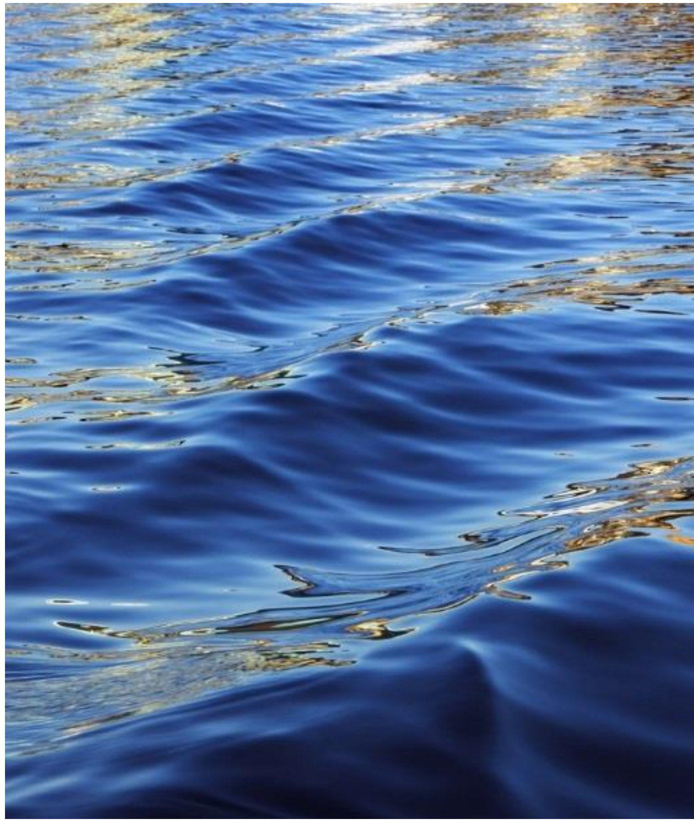
Recursion

Data structure and Algorithms (310-2101)



Recursion

What is Recursion?

Recursion is a method of solving problems that involves breaking a problem down into smaller and smaller subproblems until you get to a small enough problem that it can be solved trivially.

Usually, recursion involves a function calling itself. While it may not seem like much on the surface, recursion allows us to write elegant solutions to problems that may otherwise be very difficult to program.

Why we need recursion?

Recursion

Advantages of using recursion

- A complicated function can be split down into smaller sub-problems utilizing recursion.
- Sequence creation is simpler through recursion than utilizing any nested iteration.
- Recursive functions render the code look simple and effective.

Disadvantages of using recursion

- A lot of memory and time is taken through recursive calls which makes it expensive for use.
- Recursive functions are challenging to debug.
- The reasoning behind recursion can sometimes be tough to think through.

The Three Laws of Recursion

1. **A recursive algorithm must have a base case.**
2. **A recursive algorithm must change its state and move toward the base case.**
3. **A recursive algorithm must call itself, recursively.**

Base case: a condition to that allows the algorithm to **stop recursing**.

A base case is typically a problem that is small enough to solve directly.

Recursion in Python

Syntax

```
def func(): <--
              |
              | (recursive call)
              |
    func() ----
```

```
def dont_do_this():
    print("Hello World")
    dont_do_this()

dont_do_this()
```

Calculating the Sum of a List of Numbers

We will begin our investigation with a simple problem that you already know how to solve without using recursion. Suppose that you want to calculate the sum of a list of numbers such as: [1, 3, 5, 7, 9]. An iterative function that computes the sum is shown below.

```

def list_sum1(num_list):
    the_sum = 0

    for i in num_list:
        the_sum = the_sum + i
    return the_sum

```

แล้วถ้าหากเราไม่มี for loop / while loop

เราจะทำงานนี้ได้อย่างไร?

Calculating the Sum of a List of Numbers

Pretend for a minute that you do not have while loops or for loops.

How would you compute the sum of a list of numbers?

If you were a mathematician you might start by recalling that addition is a function that is defined for two parameters, a pair of numbers.

$$+$$

1	3	5	7	9

$$((((1 + 3) + 5) + 7) + 9)$$

$$(1 + (3 + (5 + (7 + 9))))$$

$$\text{total} = (1 + (3 + (5 + (7 + 9))))$$

$$\text{total} = (1 + (3 + (5 + 16)))$$

$$\text{total} = (1 + (3 + 21))$$

$$\text{total} = (1 + 24)$$

$$\text{total} = 25$$

Calculating the Sum of a List of Numbers

```python

def list_sum1(num_list):
    the_sum = 0

    for i in num_list:
        the_sum = the_sum + i
    return the_sum

```

```python
def list_sum2(num_list):
    if len(num_list) == 1:
        return num_list[0]
    else:
        return num_list[0] + list_sum2(num_list[1:])

```

Example of Recursion Quiz #1

Consider the following recursive function fun(x, y).

