```
#             0 2 4        12 4 6        24 6 8
gameBoard =
"1|2|3\n-----\n4|5|6\n-----\n7|8|9"
print(gameBoard)
gameBoard = " | | \n-----\n
| | \n-----\n | | "
roundNumber = 0
# playerMove = input("Enter your move: ")
#
gameBoard = gameBoard.replace(playerMove, "X")
gameIsNotOver = True

while
gameIsNotOver == True:
    if roundNumber % 2 == 0:
        currentPlayer = "X"

else:
        currentPlayer = "O"
    print("\n" + gameBoard)

playerMove = int(input(f"{currentPlayer}, Enter your move: ")) - 1

"""
    playMove-1, indexs, row,    col
    0 1 2     0 2 4   0 0 0   0 1 2

3 4 5    12 4 6   1 1 1   0 1 2
    6 7 8    24 6 8   2 2 2   0 1 2
    playerMove: 5
    row:
1
    col: 2
    index: 16 -> 12*row + col*2
    """
    row = playerMove //
3
    col = playerMove % 3
    index = row*12 + col*2
    gameBoard = gameBoard[:index] +
currentPlayer + gameBoard[index+1:]

    if gameBoard[0] != " " and gameBoard[0] ==
gameBoard[2] == gameBoard[4]: gameIsNotOver = False
    elif gameBoard[12] != " " and
gameBoard[12] == gameBoard[14] == gameBoard[16]: gameIsNotOver = False
    elif gameBoard[24]
!= " " and  gameBoard[24] == gameBoard[26] == gameBoard[28]: gameIsNotOver = False

 elif gameBoard[0] != " " and  gameBoard[0] == gameBoard[12] == gameBoard[24]:
gameIsNotOver = False
    elif gameBoard[2] != " " and  gameBoard[2] == gameBoard[14]
== gameBoard[26]: gameIsNotOver = False
    elif gameBoard[4] != " " and
gameBoard[4] == gameBoard[16] == gameBoard[28]: gameIsNotOver = False
    elif gameBoard[0] !=
" " and  gameBoard[0] == gameBoard[14] == gameBoard[28]: gameIsNotOver = False

elif gameBoard[4] != " " and  gameBoard[4] == gameBoard[14] == gameBoard[24]:
gameIsNotOver = False
    elif roundNumber == 8:
        currentPlayer = "Cat"

 gameIsNotOver = False


    roundNumber += 1



print("\n" +
gameBoard)
print(f"The winner of the game is {currentPlayer}")
```

```
import time
import
turtle
from geneticAlgorithm import *
from nn import NN

class Bird:
  def __init__(self, loc,
brain = None, size = [36, 36]):
    self.counter = 0
    self.t = turtle.Turtle()

self.t.shape("circle")
    self.t.up()
    self.startingPoint = loc

self.t.goto(loc)
    self.t.fillcolor(0.5, 1, 1)
    self.t.left(90)
    self.loc = loc

self.color = (255, 0, 0)
    self.size = size
    self.down = True
    self.score = 0

self.brain = NN([5, 8, 2, 1])
    if brain != None:
        self.brain.setState(brain)

  def
__ge__(self, other):
    return self.score >= other.score

  def __le__(self, other):

return self.score <= other.score

  def __gt__(self, other):
    return self.score >
other.score

  def __lt__(self, other):
    return self.score < other.score

  def
__eq__(self, other):
    return self.score == other.score

  def reset(self):

self.t.reset()
    self.counter = 0
    self.t.shape("circle")
    self.t.up()

self.t.fillcolor(0.5, 1, 1)
    self.t.left(90)
    self.t.goto(self.startingPoint)

self.t.showturtle()
    self.down = True
    self.score = 0

  def copy(self):
    return
Bird(self.loc, self.brain.copy())

  def think(self, pipe, top, bottom):
    inputs =
pipe.getColiderRec()
    for i in range(len(inputs)):
      inputs[i] = mapFunction(inputs[i],
bottom, top)
```

```
    inputs.append(mapFunction(self.t.pos()[1], bottom, top))
    action =
self.brain.predict(inputs)
    return action[1] > action[0]

class Pipe:
  def
__init__(self, screenSize, gap, wn):
    wn.tracer(False)
    self.pipe = turtle.Turtle()

self.gap = turtle.Turtle()
    self.gap.up(), self.pipe.up()

self.gap.shape("square"), self.pipe.shape("square")

self.gap.fillcolor(1,1,1), self.pipe.fillcolor(0,1.0,0)
    self.gap.pencolor(1,1,1),
self.pipe.pencolor(1,1,1)
    self.screenSize = screenSize
    self.gap.shapesize(gap//20, 5),
self.pipe.shapesize(screenSize[1]//(20), 5)
    self.gapWidth = 5*20
    self.gapHeight = gap

  self.ranomizePipe(screenSize)

  def getColiderRec(self):
    x, y = self.gap.pos()
    x1,
y1 = int(x) - self.gapWidth//2, int(y) - self.gapHeight//2 + 12
    x2, y2 = int(x) +
self.gapWidth//2 , int(y) + self.gapHeight//2 - 12
    return [x1, y1, x2, y2]

  def
moveForward(self, d):
    self.pipe.backward(d)
    self.gap.backward(d)

  def __str__(self):

   return str(self.loc) + str(self.size)

  def ranomizePipe(self, screen):
    screenX,
screenY = screen
    h = random.randint(-screenY//2, screenY//2)
    self.gap.setx(screenX//2),
self.gap.sety(h)
    self.pipe.setx(screenX//2)

class flappBGame:
  def __init__(self, count,
brain = None):
    self.wn = turtle.Screen()
    self.wn.listen()
    self.pipeWidth = 60

self.down = True
    total = 0
    self.screenTop, self.screenBottom =
self.wn.window_height()//2, -self.wn.window_height()//2
    gap, gravity, pipeSpeed, counter =
250, 3, 4, 0
    pipes = [Pipe((self.wn.window_width(), self.wn.window_height()), gap,
self.wn)]
    self.birds, self.savedBirds = [], []
    for i in range(count):

self.birds.append(Bird([-self.wn.window_width()//2 + self.wn.window_width()*0.2, 0], brain))

 newPipeRate = 4.5
    newPipeTime = time.time()
    self.start = time.time()
```

```
    play = True

  self.wn.tracer(False)
    while play:
      max = 0
      if newPipeTime + newPipeRate <=
time.time():
        pipes.append(Pipe((self.wn.window_width(), self.wn.window_height()), gap,
self.wn))
        newPipeTime = time.time()
      for aPipe in pipes:
aPipe.moveForward(pipeSpeed)

      for bird in self.birds:
        if len(pipes) == 0:

  newPipeTime = time.time()
          pipes = [Pipe((self.wn.window_width(),
self.wn.window_height()), gap, self.wn)]
        bird.down = bird.think(pipes[0],
self.screenTop, self.screenBottom)
        if bird.down:
          bird.down = False

bird.counter = 8
        if bird.counter > 0:
          bird.t.forward(gravity*2)

bird.counter -= 1
        else: bird.t.backward(gravity)
        if
self.colision(pipes[0].getColiderRec(), bird):
          bird.t.hideturtle()

bird.score = int(time.time() - self.start) + 1
          if bird.score > max:

max = bird.score
          total += bird.score
          self.birds.remove(bird)

self.savedBirds.append(bird)
          if len(self.birds) == 0:

            print("Max
score from previous generation: ", max)
            for temp in pipes:

temp.pipe.reset()
              temp.gap.reset()

self.wn.turtles().remove(temp.pipe)
              self.wn.turtles().remove(temp.gap)

