Introduction to programming

Labor07

Revision

- Subroutines
  - Procedures
  - Functions
- Variables
  - Local
  - Global
- Parameter passing
- Optional parameters
- `main()` function

Main function

- Entry point to the programme:

```
def main():
    pass

if __name__ == "__main__":
    main()
```

Recursive functions

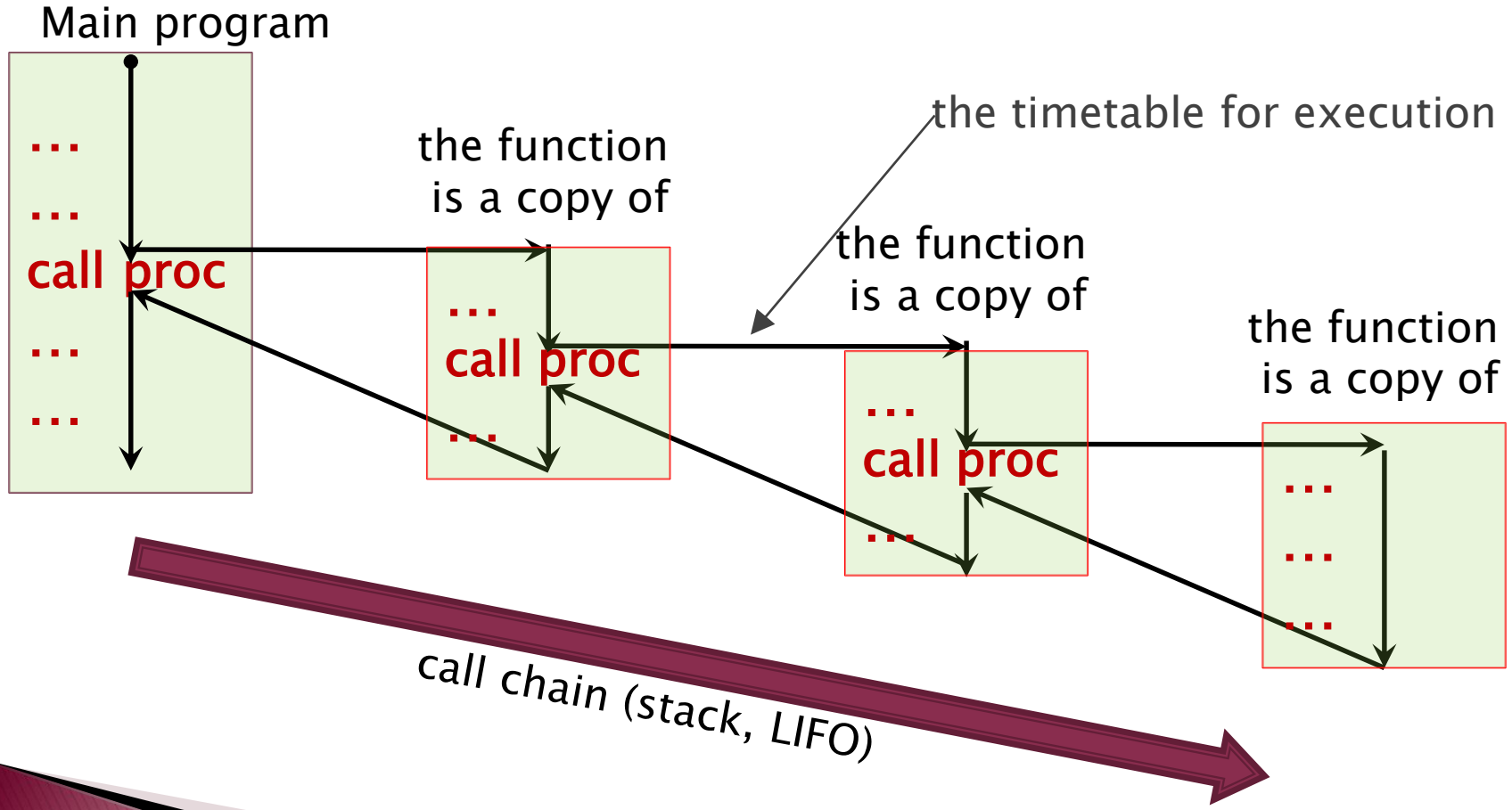
- The function **calls itself.**
- The function has at least one reference to itself in its body.
- A branch instead of a cycle:
  - True branch – stop condition
  - False branch – recursive call
- Recursive algorithms are usually used when the basic task is too complicated, but with recursive calls we can reduce this to simpler (sub)tasks
- Accordingly, a recursive task is usually defined in a similar way:
  - trivial solution
  - simplification of a general case