What is the value of fun(4, 3)

```python
def fun(x, y):
    if x == 0:
        return y
    return fun(x - 1, x + y)
```

Example of Recursion Quiz #2

Consider the following recursive function fun(n). What is the result on screen when using fun(5)

```python

def fun(n):
    if ( n<= 2 ):
        return 1
    t = fun(n-1)
    print(n)

```

What is the value of [t] for each iteration?

Example of Recursion Quiz #3

Demo2.py > ...

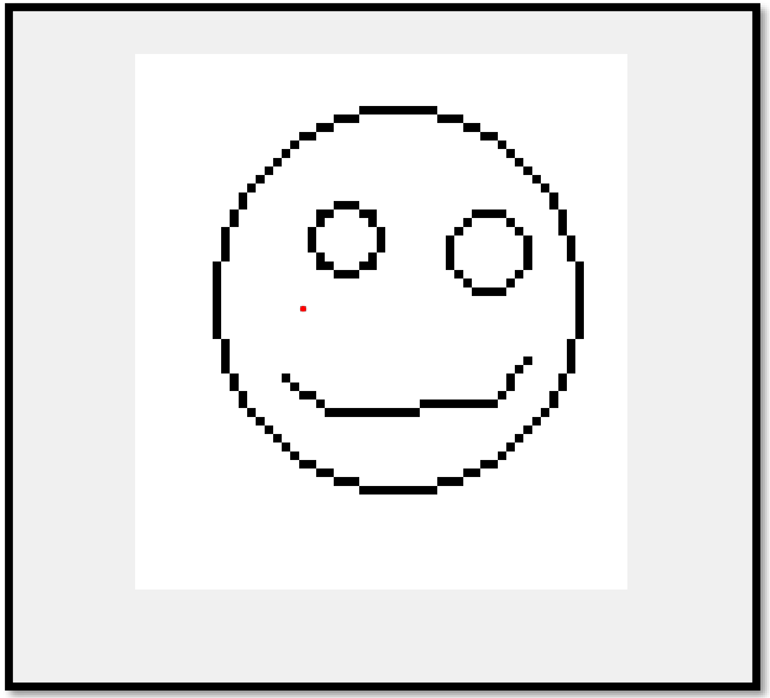
```python
def solve_maze(maze, start, end):
    def find_path(curr):
        x, y = curr

        if curr == end:
            return [curr]

        if maze[x][y] == 1:
            return []

        maze[x][y] = 1  # Mark as visited

        for next_move in [(x+1, y), (x-1, y), (x, y+1), (x, y-1)]:
            if 0 <= next_move[0] < len(maze) and 0 <= next_move[1] < len(maze[0]):
                path = find_path(next_move)
                if path:
                    return [curr] + path

        return []

    return find_path(start)
```