 self.birds = nextGeneration(self.savedBirds, total)

            for bird in self.birds:
bird.reset()
            self.savedBirds = []
            pipes =
[Pipe((self.wn.window_width(), self.wn.window_height()), gap, self.wn)]
            newPipeTime
= time.time()
            self.start = time.time()
            total = 0
      if
pipes[0].getColiderRec()[0] < - self.wn.window_width()/2:
        temp = pipes.pop(0)

 temp.pipe.reset()
        temp.gap.reset()
        self.wn.turtles().remove(temp.pipe)

self.wn.turtles().remove(temp.gap)
      self.wn.update()
```

```
    self.wn.clear()

  def
spaceBar(self):
    self.down = not(self.down)

  def colision(self, coliderRec, bird):

birdX, birdY = bird.t.pos()
    birdX = int(birdX)
    y = self.wn.window_height()//2
    if
coliderRec[0] in range(birdX - self.pipeWidth, birdX):
      if not(birdY in
range(coliderRec[1], coliderRec[3])):
        return True
    if birdY in range(-y, y):


return False
    else:
      return True

while True:
  game = flappBGame(50)

from queue
import Queue
from PIL import Image

class Pixel:
  def __init__(self, loc, color):
    self.x =
loc[0]
    self.y = loc[1]
    self.loc = loc
    self.color = color
    self.r = color[0]

self.g = color[1]
    self.b = color[2]

  def __str__(self):
    return str(self.loc) +
": " + str(self.color)

  def __repr__(self):
    return str(self)




def
checkColors(a, b, tolerance = 1):
  if abs(a[0]- b[0])/100.0 > tolerance: return False
  if
abs(a[1]- b[1])/100.0 > tolerance: return False
  if abs(a[2]- b[2])/100.0 > tolerance:
return False
  return True

def fill(img, start, color):
  pixelsL = img.load()
  q = Queue()


 inside = pixelsL[start[0],start[1]]


  outside = color
  q.enqueue(Pixel(start, inside))


while q.length >= 1:
    p = q.dequeue()
    edgePix = [Pixel((p.x+1, p.y), pixelsL[p.x+1,
```

```
p.y]),
               Pixel((p.x-1, p.y), pixelsL[p.x-1, p.y]),
               Pixel((p.x,
p.y+1), pixelsL[p.x, p.y+1]),
               Pixel((p.x, p.y-1), pixelsL[p.x, p.y-1])]
    for
pix in edgePix:
      if checkColors(pix.color, inside, 0.8):
        q.enqueue(pix)

pixelsL[pix.x, pix.y] = outside
  return img

img = Image.open("pic.jpg")
import
time
s = time.time()
img = fill(img, (300, 200), (255, 255, 255))
img = fill(img, (190, 100),
(255, 0, 0))
img = fill(img, (250, 100), (0, 255, 0))
print(time.time() -
s)
img.save("floodFillPic.jpg")


import turtle
import random

class Cell:
    def
__init__(self, loc, size = 20, vis = False, color = None, offset = -250):
        if vis:

      self.t = turtle.Turtle()
            self.t.shape("square")

self.t.penup()
            self.t.speed(0)
            self.t.setpos(loc)

self.t.pensize(5)
            self.t.color(color)
            x,y = loc

self.t.goto(x*size+offset,y*size+offset)
        self.vis = vis
        self.loc = loc

self.color = color

    def setColor(self, color):
        self.color = color

self.t.color(color)


    def __str__(self):
        return str(self.loc)

    def
__repr__(self):
        return str(self)

    def setPos(self, loc):
        if self.vis:

      self.t.goto(loc)
        self.loc = loc

    def getPos(self):
        return
self.loc

class floodItGame:
```

```
    def __init__(self, boardSize = 20, cellSize = 20, numColors=
6):
        self.cells = []
        colors = ["red",
"blue","green","yellow","purple","orange"]

     size = 20
        self.boardSize = boardSize
        for i in range(boardSize):

 temp = []
            for j in range(boardSize):
                temp.append(Cell((i, j),
cellSize, True, random.choice(colors)))
            self.cells.append(temp)


self.coloredCells = [self.cells[0][0]]

    def getFringe(self):
        fringe, temp = [], []

       for cell in self.coloredCells:
            x,y = cell.getPos()
            if y <
self.boardSize-1: temp.append(self.cells[x][y+1])
            if y > 0:
temp.append(self.cells[x][y - 1])
            if x < self.boardSize-1:
temp.append(self.cells[x+1][y])
            if x < 0: temp.append(self.cells[x-1][y])


  for cell in temp:
            if cell.color != self.coloredCells[0].color:

fringe.append(cell)
        return fringe

    def getColoredCells(self):

self.coloredCells, temp = [], [self.cells[0][0]]
        while temp != []:
            cell =
temp.pop()
            if cell.color == self.cells[0][0].color and cell not in
self.coloredCells:
                self.coloredCells.append(cell)
                x, y =
cell.getPos()
                if y < self.boardSize-1: temp.append(self.cells[x][y + 1])

            if y > 0: temp.append(self.cells[x][y - 1])
                if x <
self.boardSize-1: temp.append(self.cells[x + 1][y])
                if x < 0:
temp.append(self.cells[x - 1][y])




    def getMoves(self):
        fringe =
self.getFringe()
        fringeColors = []
        for cell in fringe:

fringeColors.append(cell.color)
        return list(set(fringeColors))

    def makeMove(self,
decide):
        # self.getFringe() #maybe redundent
        colors = self.getMoves()
```

```
color = decide(colors)
        for cell in self.coloredCells:
            cell.setColor(color)

       self.getColoredCells()
        return len(self.coloredCells) == self.boardSize**2

def
randomComputer(colors):
    return random.choice(colors)

def humanPlayer(colors):
    for i,
color in enumerate(colors):
        print(i, color)
    return colors[int(input("Color:
"))]

def AI(colors):
    pass

wn = turtle.Screen()
wn.tracer(0,0)
game =
floodItGame(60)
game.getColoredCells()
gameOver = False
while gameOver == False:

wn.update()
    gameOver =
game.makeMove(randomComputer)

wn.update()
wn.exitonclick()




import random
from nn import
*

# resets birds to starting state and sets score to 0
def resetGame(birds):
  for bird in
birds: bird.reset()
  return birds

# This function creates a list of birds that evolve from
the previous generation
# The bird list maintains the same lenght, all bird data is reset.
#
The bird neural net is updated in the generate function
# This function generates a pool of
bird NNs where the number
# of times that the bird is added to the list equal to the bird's
#
performance so that the birds that perform the best are most likely
# be better represented in
the next generation of birds.  After the
# next generation of birds is selected, each bird NN
is mutated slightly.