Recursive functions

- The advantages of recursion
  - Recursive functions help to keep your code clean and elegant.
  - A complex problem can be broken down into simpler sub-problems by recursion.
  - Sequence generation is easier with recursion than using some nested iterations.
- The disadvantages of recursion
  - Sometimes the logic behind recursion is difficult to follow.
  - Recursive calls are expensive (inefficient) because they require a lot of memory and time.
  - Recursive functions are difficult to debug.

Recursion

- A subprogram calls itself

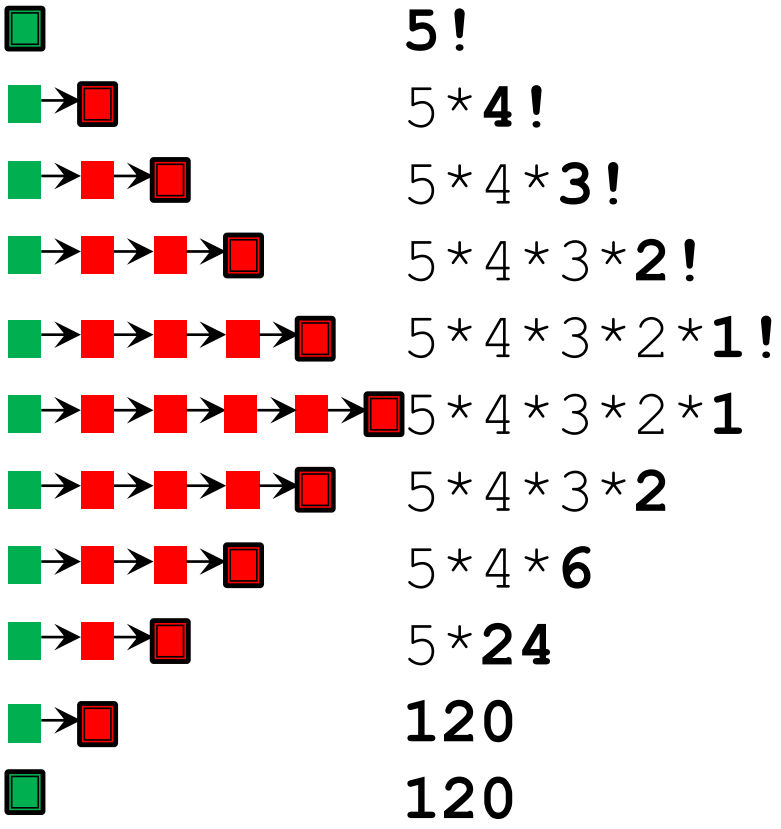


Exercise

- Write a recursive function that determines the factorial of N.

Recursion

- Factorial:
  - If the (N) is 1, than the factorial is 1.
  - Else N! = N * (N-1)!



Solution1

```
def n_factorial(n: int) -> int:
    if n == 1:
        return 1
    else:
        return n * n_factorial(n - 1)


def main():
    print(n_factorial(5))

if __name__ == "__main__":
    main()
```

Solution2

```
def n_factorial(n: int) -> int:
    return 1 if n == 1 else n_factorial(n - 1) * n


def main():
    print(n_factorial(5))

if __name__ == "__main__":
    main()
```

```python
import sys

def n_factorial(n: int) -> int:
    if n == 1:
        return 1
    else:
        return n * n_factorial(n - 1)

def n_factorial1(n: int) -> int:
    return 1 if n == 1 else n *
           n_factorial1(n - 1)

def read() -> None:
# read until EOF
    while True:
        try:
            n = int(input())
            print(n_factorial(n))
        except EOFError:
            break
```

```python
def read1() -> None:
# read until EOF
    for line in sys.stdin:
        n = int(line)
        print(n_factorial(n))

def read2() -> None:
# N test cases
    n=int(input())
    for i in range(n):
        num=int(input())
        print(n_factorial(num))

def read3() -> None:
# numbers between [0,50]
    while True:
        n=int(input())
        if n<0 or n>50:
            break
        print(n_factorial(n))

def main():
    read3()

if __name__ == '__main__':
    main()
```

Exercise

- Write a recursive power function defining X^n.