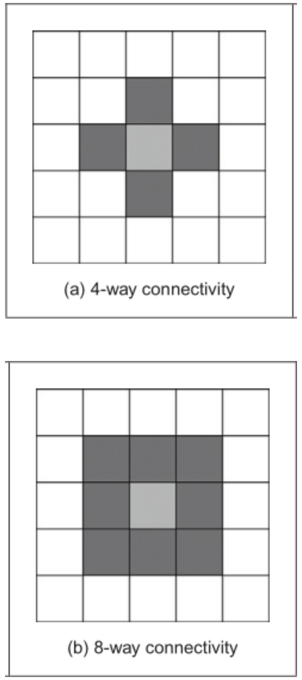
Example of Recursion Quiz #3

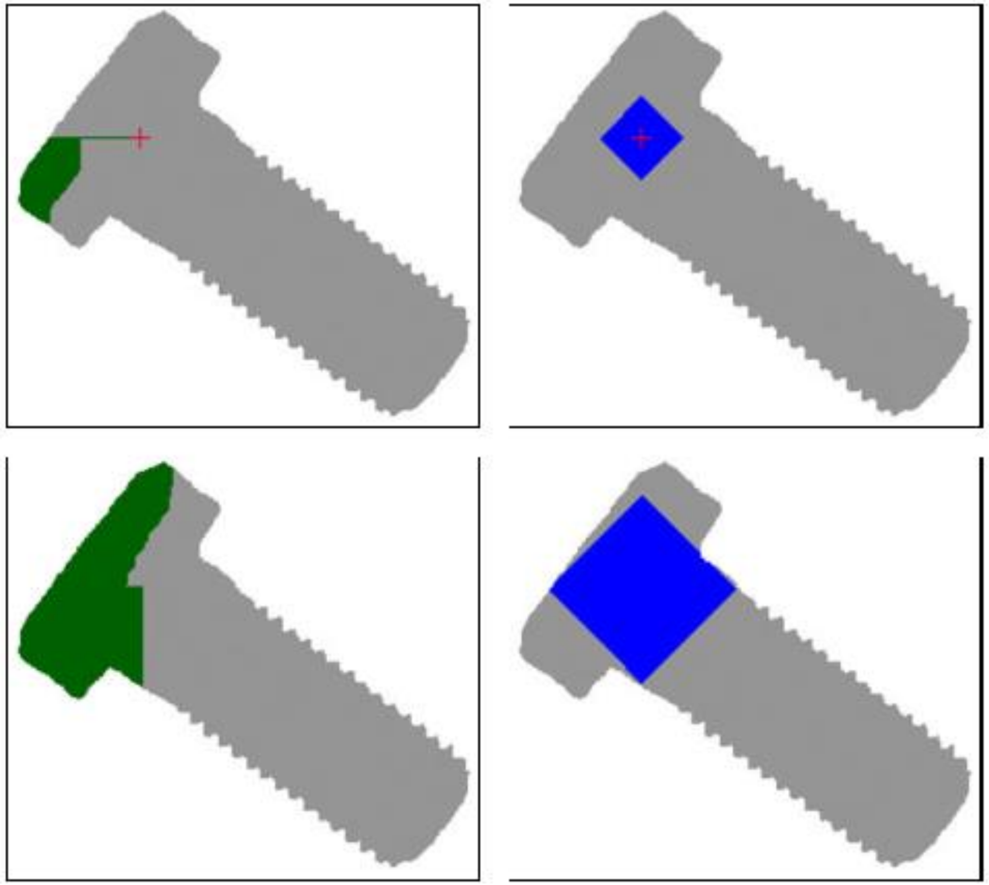
```python
if __name__ == '__main__':
    # Example maze represented as a 2D array
    example_maze = [
        [0, 1, 0, 0, 0],
        [0, 1, 0, 1, 0],
        [0, 0, 0, 0, 0],
        [0, 1, 1, 1, 0],
        [0, 0, 0, 0, 0]
    ]

    start_position = (0, 0)
    end_position = (4, 4)

    solution = solve_maze(example_maze, start_position, end_position)
    if solution:
        print(f"Solution path: {solution}")
    else:
        print("No solution found.")

```

Recursion Applications

Algorithm to filling color in object





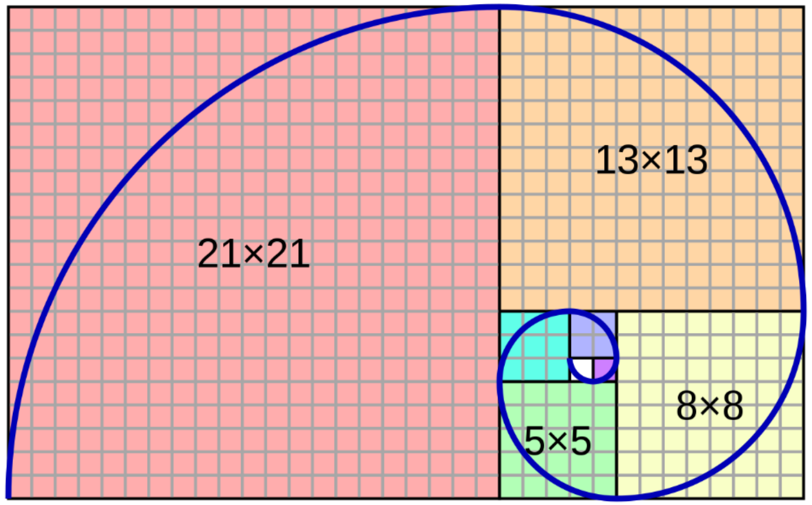


Fibonacci Sequence

The Fibonacci sequence is a series of numbers where a number is the addition of the last two numbers, starting with 0, and 1.

The Fibonacci Sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55...

$$X_n = X_{n-1} + X_{n-2}$$



The Fibonacci Spiral And The Golden Ratio

4_2_Fibo.py > ...

```python
# Print the fibonacci series upto n_terms
# Recursive function
def recursive_fibonacci(n):
    if n <= 1:
        return n
    else:
        return(recursive_fibonacci(n-1) + recursive_fibonacci(n-2))

if __name__ == '__main__':

    n_terms = 10

    # check if the number of terms is valid
    if n_terms <= 0:
        print("Invalid input ! Please input a positive value")
    else:
        print("Fibonacci series:")
    for i in range(n_terms):
        print(recursive_fibonacci(i))

    print("=== END PROGRAM ===")

```

Tower of Hanoi

Tower of Hanoi is a mathematical puzzle where we have three rods and n disks.
The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