def nextGeneration(oldBirds, total):
  worst, best = min(oldBirds),
max(oldBirds)
  data = open("nnData" + str(best.brain.hiddenNodes) +
".txt", "r").read().split("\n")
  bestScores = [0]
  for line in
data:
      line = line.split(" : ")
      if int(line[0]) ==
best.brain.hiddenNodes:
        bestScores.append(float(line[1]))
```

```
  if max(bestScores) <
best.score:
      f = open("nnData" + str(best.brain.hiddenNodes) +
"b.txt", "a")
      f.write("\n" + str(best.brain.hiddenNodes) +
" : " +  str(best.score) + " : " + str(best.brain.getNetwork()))
  if
worst.score < 1:
      total = 0
      for bird in oldBirds:
          bird.score =
mapFunction(bird.score, 0, best.score, 5, 1)
          total += bird.score

  birdPool = []

for bird in oldBirds:
    fitness = int((100*bird.score**2) / total)

birdPool.extend([bird.brain]*fitness)
  for i in range(len(oldBirds)-1):
    r =
random.randint(0, len(birdPool)-1)
    oldBirds[i].brain = birdPool[r].copy()

oldBirds[i].brain.mutateNN()
  return oldBirds[:]

import numpy
import random
from copy import
deepcopy
class NN:
  def __init__(self, structure = [], defaults = []):
    from math import e

     self.e = e
    if len(structure) == 3: inputs, hidden, outputs = structure
    elif
len(structure) == 4: inputs, hidden, outputs, hiddenLayers = structure
    if defaults == []:

      self.biass = [2 * numpy.random.rand(hidden) - 1]
        self.weights = [2 *
numpy.random.rand(hidden, inputs) - 1]
        for i in range(hiddenLayers - 1):

self.weights.append(2 * numpy.random.rand(hidden, hidden) - 1)
            self.biass.append(2
* numpy.random.rand(hidden, hidden) - 1)
        self.weights.append(2 *
numpy.random.rand(outputs, hidden) - 1)
        self.biass.append(2*numpy.random.rand(outputs)
- 1)

    else:
        self.weights = defaults[0]
        self.biass = defaults[1]

hidden = len(self.weights[0])
    self.hiddenNodes = hidden

  def sigmoid(self, x):
    if -x
> numpy.log(numpy.finfo(type(x)).max):
      return 0.0
    return 1 / (1 + self.e**(-x))

def predict(self, inputs):
    layer = numpy.array(inputs)
    for i in
range(len(self.weights)):
        layer = self.weights[i].dot(layer) + self.biass[i]

for row in range(len(layer)):
```

```
            val = layer[row]
            layer[row] =
self.sigmoid(val)
    return numpy.array(layer)
  def mutateNN(self, rate=0.1):
    def
mutation(array):
      temp = []
      for i in range(len(array)):
        if random.random()
< rate: temp.append(random.gauss(0, 1))
        else: temp.append(0)
      tempArray =
numpy.array(temp)
      return numpy.add(array, tempArray)

    for i in
range(len(self.weights)):
        for j in range(len(self.weights[i])):

self.weights[i][j] = mutation(self.weights[i][j])
        self.biass[i] =
mutation(self.biass[i])
  def copy(self):
    return NN(defaults=self.getState())

  def
getState(self):
      return [deepcopy(self.weights), deepcopy(self.biass)]
  def
getNetwork(self):
      l = [[], []]
      for array in self.weights:

l[0].append(array.tolist())
      for array in self.biass:

l[1].append(array.tolist())
      return l
  def setState(self, defaults):
    self.weights =
[]
    for aList in defaults[0]:
        self.weights.append(numpy.array(aList))
    self.biass
= []
    for aList in defaults[1]:
        self.biass.append(numpy.array(aList))

def
mapFunction(n, oldMin, oldMax, newMax = 1, newMin = 0):
    return (((n - oldMin) * (newMax -
newMin)) / (oldMax - oldMin)) + newMin


class Node:
    def __init__(self,initdata):

self.data = initdata
        self.next = None
        self.previous = None

    def
getData(self):
        return self.data

    def getNext(self):
        return self.next


def getPrevious(self):
        return self.previous

    def setData(self,newdata):
```

```
self.data = newdata

    def setNext(self,newnext):
        self.next = newnext

    def
setPrevious(self,newprevious):
        self.previous = newprevious

    def __str__(self):

 return "Node: " + str(self.data)



import random
from nn import *

# resets birds
to starting state and sets score to 0
def resetGame(birds):
  for bird in birds: bird.reset()

return birds

# This function creates a list of birds that evolve from the previous
generation
# The bird list maintains the same lenght, all bird data is reset.
# The bird neural
net is updated in the generate function
# This function generates a pool of bird NNs where the
number
# of times that the bird is added to the list equal to the bird's
# performance so that
the birds that perform the best are most likely
# be better represented in the next generation
of birds.  After the
# next generation of birds is selected, each bird NN is mutated
slightly.
