGUC859v4

334598.939404.qx3zqy7

LAB
ACTIVITY

17.8.1: LAB: All permutations of names

0 / 10

main.py

[Load default template...](#)

```
1 def all_permutations(permList, nameList):  
2     # TODO: Implement method to create and output all permutations of the list of n  
3  
4     if __name__ == "__main__":  
5         nameList = input().split(' ')  
6         permList = []  
7         all_permutations(permList, nameList)
```

[Develop mode](#)

[Submit mode](#)

Run your program as often as you'd like, before
submitting for grading. Below, type any needed input
values in the first box, then click **Run program** and
observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above)

**main.py**
(Your program)

0

Program output displayed here

©zyBooks 09/27/22 12:27 469702

Steven Cameron
WGUC859v4Coding trail of your work [What is this?](#)

History of your effort will appear here once you begin working on this zyLab.

17.9 LAB: Number pattern



This section has been set as optional by your instructor.

Write a recursive function called `print_num_pattern()` to output the following number pattern.

Given a positive integer as input (Ex: 12), subtract another positive integer (Ex: 3) continually until 0 or a negative value is reached, and then continually add the second integer until the first integer is again reached. For this lab, do not end output with a newline.

Ex. If the input is:

12
3

the output is:

12 9 6 3 0 3 6 9 12

©zyBooks 09/27/22 12:27 469702

334598.939404.qx3zqy7

Steven Cameron
WGUC859v4**LAB ACTIVITY**

17.9.1: LAB: Number pattern

0 / 10

**main.py**[Load default template...](#)

```
1 # TODO: Write recursive print_num_pattern() function
```

©zyBooks 09/27/22 12:27 469702
Steven Cameron
WGUC859v4

Develop mode

Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above)



main.py
(Your program)



0

Program output displayed here

Coding trail of your work [What is this?](#)

History of your effort will appear here once you begin working on this zyLab.

©zyBooks 09/27/22 12:27 469702
Steven Cameron
WGUC859v4

17.10 LAB: Count the digits



This section has been set as optional by your instructor.

Write a recursive function called `digit_count()` that takes a positive integer as a parameter and returns the number of digits in the integer. Hint: The number of digits increases by 1 whenever the input number is divided by 10.

Ex: If the input is:

345

©zyBooks 09/27/22 12:27 469702

the function `digit_count()` returns and the program outputs:

Steven Cameron
WGUC859v4

3

334598.939404.qx3zqy7

LAB
ACTIVITY

17.10.1: LAB: Count the digits

0 / 10



main.py

[Load default template...](#)

```
1 # TODO: Write recursive digit_count() function here.  
2  
3 if __name__ == '__main__':  
4     num = int(input())  
5     digit = digit_count(num)  
6     print(digit)
```

[Develop mode](#)

[Submit mode](#)

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

WGUC859v4

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above)



main.py
(Your program)



0

Program output displayed here

Coding trail of your work [What is this?](#)

©zyBooks 09/27/22 12:27 469702

History of your effort will appear here once you begin
working on this zyLab.

WGUC859v4

17.11 LAB: Drawing a right side up triangle



This section has been set as optional by your instructor.

Write a recursive function called `draw_triangle()` that outputs lines of '*' to form a right side up isosceles triangle. Function `draw_triangle()` has one parameter, an integer representing the base length of the triangle. Assume the base length is always odd and less than 20. Output 9 spaces before the first '*' on the first line for correct formatting.

Hint: The number of '*' increases by 2 for every line drawn.

Ex: If the input of the program is:

3

the function `draw_triangle()` outputs:

*

Ex: If the input of the program is:

19

©zyBooks 09/27/22 12:27 469702

Steven Cameron
WGUC859v4

the function `draw_triangle()` outputs:

*

The image shows a decorative border composed entirely of the asterisk character (*). It is arranged in a rectangular pattern with four horizontal rows and four vertical columns, meeting at the corners to form a complete frame.

wguc859v4

Note: No space is output before the first '*' on the last line when the base length is 19.

334598.939404.qx3zqy7

LAB ACTIVITY

17.11.1: LAB: Drawing a right side up triangle

0 / 10



main.py

Load default template...

```
1 # TODO: Write recursive draw_triangle() function here.
2
3
4 if __name__ == '__main__':
5     base_length = int(input())
6     draw_triangle(base_length)
```

Develop mode

Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above)

A black arrow pointing to the right, indicating the direction of the next section.

main.py (Your program)

0

Program output displayed here

Coding trail of your work [What is this?](#)

History of your effort will appear here once you begin
working on this zyLab.

©zyBooks 09/27/22 12:27 469702
Steven Cameron
WGUC859v4

17.12 LAB: Output a linked list



This section has been set as optional by your instructor.

Write a recursive function called `print_list()` that outputs the integer value of each node in a linked list. Function `print_list()` has one parameter, the head node of a list. The main program reads the size of the linked list, followed by the values in the list. Assume the linked list has at least 1 node.

Ex: If the input of the program is:

```
5
1
2
3
4
5
```

the output of the `print_list()` function is:

```
1, 2, 3, 4, 5,
```

Hint: Output the value of the current node, then call the `print_list()` function repeatedly until the end of the list is reached. Refer to the Node class to explore any available instance methods that can be used for implementing the `print_list()` function.

334598.939404.qx3zqy7

LAB
ACTIVITY

17.12.1: LAB: Output a linked list

0 / 10

main.py

[Load default template...](#)

```
1 class Node:  
2     def __init__(self, value):  
3         self.data_val = value  
4         self.next_node = None  
5  
6     def insert_after(self, node):  
7         tmp_node = self.next_node  
8         self.next_node = node  
9         node.next_node = tmp_node  
10  
11    def get_next(self):  
12        return self.next_node  
13  
14    def print_data(self):  
15        print(self.data_val, end=", ")  
16  
17 # TODO: Write recursive print_list() function here.
```

©zyBooks 09/27/22 12:27 469702
Steven Cameron
WGUC859v4

[Develop mode](#)[Submit mode](#)

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

[Run program](#)

Input (from above)



main.py
(Your program)



0

Program output displayed here

Coding trail of your work

[What is this?](#)

©zyBooks 09/27/22 12:27 469702

Steven Cameron

History of your effort will appear here once you begin working on this zyLab.