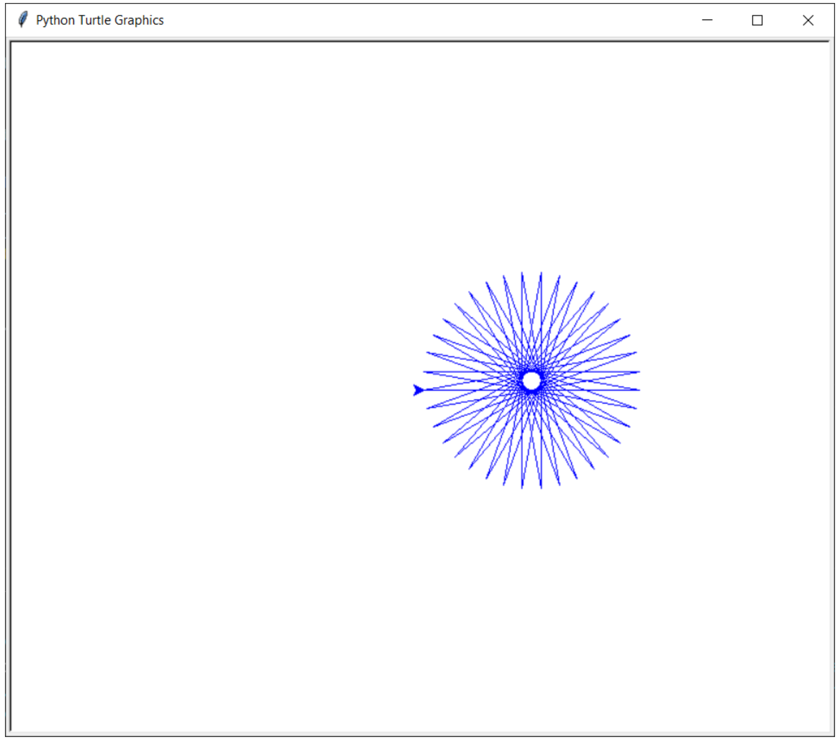
1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
3. No disk may be placed on top of a smaller disk.

Tower of Hanoi

```python

# Play with Hanoi
# https://www.mathsisfun.com/games/towerofhanoi.html

def move_tower(height, from_pole, to_pole, with_pole):
    if height >= 1:
        move_tower(height-1, from_pole, with_pole, to_pole)
        move_disk(from_pole, to_pole)
        move_tower(height-1, with_pole, to_pole, from_pole)

def move_disk(fp, tp):
    print("moving disk from", fp, "to", tp)

move_tower(3, "A", "C", "B")

```

Python Turtle Graphic

Turtle graphics is a popular way for introducing programming to kids. It was part of the original Logo programming language developed by Wally Feurzeig, Seymour Papert and Cynthia Solomon in



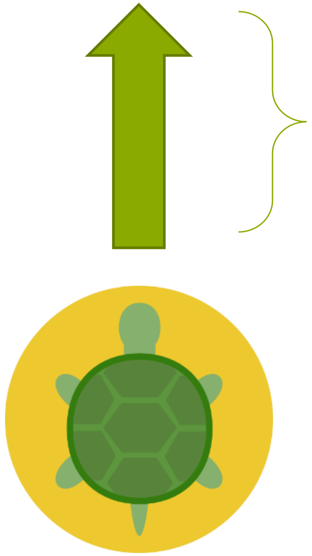
4_1_Turtle.py > ...

```python

import turtle

my_turtle = turtle.Turtle()
my_win = turtle.Screen()

my_turtle.color('blue')

while True:
    my_turtle.forward(200)
    my_turtle.left(170)
    if(abs(my_turtle.pos()) < 1):
        break

my_win.exitonclick()

```

Reference: https://docs.python.org/3/library/turtle.html

Python Turtle Graphic

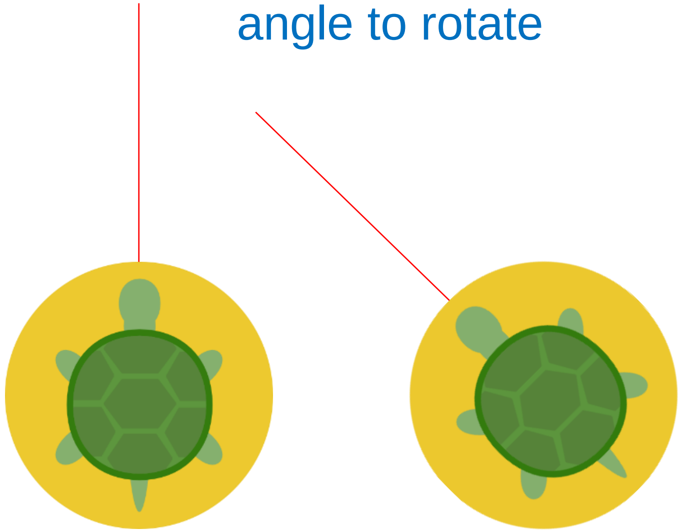
forward(…) backward(…)



points to move

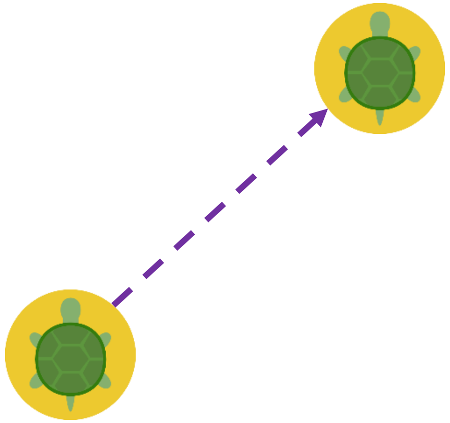
left(…) right(…)