def nextGeneration(oldBirds, total):
  worst, best = min(oldBirds), max(oldBirds)

data = open("nnData" + str(best.brain.hiddenNodes) + ".txt",
"r").read().split("\n")
  bestScores = [0]
  for line in data:
      line =
line.split(" : ")
      if int(line[0]) == best.brain.hiddenNodes:

bestScores.append(float(line[1]))
  if max(bestScores) < best.score:
      f =
open("nnData" + str(best.brain.hiddenNodes) + "b.txt", "a")

f.write("\n" + str(best.brain.hiddenNodes) + " : " +  str(best.score) +
" : " + str(best.brain.getNetwork()))
  if worst.score < 1:
      total = 0

for bird in oldBirds:
          bird.score = mapFunction(bird.score, 0, best.score, 5, 1)

    total += bird.score

  birdPool = []
  for bird in oldBirds:
    fitness =
int((100*bird.score**2) / total)
    birdPool.extend([bird.brain]*fitness)
  for i in
range(len(oldBirds)-1):
    r = random.randint(0, len(birdPool)-1)
    oldBirds[i].brain =
birdPool[r].copy()
```

```
    oldBirds[i].brain.mutateNN()
  return oldBirds[:]

import random
from
copy import deepcopy

class sudoku:
    def __init__(self):
        self.gameState = []

for i in range(9):
            temp = []
            for j in range(9):

temp.append(" ")
            self.gameState.append(temp)

    def getRowValues(self,
row):
        return set(self.gameState[row]) - set(" ")

    def getColValues(self,
col):
        l = []
        for i in range(9):

l.append(str(self.gameState[i][col]))
        return set(l) - set(" ")

    def
getBlockValues(self, row, col):
        l = []
        row = row//3*3
        col = col//3*3

     for i in range(row, row+3):
            for j in range(col, col+3):

l.append(str(self.gameState[i][j]))
        return set(l) - set(" ")

    def
getGameState(self):
        s = ""
        for aList in self.gameState:

s+= str(aList)
        return s

    def insert(self, row, col, value):
        if str(value)
in self.getColValues(col): return
        if str(value) in self.getRowValues(row): return

  if str(value) in self.getBlockValues(row, col): return
        self.gameState[row][col] =
value

    def remove(self, row, col):
        self.gameState[row][col] = " "


def gameBoardCompleted(self):
        for aRow in self.gameState:
            if " "
in aRow:
                return False
        return True
        x = 7

    def
getOpenSpaces(self):
        l = []
        for row in range(len(self.gameState)):
```

```
for col in range(len(self.gameState)):
                if self.gameState[row][col] == "
":
                    l.append((row, col))
        return l

    def createPuzzel(self,
emptySpaces):
        while len(self.getOpenSpaces()) < emptySpaces:
            row, col =
random.randint(0, 8), random.randint(0,8)
            if self.gameState[row][col] != "
":
                self.remove(row, col)

    def startGameBoard(self):
        l =
["0","1","2","3","4","5","6&qu
ot;,"7","8"]
        random.shuffle(l)
        self.gameState[0] = l


  row0C3_8 = l[3:]
        random.shuffle(row0C3_8)
        row0C1_2 = l[1:3]

random.shuffle(row0C1_2)
        self.gameState[1][0] = str(row0C3_8[0])

self.gameState[2][0] = str(row0C3_8[1])
        row0C3_8 = row0C3_8[2:] + row0C1_2

random.shuffle(row0C3_8)
        for i, item in enumerate(row0C3_8):

self.gameState[i+3][0] = str(item)
        gridL = list(set(l) - self.getBlockValues(0,0))

   random.shuffle(gridL)
        self.gameState[1][1] = gridL[0]
        self.gameState[1][2] =
gridL[1]
        self.gameState[2][1] = gridL[2]
        self.gameState[2][2] = gridL[3]

 temp = set(l) -  set(self.getBlockValues(0, 3) | self.getRowValues(1))
        forSure =
list(temp & self.getRowValues(2) & self.getBlockValues(0,6))
        temp = list(temp)

       random.shuffle(temp)
        temp = list(set(temp) - set(forSure))
        gridL =
forSure + temp[:3-len(forSure)]
        random.shuffle(gridL)
        self.gameState[1][3] =
gridL[0]
        self.gameState[1][4] = gridL[1]
        self.gameState[1][5] = gridL[2]

 gridL = list(set(l) -  self.getRowValues(1))
        random.shuffle(gridL)

self.gameState[1][6] = gridL[0]
        self.gameState[1][7] = gridL[1]

self.gameState[1][8] = gridL[2]
        gridL = list(set(l) -  self.getBlockValues(0,3))

 random.shuffle(gridL)
        self.gameState[2][3] = gridL[0]
        self.gameState[2][4] =
gridL[1]
```

```
        self.gameState[2][5] = gridL[2]
        gridL = list(set(l) -
self.getBlockValues(0,6))
        random.shuffle(gridL)
        self.gameState[2][6] =
gridL[0]
        self.gameState[2][7] = gridL[1]
        self.gameState[2][8] = gridL[2]


def __str__(self):
        s = "-"*19 + "\n|"
        for aRow in
self.gameState:
            for char in aRow:
                s+= char + "|"

   s+= "\n" + "-"*19 + "\n" + "|"
        return s[:-1]
+ "\n\n"

    def getPossibleValues(self, row, col):
        l =
set(["0","1","2","3","4","5","
6","7","8"])
        return l - set(self.getBlockValues(row, col) |
self.getRowValues(row) | self.getColValues(col))

    def
recursiveBacktrackingAlgorithmToFillBoard(self):
        openSpaces = self.getOpenSpaces()

   if len(openSpaces) == 0: return True
        # ((row, col), avaliableMovesAt (row, col))

    min, minList = ((-1,-1), [""]*10), []
        for row, col in openSpaces:

    avaliableMoves = self.getPossibleValues(row, col)
            numMoves =
len(avaliableMoves)
            if numMoves < len(min[1]):
                min = ((row,
col), list(avaliableMoves))
                minList = [min]
            elif numMoves ==
len(min[1]):
                minList.append(((row, col), list(avaliableMoves)))
        if
len(min[1]) == 0:
            return False
        else:
            min =
random.choice(minList)
            move = random.choice(min[1])
            row, col = min[0]

          self.insert(row, col, move)
            return
self.recursiveBacktrackingAlgorithmToFillBoard()

    def recursivelyFillBoard(self):

board = False
        backUp = deepcopy(self.gameState)
        while board == False:

  board = self.recursiveBacktrackingAlgorithmToFillBoard()
            if board == False:

          self.gameState = deepcopy(backUp)

    def randomlyCreateGameBoard(self):

self.startGameBoard()
```

```
        self.recursivelyFillBoard()


gb =
sudoku()
gb.randomlyCreateGameBoard()
print(gb)
gb.createPuzzel(60)
print(gb)
gb.recursivelyFil
lBoard()
print(gb)


#
#
import turtle
import random


def mapFunction(n=0, oldMin=0, oldMax=1,
newMin = 0, newMax = 1):
    newValue =  (((n - oldMin) * (newMax - newMin)) / (oldMax -
oldMin)) + newMin

    return newValue

def distanceFunction(x1, y1, x2, y2):
    a = x1- x2

 b = y1 - y2
    dist = (a**2 + b**2)**0.5

    return dist



dist = distanceFunction(0,0,6,8)
#veghtyu
dist1 = distanceFunction(0,0,3,4) #rgrdehjty
print(dist, dist1) #
wrgethryjtuy
print(dist + dist1) # rghtrjyuk

t = turtle.Turtle() #sgdhfjg
wn =
turtle.Screen()
wn.tracer(0)
r1, g1, b1 = random.random(), random.random(), random.random()
r2,
g2, b2 = random.random(), random.random(), random.random()
t.color(r1,g1,
b1)
t.shape("square")
w, h, cellSize = 700, 700, 21
t.up()
y = 0
maxDist =
distanceFunction(-w//2, -h // 2, w//2, h // 2)
print(maxDist)
for x in range(-w//2, w//2,
cellSize):
    for y in range(-h // 2, h // 2, cellSize):
        t.goto(x,y)
        dist =
distanceFunction(x,y, w//2, h//2)
        newRed = mapFunction(dist, 0, maxDist, r1, r2)

 newGreen = mapFunction(dist, 0, maxDist, g1, g2)
        newBlue = mapFunction(dist, 0,
maxDist, b1, b2)
        t.color(newRed, newGreen, newBlue)

t.stamp()
```

```
wn.exitonclick()


from LineClass1 import Line
from PointClass1 import Point
from
NumberPlane import drawNumberPlane
import turtle

class Circle:
    def __init__(self,
points=[], center=None, radius=None, plot=True):
        if points != []:

"""
            compute the center and radius
            1   create a line
segemtents from p1 to p2
                create a line from p2 to p3
            2   get the
midpoint between p1 and p2
                get the midpoint between p2 and p3
            3
create perp lines
            4 boom find the intersection of the perp lines

"""
            p1, p2, p3 = points
            lineSegement1 =
Line(startPoint=p1, endPoint=p2, plot=False)
            lineSegement2 = Line(startPoint=p2,
endPoint=p3, plot=False)
            lineSegement1MidPoint = p1.getMidpointBetween(p2,
plot=False)
            lineSegement2MidPoint = p2.getMidpointBetween(p3, plot=False)

  lineSegement1Perp = lineSegement1.findPerpindicularLine(lineSegement1MidPoint, plot=False)

         lineSegement2Perp = lineSegement2.findPerpindicularLine(lineSegement2MidPoint,
plot=False)

            self.center =
lineSegement1Perp.findPointOfIntersection(lineSegement2Perp, label=False, plot=True)

 self.radius = self.center.getDistanceBetween(p1)

        else:
            self.center,
self.radius = center, radius

        if plot:
            # pointOnCircumference = self.center
- Point(0, self.radius)
            # pointOnCircumference.t.circle(self.radius)

self.center.setY(self.center.getY() - self.radius)
            self.center.t.down()

# self.center.t.hideturtle()
            self.center.t.circle(self.radius)

self.center.t.up()
            self.center.setY(self.center.getY() + self.radius)


    def
__contains__(self, point):
        return self.center.getDistanceBetween(point) <
self.radius


    def __str__(self):
        x,y = self.center.getXY()
        return f"(x
```

```
- {x})^2 + (y-{y})^2 ={self.radius}^2"

if __name__ == "__main__":
    wn =
turtle.Screen()

    drawNumberPlane(20, wn)
    p1 = Point(-180, -100, color="red")

   p2 = Point(-40, 80, color="green")
    points = [Point(-150, 50, label=False),
Point(-80, 175, label=False), Point(100, 100, label=False)]
    c1 = Circle(points = points,
plot=True)


    print(f"Does the circle contain the red point {p1}, {p1 in c1}")

print(f"Does the circle contain the green point {p2}, {p2 in c1}")
    # c1 =
Circle(center=Point(350, 80), radius=60)
    print(c1)


    wn.update()

wn.exitonclick()

from NumberPlane import drawNumberPlane
from PointClass1 import Point
import
turtle

class Line:
    def __init__(self, slope=None, y_intercept=None, startPoint=None,
endPoint=None, plot=True, color="blue"):
        if slope != None and y_intercept !=
None:
            startPoint = Point(0, y_intercept, label=False, plot=False)
            x =
-y_intercept/slope
            endPoint = Point(x, 0, label=False, plot=False)
        elif
startPoint != None and endPoint != None:
            x1, y1 = startPoint.getXY()

x2, y2 = endPoint.getXY()
            slope = (y1- y2)/(x1-x2)
            y_intercept = y1 -
slope * x1

        self.slope, self.y_intercept, self.startPoint, self.endPoint = slope,
y_intercept, startPoint, endPoint
        if plot:
            startPoint.t.down()

originalColor = startPoint.t.color()[0]
            startPoint.t.color(color)

startPoint.t.pensize(3)
            startPoint.t.goto(endPoint.getXY())

startPoint.t.goto(startPoint.getXY())
            startPoint.t.up()

startPoint.t.pensize(1)
            startPoint.t.color(originalColor)

    def __str__(self):

      return f"""y={round(self.slope, 5)}*x + {round(self.y_intercept, 5)}

slope: {self.slope}
    y-intetcept: {self.y_intercept}
    startPoint: {self.startPoint}
```

```
endPoint: {self.endPoint}"""

    def findPointOfIntersection(self, other, label
= True, plot = True):

        """
        self:  y=self.slope*x + self.yint

    other: y=other.slope*x + other.yint
        x = (other.yint - self.yint)/((self.slope -
other.slope))
        """
        x = (other.y_intercept - self.y_intercept) /
((self.slope - other.slope))

        y = self.slope*x + self.y_intercept
        return
Point(x, y, label=label, plot=plot)

    def findPerpindicularLine(self, pointOfIntersection,
plot=True):
        oldSlope = self.slope
        newSlope = -1/oldSlope
        x, y =
pointOfIntersection.getXY()
        b = y - newSlope * x
        return Line(slope=newSlope,
y_intercept=b, color= "green", plot=plot)


if __name__ == "__main__":

wn = turtle.Screen()
    drawNumberPlane(50, wn)
    slope = 1.5
    line = Line(3, 10)


line1 = Line(startPoint=Point(320,120, label=False), endPoint=Point( -150,160, label=False))

 p = line1.findPointOfIntersection(line)
    print(p)
    # print(f"slope:
{line.slope}\n"
    #       f"y-intetcept: {line.y_intercept}\n"
    #
f"startPoint: {line.startPoint}\n"
    #       f"endPoint:
{line.endPoint}\n")
    wn.update()
    wn.exitonclick()

from CircleClass2 import
Circle
from PointClass2 import Point
from NumberPlane import drawNumberPlane
import
turtle
import random

def getBoundingBox(circles):
    horizontalBoundaries, verticalBoundaries
= [], []
    for aCircle in circles:
        leftBoundary = aCircle.getCenter().getX() -
aCircle.getRadius()
        rightBoundary = aCircle.getCenter().getX() + aCircle.getRadius()

     horizontalBoundaries.append(int(leftBoundary))

horizontalBoundaries.append(int(rightBoundary))

        bottomBoundary =
aCircle.getCenter().getY() - aCircle.getRadius()
```

```
        topBoundary =
aCircle.getCenter().getY() + aCircle.getRadius()


verticalBoundaries.append(int(bottomBoundary))

verticalBoundaries.append(int(topBoundary))

    horizontalBoundaries.sort()

verticalBoundaries.sort()

    print("horizontal: ", horizontalBoundaries)

print("veritcal: ", verticalBoundaries)

    return horizontalBoundaries[2],
horizontalBoundaries[3], verticalBoundaries[2], verticalBoundaries[3]


def
monteCarloSim(circles, numPoints):