angle to rotate



goto(… , …)

move turtle to an absolute position

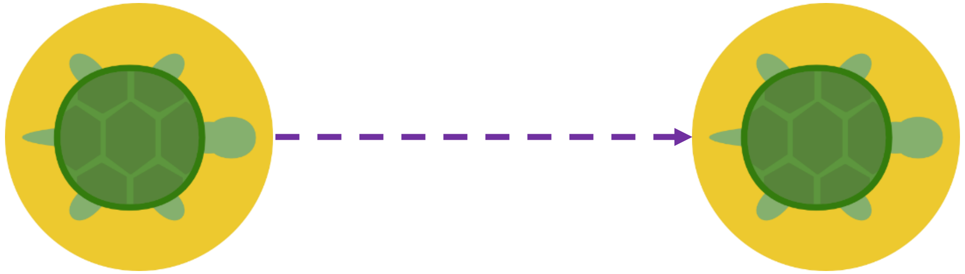


Reference: https://docs.python.org/3/library/turtle.html

Python Turtle Graphic

pendown(…) Drawing when moving

penup(…) No drawing when moving



Color control

`color()`

`pencolor()`

`fillcolor()`

Filling

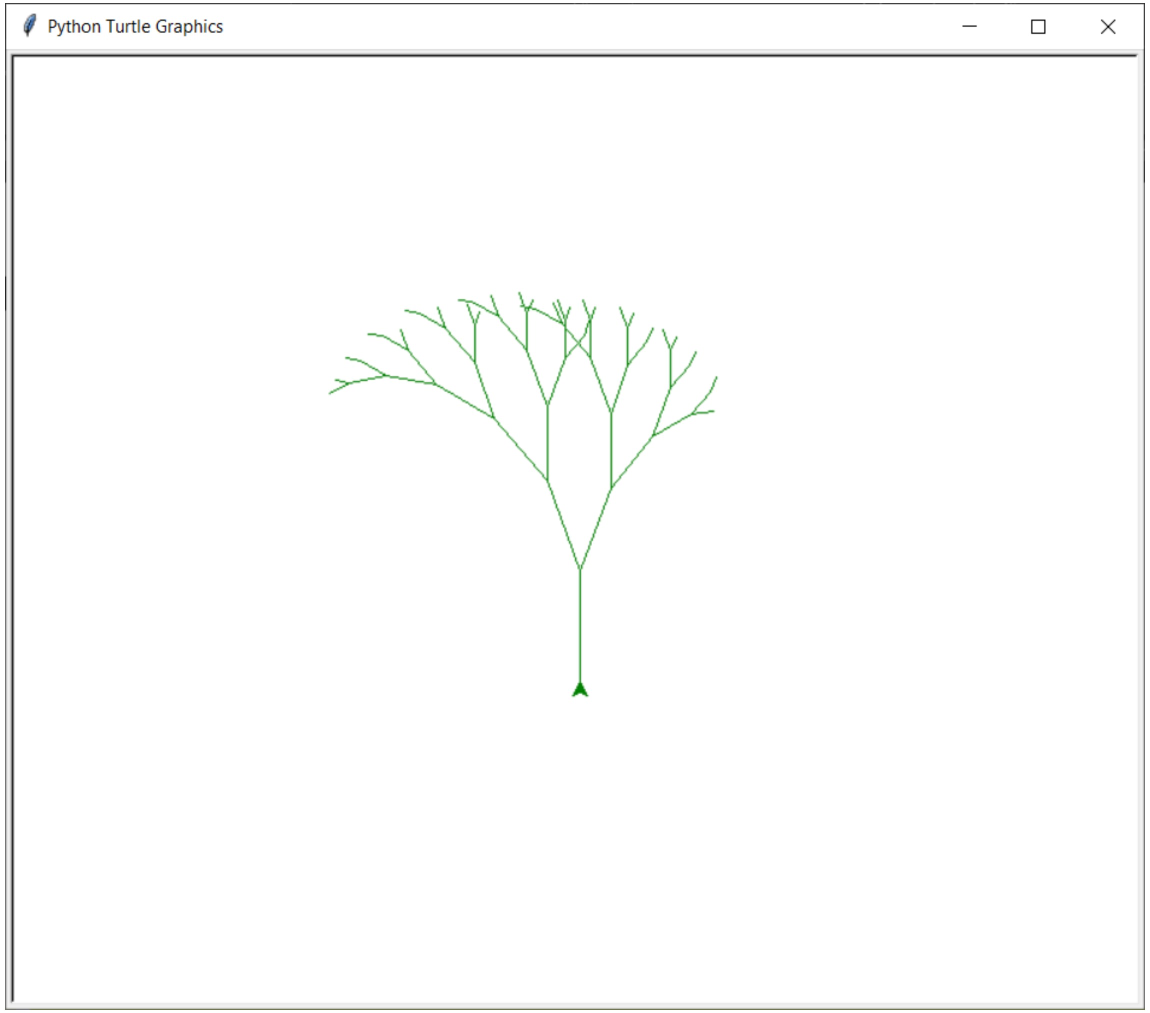
`filling()`

`begin_fill()`

`end_fill()`

Reference: https://docs.python.org/3/library/turtle.html

Python Turtle Graphic: Drawing a Tree with Turtle




```python
import turtle

def tree(branch_len, t):
    if branch_len > 5:
        t.forward(branch_len)
        t.right(20)
        tree(branch_len - 15, t)
        t.left(40)
        tree(branch_len - 10, t)
        t.right(20)
        t.backward(branch_len)

def main():
    t = turtle.Turtle()