    leftBoundary, rightBoundary, bottomBoundary, topBoundary
= getBoundingBox(circles)
    print("left", leftBoundary, "right",
rightBoundary, "bottom", bottomBoundary, "top", topBoundary)

numPointsInAllCirlces = 0
    for i in range(numPoints):
        randomX =
random.randint(leftBoundary, rightBoundary)
        randomY = random.randint(bottomBoundary,
topBoundary)
        randomPoint = Point(randomX, randomY, plot=True, label=False,
color="red")
        pIsInAllCircles = True
        for aCircle in circles:

  if not aCircle.isPointInCircle(randomPoint):
                pIsInAllCircles = False


if pIsInAllCircles:
            randomPoint.t.color("green")

numPointsInAllCirlces += 1

    areaOfBoundingBox = (rightBoundary- leftBoundary) *
(topBoundary - bottomBoundary)

    return
numPointsInAllCirlces/numPoints*areaOfBoundingBox

if __name__ == "__main__":
    wn
= turtle.Screen()
    wn.tracer(0)
    circles = []
    drawNumberPlane(50, wn)
    points =
[Point(120, 100, label=False), Point(-90, 300, label=False), Point(-330, 170, label=False)]

circles.append(Circle(points=points))
    points = [Point(-120, 150, label=False), Point(190,
100, label=False), Point(-130, 70, label=False)]
    circles.append(Circle(points=points))

points = [Point(20, 0, label=False), Point(-190, 200, label=False), Point(230, 400,
label=False)]
    circles.append(Circle(points=points))
    print(getBoundingBox(circles))


areaEstimate = monteCarloSim(circles, 1400)
    print(f"The intersection of the 3 circels
```

```
has an approximate area of: {areaEstimate} units^2")
    wn.update()

wn.exitonclick()







import turtle

def drawNumberPlane(gridSize, wn):  # Does not need to be
covered but it is fun
    t = turtle.Turtle()
    wn.tracer(0)
    t.up()

    w, h =
wn.screensize()
    for i in range(-h, h + 1, gridSize):
        t.goto(-w, i)

t.down()
        t.goto(w, i)
        t.up()

    for i in range(-w, w + 1, gridSize):

t.goto(i, -h)
        t.down()
        t.goto(i, h)
        t.up()

    t.pensize(4)

t.goto(0, -h)
    t.down()
    t.goto(0, h)
    t.up()
    t.goto(-w, 0)
    t.down()

t.goto(w, 0)
    t.up()
    wn.update()

    wn.turtles().remove(t)



if __name__ ==
"__main__":
    print("printed form number plane")
    wn =
turtle.Screen()
    drawNumberPlane(50, wn)
    wn.exitonclick()



import turtle
from
NumberPlane import drawNumberPlane

class Point:
    def __init__(self, x, y,
color="red", shapeSize=0.5, label=True):
        self.x = x
        self.y = y

 self.t = turtle.Turtle()
        self.t.up()
        self.t.shape("circle")
```

```
self.t.color(color)
        self.t.shapesize(shapeSize)
        if label:

self.t.goto(x + 5, y + 5)
            self.t.write(str(self), font=('Arial', 18, 'normal'))

    self.t.goto(x, y)

    def distanceBetween(self, other):
        a = self.getX() -
other.getX()
        b = self.getY() - other.getY()
        return (a ** 2 + b ** 2) ** 0.5


 def midPointBetween(self, other):
        x = (self.getX() + other.getX()) / 2
        y =
(self.getY() + other.getY()) / 2
        return Point(x, y)

    def __add__(self, other):

   x = self.getX() + other.getX()
        y = self.getY() + other.getY()
        return
Point(x, y)

    def __sub__(self, other):
        return self + other*-1

    def
__mul__(self, scalar):
        return Point(self.getX() * scalar, self.getY() * scalar)


def __rmul__(self, scalar):
        return self * scalar

    def __str__(self):
        return
f"({self.x}, {self.y})"

    def getX(self):
        return self.x

    def
getY(self):
        return self.y

    def getPointAsTuple(self):
        return self.x,
self.y

    def setX(self, x):
        self.x = x

    def setY(self, y):
        self.y =
y


if __name__ == "__main__":
    wn = turtle.Screen()

    drawNumberPlane(25,
wn)

    p1 = Point(60, 80, shapeSize=0.73)
    p2 = Point(10, -30, "blue")



    p4
```

```
= p1.__mul__(3)
    print(p4)
    p4 = 3 * p1

    print(p4)

    wn.update()

wn.exitonclick()



import turtle
import random
import math

def computeAngle(loc):
    x, y =
loc
    if x == 0: x += 0.001
    if y == 0: y += 0.001
    radius = (x ** 2 + y ** 2) ** 0.5

  add180 = 0
    if bool(x > 0) != bool(y > 0):
        add180 = 180
    return add180 +
math.degrees(math.asin(y / radius)), radius

def random_offset2(loc, color):
    r, g, b =
color
    a, radius = computeAngle(loc)
    if a >= -90 and a < 30:
        r =
(random.betavariate(radius/8, (radius/8)**0.5))
    elif a >= 30 and a < 150:
        g =
(random.betavariate(radius/8, (radius/8)**0.5))
    elif a > 150 and a < 270:
        b =
(random.betavariate(radius/8, (radius/8)**0.5))

    if r < 0: r = 0
    if b < 0: b = 0

   if g < 0: g = 0
    if r > 1: r = 1
    if b > 1: b = 1
    if g > 1: g = 1

return r, g, b

def random_offset3(loc, color):
    r, g, b = color
    a, radius =
computeAngle(loc)
    randomColorNum = random.randint(0,3)
    if randomColorNum == 0:

r += (random.betavariate(1, 1) - 0.5)*(radius/800)
    elif randomColorNum == 1:
        g +=
(random.betavariate(1, 1) - 0.5)*(radius/800)
    else:
        b += (random.betavariate(1, 1)
- 0.5)*(radius/800)

    if r < 0: r = 0
    if b < 0: b = 0
    if g < 0: g = 0

if r > 1: r = 1
    if b > 1: b = 1
    if g > 1: g = 1
```

```
    return r, g, b

def
random_offset4(loc, color):
    r, g, b = color
    a, radius = computeAngle(loc)

randomColorNum = random.randint(0,3)
    if randomColorNum == 0:
        r +=
(random.betavariate(1, 1) - 0.5)*(800/radius)
    elif randomColorNum == 1:
        g +=
(random.betavariate(1, 1) - 0.5)*(800/radius)
    else:
        b += (random.betavariate(1, 1)
- 0.5)*(800/radius)

    if r < 0: r = 0
    if b < 0: b = 0
    if g < 0: g = 0

if r > 1: r = 1
    if b > 1: b = 1
    if g > 1: g = 1
    return r, g, b

def
generateNewColorComponet(value):
    value += (random.betavariate(2, 2) - 0.5) / 10
    if
value < 0:
        return 0
    elif value > 1:
        return 1
    return value

def
random_offset(color):
    r,g,b = color

    r = generateNewColorComponet(r)
    g =
generateNewColorComponet(g)
    b = generateNewColorComponet(b)
    return r,g,b


def
draw_line(x1, y2):

    r,g,b = random_offset(t.color()[0])
    t.color((r,g,b))
    t.goto(x1,
0)
    t.down()
    t.goto(0, y2)
    t.up()
    t.goto(0,0)


"""To run this
program, click and drag the circle in the middle
 of the canvas."""
t =
turtle.Turtle()
t.up()
t.shape("circle")
t.color((random.random(), random.random(),
random.random()))
wn =
turtle.Screen()
wn.tracer(0)
wn.listen()
t.ondrag(draw_line)
```

```
wn.mainloop()


import
turtle
import random
# https://en.wikipedia.org/wiki/Marching_squares

def createMatrix(rows,
cols):
    """
    This function creates a 2D list where each inner list

contains randomly generated 1s and 0s.