    my_win = turtle.Screen()
    t.left(90)
    t.up()
    t.backward(100)
    t.down()
    t.color("green")
    tree(75, t)
    my_win.exitonclick()

main()
```

Sierpiński triangle

Sierpiński triangle is a fractal attractive fixed set with the **overall shape of an equilateral triangle**, **subdivided recursively into smaller equilateral triangles**.

The procedure for drawing a Sierpiński triangle by hand is simple.

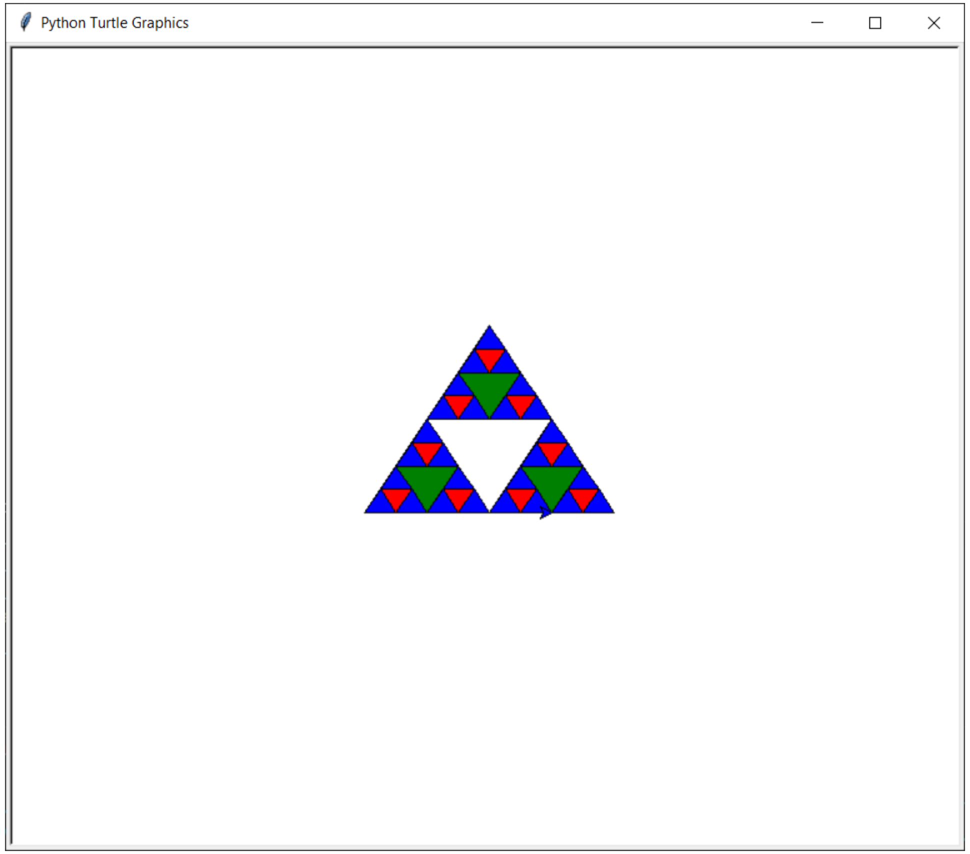
1. Start with a single large triangle.
2. Divide this large triangle into four new triangles by connecting the midpoint of each side.
3. Ignoring the middle triangle that you just created, apply the same procedure to each of the three corner triangles.
4. Each time you create a new set of triangles, you recursively apply this procedure to the three smaller corner triangles.