    :param rows: How many inner lists are in the 2D
list
    :param cols: How many values are in each inner list
    :return: a 2D list
    rows,
cols = 3, 4
    [
        [0,1,1,0],
        [1,0,0,0],
        []
    ]

"""
    cells = []
    for row in range(rows+1):
        innerList = []

for col in range(cols+1):
            innerList.append(random.randint(0, 1))

cells.append(innerList)

    return cells


def markCorners(cells, t):
    """

  This function plots each corner of a 2D grid as either
    red or pink for each value in the
2D list cells. It can
    be called or not. It just illustrates the grid created by
    the 2D
list cells.
    :param cells: a 2D list containging random 1s and 0s
    :param t: a turtle

:return: None
    """
    for rowIndex in range(len(cells)):
        for
colIndex in range(len(cells[0])):
            if cells[rowIndex][colIndex] == 1:

 t.color("red")
            else:
                t.color("pink")

  t.goto(colIndex, rowIndex)
            t.stamp()

def midpoint(p1, p2):

"""
    This function computes the midpoint between p1 and p2.
    :param p1: a
tuple representing a point
    :param p2: a tuple representing a point
    :return: a tuple
representing a point
    p1 = (10, 2)
```

```
    """
    x1, y1 = p1
    x2, y2 = p2

 return (x1 + x2) / 2, (y1 + y2) / 2



def drawLineSegment(t, p1,p2,p3,p4):

"""
    This function draws a line starting at the midpoint of
    p1 and p2 and
ending at the midpoint of p3 and p4.
    :param t: a turtle
    :param p1: a tuple representing
a point
    :param p2: a tuple representing a point
    :param p3: a tuple representing a
point
    :param p4: a tuple representing a point
    :return: None
    """


midpoint1 = midpoint(p1, p2)
    midpoint2 = midpoint(p3, p4)
    t.up()
    t.goto(midpoint1)

   t.down()
    t.goto(midpoint2)
    t.up()







def marchingSqs(cells, t):

"""
    https://en.wikipedia.org/wiki/Marching_squares
    This function
implements the marching squares algorithm.
    This algorithm has 16 different cases depending
on which
    corners of a square are 1s and which ones are 0s. After
    determining which case
a square in the grid falls into, it
    draws a line segment corresponding to that case.

:param cells: a 2D list
    :param t: a turtle
    :return: None
    """
    for
y in range(len(cells)-1):
        for x in range(len(cells[0])-1):
            pass

upperLeft = (x, y+1)
            upperRight = (x+1, y+1)
            lowerRight = (x+1, y)

       lowerLeft = (x, y)
            upperLeftValue = cells[y+1][x]

upperRightValue = cells[y+1][x+1]
            lowerRightValue = cells[y][x+1]

lowerLeftValue = cells[y][x]
            case = str(upperLeftValue) + str(upperRightValue) +
str(lowerRightValue) + str(lowerLeftValue)
            print(case, (x,y))
```

```
            if case
== "0000": continue
            elif case == "0001": drawLineSegment(t,
lowerLeft, lowerRight, upperLeft, lowerLeft)
            elif case == "0010":
drawLineSegment(t, lowerLeft, lowerRight, lowerRight, upperRight)
            elif case ==
"0011": drawLineSegment(t, lowerLeft, upperLeft, lowerRight, upperRight)

elif case == "0100": drawLineSegment(t, upperLeft, upperRight, lowerRight,
upperRight)
            elif case == "0101":
                drawLineSegment(t,
lowerLeft, upperLeft, upperLeft, upperRight)
                drawLineSegment(t, lowerLeft,
lowerRight, lowerRight, upperRight)
            elif case == "0110":

drawLineSegment(t, lowerLeft, lowerRight, upperLeft, upperRight)
            elif case ==
"0111":
                drawLineSegment(t, lowerLeft, upperLeft, upperLeft,
upperRight)
            elif case == "1000":
                drawLineSegment(t,
lowerLeft, upperLeft, upperLeft, upperRight)
            elif case == "1001":

        drawLineSegment(t, lowerLeft, lowerRight, upperLeft, upperRight)
            elif case
== "1010":
                drawLineSegment(t, lowerLeft, lowerRight, lowerLeft,
upperLeft)
                drawLineSegment(t, upperLeft, upperRight, lowerRight, upperRight)

         elif case == "1011":
                drawLineSegment(t, upperLeft,
upperRight, lowerRight, upperRight)
            elif case == "1100":

drawLineSegment(t, lowerLeft, upperLeft, lowerRight, upperRight)
            elif case ==
"1101":
                drawLineSegment(t, lowerLeft, lowerRight, lowerRight,
upperRight)
            elif case == "1110":
                drawLineSegment(t,
lowerLeft, lowerRight, lowerLeft, upperLeft)
            elif case == "1111":

        continue
scale = 20
rows, cols = 1*scale, 1*scale
wn =
turtle.Screen()
wn.setworldcoordinates(0, 0, cols, rows)
t =
turtle.Turtle()
wn.tracer(0)
t.up()
t.hideturtle()
t.shape("circle")
t.shapesize(0.5)


cells = createMatrix(rows, cols)
for i, row in enumerate(cells):
    print(i,
row)
print(cells)
markCorners(cells, t)
t.color("black",
"red")
marchingSqs(cells, t)
```

```
wn.update()
wn.exitonclick()


import math
import
turtle
# http://hyperphysics.phy-astr.gsu.edu/hbase/traj.html

gravity = 9.8
def
trajectory(theta, velocity, elapsedTime):
    '''
    This function computes the location of
the projectile after
    being launched for an elapsedTime amount of time with an
    angle of
theta and a force of velocity.

    :param theta: the angle at which the projectile is
launched
    :param velocity: the force applied to the projectile
    :param elapsedTime: the
current amount of time the projectile has
                 been in the air
    :return: the x,
y coordinate of where the projectile is at
             the current time
    '''
    vSub0X =
velocity * math.sin(theta)
    vSub0Y = velocity * math.cos(theta)
    x = vSub0X*elapsedTime

  y = vSub0Y * elapsedTime - 0.5*gravity*elapsedTime**2
    return x, y

for i in range(1,
10):
    pt = trajectory(math.radians(45), 80, i/10)

print(f"{pt[0]:0.2f}\t{pt[1]:0.2f}")

def flightTime(theta, velocity):

"""
    This function computes the total time that projectile is
    in the
air.