Sierpiński triangle

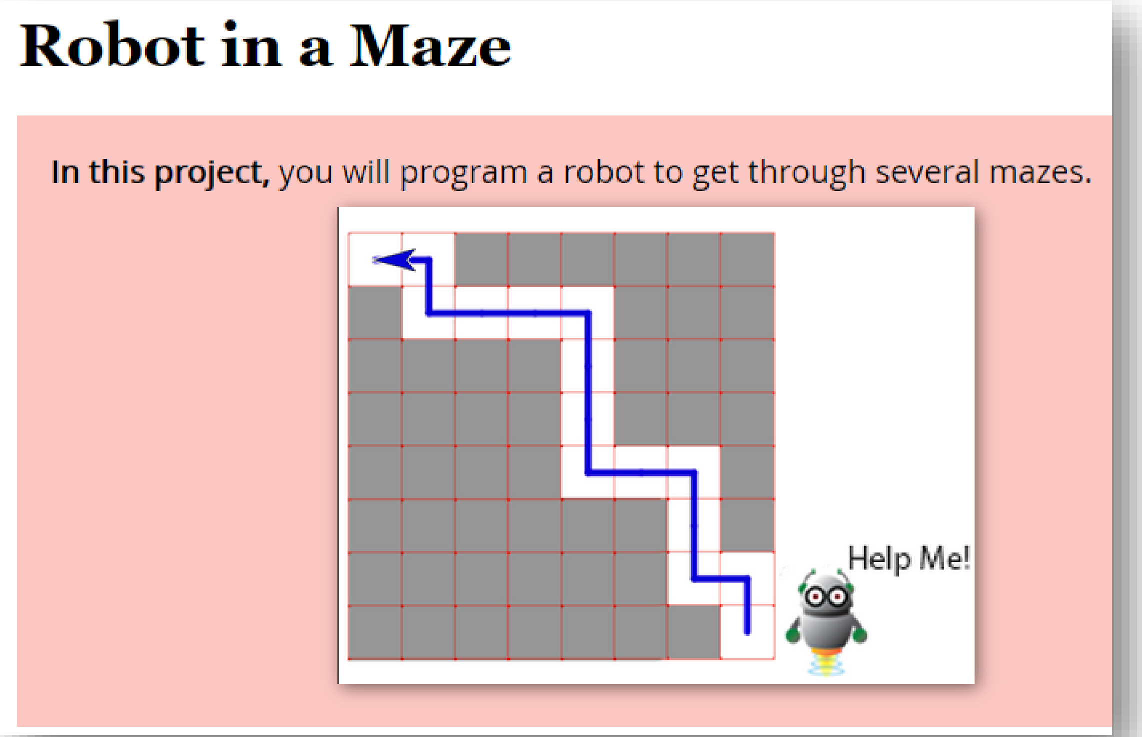
```python
import turtle

def draw_triangle(points, color, my_turtle):
    my_turtle.fillcolor(color)
    my_turtle.up()
    my_turtle.goto(points[0][0], points[0][1])
    my_turtle.down()
    my_turtle.begin_fill()
    my_turtle.goto(points[1][0], points[1][1])
    my_turtle.goto(points[2][0], points[2][1])
    my_turtle.goto(points[0][0], points[0][1])
    my_turtle.end_fill()

def get_mid(p1, p2):
    return ((p1[0] + p2[0])/2,  (p1[1] + p2[1])/2)

def sierpinski(points, degree, my_turtle):
    color_map = ['blue', 'red', 'green', 'white',
                 'yellow', 'violet', 'orange']

    draw_triangle(points, color_map[degree], my_turtle)
    if degree > 0:
        sierpinski(
            [points[0],
            get_mid(points[0], points[1]),
            get_mid(points[0], points[2])],
            degree-1, my_turtle
        )
        sierpinski(
            [points[1],
            get_mid(points[0], points[1]),
            get_mid(points[1], points[2])],
            degree-1, my_turtle
        )
        sierpinski(
            [points[2],
            get_mid(points[2], points[1]),
            get_mid(points[0], points[2])],
            degree-1, my_turtle
        )
```

Sierpiński triangle

```python
def main():
    my_turtle = turtle.Turtle()
    my_win = turtle.Screen()
    my_points = [[-100, -50], [0, 100], [100, -50]]
    sierpinski(my_points, 3, my_turtle)
    my_win.exitonclick()

main()
```