    :param theta: the angle at which the projectile is launched
    :param velocity: the
force applied to the projectile
    :return: the total amount of time the projectile is in the
air
    """
    vSub0Y = velocity * math.cos(theta)
    return
2*vSub0Y/gravity

def distanceFunc(theta, velocity):
    """
    This function
computes the total distance traveled by
    the projectile.
    :param theta: the angle at
which the projectile is launched
    :param velocity: the force applied to the projectile

:return: the total distance traveled by the projectile
    """
    timeInAir =
flightTime(theta, velocity)
    x,y = trajectory(theta, velocity, timeInAir)
    return x
```

```
def
launchProjectile(rocket, theta, velocity):
    """
    This function draws the
arc of the projectile launched with
    an angle of theta and a force of velocity. It uses a
for
    loop to compute each location of the projectile.
    :param rocket: the turtle
representing the projectile
    :param theta: the angle at which the projectile is launched

:param velocity: the force applied to the projectile
    :return: None
    """

  offsetX, offsetY= -300, -300
    dist = distanceFunc(theta, velocity)#826.5 -> 827

timeInAir = flightTime(theta, velocity)#12.9876 -> 13
    rocket.goto(offsetX, offsetY)

rocket.down()
    for interval in range(1, round(dist), 10):
        partialTime =
interval*timeInAir/dist
        x, y = trajectory(theta, velocity, partialTime)

rocket.goto(x+offsetX,y+offsetY)
    x, y = trajectory(theta, velocity, partialTime)


rocket.goto(x + offsetX, y + offsetY)


def main():
    """
    This function
gathers user inputs and calls the
    launch projectile function
    :return:None

"""
    rocket = turtle.Turtle()
    rocket.color("purple")


rocket.up()
    rocket.goto(-300, -300)
    rocket.down()
    rocket.goto(500, -300)

rocket.up()
    rocket.speed(10)

    angle = input("Angle: ")
    power =
input("Power: ")
    theta = math.radians(int(angle))
    velocity = int(power)

launchProjectile(rocket, theta,
velocity)


main()
main()
main()
turtle.exitonclick()

"""
upperLeadingEdgeRadiu
s (Rleu)
lowerLeadingEdgeRadius (Rlel)
```

```
PositionOfUpperCrestPoint (Xup)
upperCrestPoint
(Yup)
upperCrestCurvature  (YXXup)
PositionOfLowerCrestPoint (Xlo)
lowerCrestPoint
(Ylo)
lowerCrestCurvature (YXXlo)
trailingEdgeThickness (TTE)
trailingEdgeOffset
(Toff)
trailingEdgeDirectionAngle (alphaTE)
trailingEdgeWedgeAngle (betaTE)
ai and bi are the
coefficants representing the 12 control variables
Unknown coefficants: ai, bi for i in
range(1,6)
yu (upperCurveEQ) -> summation from 1...6 (ai*X^(i-0.5))
yl (lowerCurveEQ) ->
summation from 1...6 (bi*X^(i-0.5))
1.  x(u, l) = maximum
    y(u, l) = maximum
2.  x(u, l) =
maximum
    (dy(u, l)/dx^2) = 0
3.  x(u, l) = maximum
    (d^2y(u, l)/dx^2) = 0
4.  x_u = 1

y_u = T_off + T_TE
5.  x_l = 1
    y_l = T_off - T_TE/2
6.  X_u = 1
    dy_u/dx = tan(alphaTE -
betaTE/2)
7.  X_l = 1
    dy_l/dx = tan(alphaTE - betaTE/2)
"""
#
upperLeadingEdgeRadius = 0.0216
# lowerLeadingEdgeRadius = 0.008
# PositionOfUpperCrestPoint =
0.3445
# upperCrestPoint = 0.07912
# upperCrestCurvature = -0.6448
#
#
PositionOfLowerCrestPoint = 0.17
# lowerCrestPoint = -0.033797
# lowerCrestCurvature = 0.6748
#
trailingEdgeThickness = 0
# trailingEdgeOffset = 0
#
# trailingEdgeDirectionAngle = -4.785
#
trailingEdgeWedgeAngle = 15.082
import turtle
import math

wn = turtle.Screen()

def foilEq(t,
x):
    x = x /100
    temp = 0.2969*x**0.5
    temp -= 0.126*x
    temp -= 0.3516*x**2

temp += 0.2843*x**3
    temp -= 0.1036*x**4
    return 5*t*temp

def
saveAirFoilPoints(airFoilName):
```

```
    points = generateFoilPoints(airFoilName)
    f =
open(str(airFoilName)+".csv", "w")
    for aPoint in points:

f.write(str(round(aPoint[0], 5)) + ", " + str(round(aPoint[1], 5)) + "\n")

   f.close()


def camberedFoilEq(m, p, t, x, c =100):
    m /= 100
    p /= 10
    if x >=
0 and x <= p*c:
        camberPointY = (m/p**2)*(2*p*(x/c)-(x/c)**2)*c
        dydx =
(2*m/p**2)*(p-x/c)
    else:
        camberPointY = (m/(1-p)**2)*((1-2*p) + 2*p*(x/c) -
(x/c)**2)*c
        dydx = (2 * m / (1-p) ** 2) * (p - x / c)

    theta = math.asin(dydx)

yt = foilEq(t, x)
    x_u = x - yt * math.sin(theta)
    x_l = x + yt * math.sin(theta)
    y_u
= camberPointY + yt * math.cos(theta)
    y_l = camberPointY - yt * math.cos(theta)
    return
x_u, y_u, x_l, y_l, x, camberPointY

def parseAirFoilName(airFoilName):
    m = airFoilName //
1000
    airFoilName -= m * 1000
    p = airFoilName // 100
    t = airFoilName - p * 100

return m, p, t

def printPoints(points):
    for point in points:
        print(point)

def
generateFoilPoints(airFoilName):
    m,p,t = parseAirFoilName(airFoilName)
    points = []


for x in range(0, 101):
        x1, y1, x2, y2, x3, y3 = camberedFoilEq(m, p, t, x, 100)

 points.insert(0, (x1, y1))
        points.insert(-1, (x2, y2))
    return points

def
drawFoil(loc, airFoilName, scale=1, color = "black"):
    xOffset, yOffset = loc

top = turtle.Turtle()
    top.color(color)
    bottom = turtle.Turtle()

bottom.color(color)
    top.speed(0)
    top.up()
    bottom.up()
    top.hideturtle()

bottom.hideturtle()
```

```
    points = generateFoilPoints(airFoilName)
    top.goto(points[0])

bottom.goto(points[0])
    top.down()
    bottom.down()
    print(points[0])
    s =
""
    for x, y in points[1:]:
        top.goto(x*scale + xOffset, y*scale +
yOffset)
        s += f"{x*scale + xOffset}\t{y*scale + yOffset}\n"

print(s)

drawFoil((-180, -180), 6419, 3, "black")
drawFoil((-180, -180), 2419, 3,
"red")
```