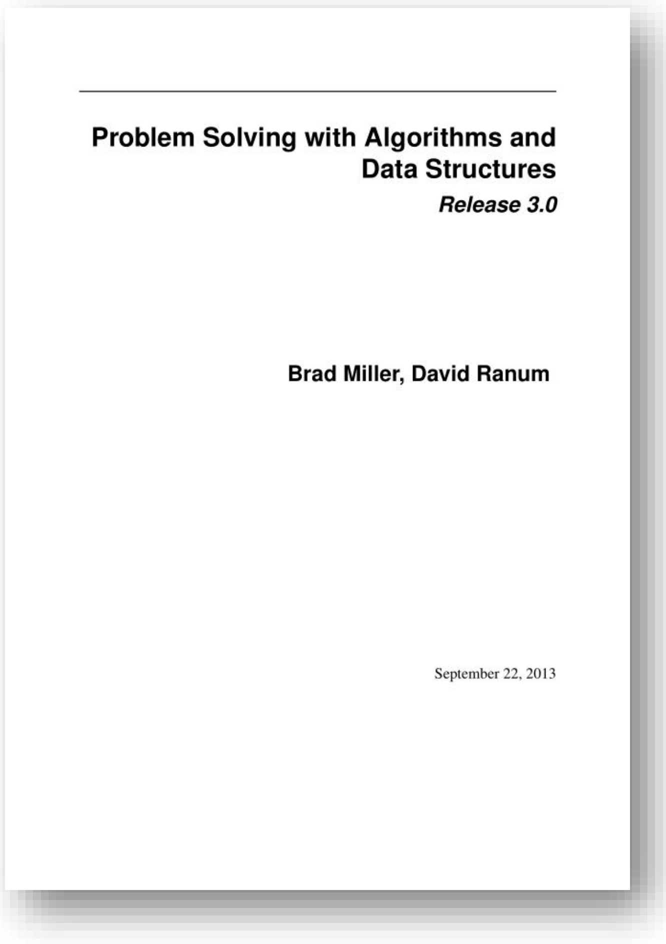
Exploring the Maze



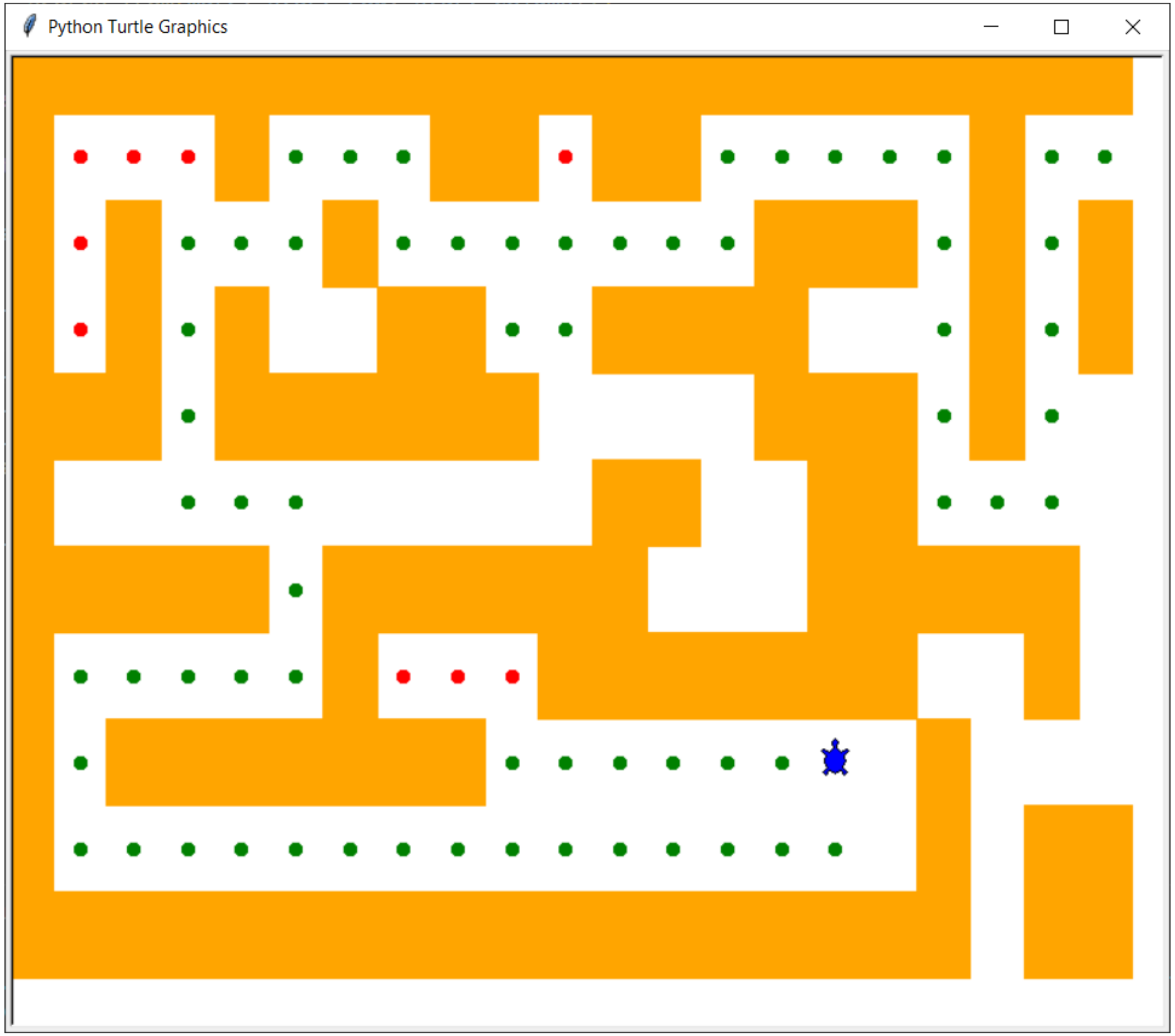
https://bjc.edc.org/March2019/bjc-r/cur/programming/2-complexity/4-abstraction/1-robot-in-a-maze.html

Beauty and Joy of Computing

Exploring the Maze



Page 141 - 144



Take Home Message / Things you can do now

[1] You should know about recursion concept.

[2] Can create / edit recursion function in Python

(Avoid an infinite loop in recursion function)

[3] Can explain what recursion function do…

[4] Can give an example of recursion applications.