Solution

```
def power(x: int, n: int) -> int:
    if n == 1:
        return x
    else:
        return x * power(x, n - 1)

def main():
    x = int(input("x = "))
    n = int(input("n = "))
    print(power(x, n))

if __name__ == "__main__":
    main()
```

Solution

```
def power(x: int, n: int) -> int:
    return x if n == 1 else x * power(x, n - 1)

def main():
    x = int(input("x = "))
    n = int(input("n = "))
    print(power(x, n))

if __name__ == "__main__":
    main()
```

```python
import math
def power_it(x: int, n: int) -> int:
    p = 1
    for i in range(n):
        p = p * x
    return p

def power_rec1(x: int, n: int) -> int:
    if n == 1:
        return x
    else:
        return x * power_rec1(x,n-1)

def power_rec2(x: int, n: int) -> int:
    return x if n == 1 else x * power_rec2(x,n-1)

def main():
    x=int(input("x="))
    n=int(input("n="))
    print(power_it(x,n))
    print(power_rec1(x, n))
    print(power_rec2(x, n))
    print(int(math.pow(x,n)))
if __name__ == '__main__':
    main()
```

Recursion

- Write a recursive function that determines the sum of the numbers from 1 to N.

$$\sum_{i=1}^{n} i$$

Solution

```
def n_sum(n: int) -> int:
    if n == 1:
        return 1
    else:
        return n + sum(n - 1)

def main():
      print(n_sum(n))

if __name__ == "__main__":
    main()
```

Solution

```
def n_sum1(n: int) -> int:
    return 1 if n == 1 else n + sum1(n - 1)

def main():
    print(n_sum1(n))

if __name__ == "__main__":
    main()
```

Recursion

- Write a recursive function that determines the sum of the squares of the numbers from 1 to N.

$$\sum_{i=1}^{n} i^2$$

Solution

```
def sum2(n: int) -> int:
    if n == 1:
        return 1
    else:
        return n * n + sum2(n - 1)

def sum2(n: int) -> int:
    return 1 if n == 1 else n * n + sum2(n - 1)
```

Exercise

- Write a recursive function that determines the following sum:

$$\sum_{i=1}^{n} (i^2 - 2)$$

Solution

```
def sum3(n: int) -> int:
    if n == 1:
        return -1
    else:
        return n * n - 2 + sum3(n - 1)

def sum3(n: int) -> int:
    return -1 if n == 1 else n * n - 2 + sum3(n - 1)
```

Exercise

- Write a recursive function that determines the following sum:

$$\sum_{i=3}^{n} (i^2 - 2)$$

Solution

```
def sum4(n: int) -> int:
    if n == 3:
        return 7
    else:
        return n * n - 2 + sum4(n - 1)
```

```
def sum4(n: int) -> int:
    return 7 if n == 3 else n * n - 2 + sum4(n-1)
```

Exercise

- Write a recursive function that determines the following sum:

$$\sum_{i=4}^{n} i(i^2 - 1)$$

Solution

```
def sum5(n: int) -> int:
    if n == 4:
        return 60
    else:
        return n * (n * n - 1) + sum5(n - 1)
```

```
def sum5(n: int) -> int:
   return 60 if n == 4 else n * (n * n - 1) + sum5(n-1)
```

Exercise

- Write a recursive function that determines the following sum:

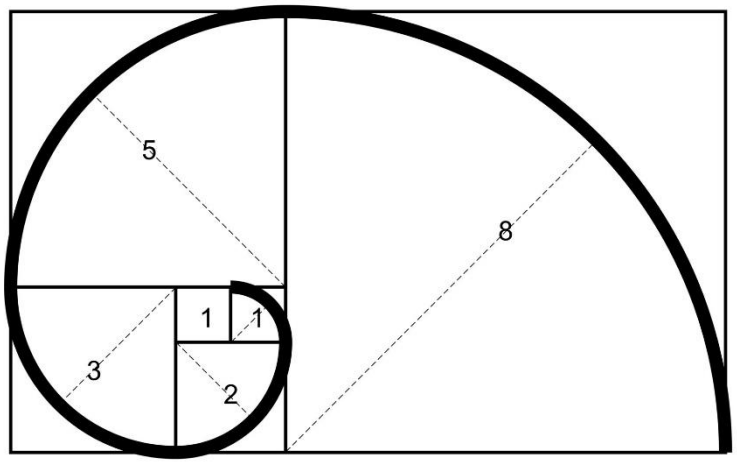
$$\sum_{i=2}^{n} (i^3 - 5)$$

Solution

```
def sum6(n: int) -> int:
    if n == 2:
        return 3
    else:
        return n * n * n - 5 + sum6(n - 1)
```

```
def sum6(n: int) -> int:
  return 3 if n == 2 else n * n * n - 5 + sum6(n-1)
```

Exercise



- Write a recursive function to determine the n^{th} Fibbonacci number.

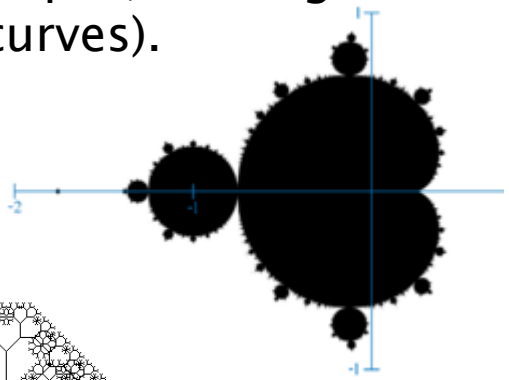
$$f(n) = \begin{cases} 0 & if \quad n = 0 \\ 1 & if \quad n = 1 \\ F(n-1) + F(n-2) & if \quad n > 1 \end{cases}$$

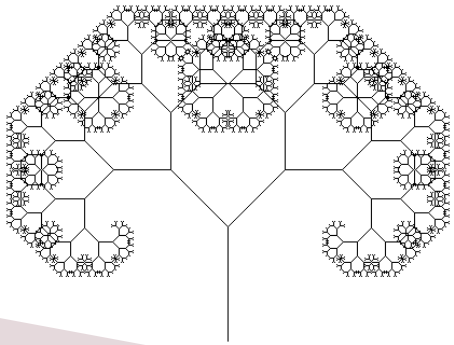
Solution

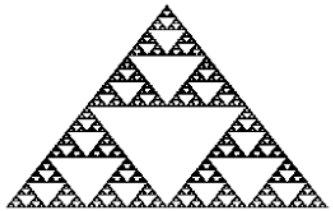
```
def n_fibonacci (n: int) -> int:
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return n_fibonacci (n - 1) + n_fibonacci (n - 2)
```

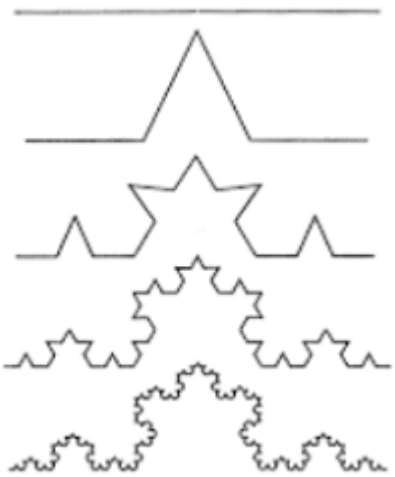
Applications of recursion

- Fractals
- Fractals are "self-similar", infinitely complex mathematical shapes, whose varied forms have at least one recurrence that is recognisable (i.e. describable by mathematical means).
- The name was given in 1975 by Benoît Mandelbrot, from the Latin word fractus (meaning broken; fracture), which refers to the fractional number of dimensions of such shapes, although not all fractals are fractional (such as plane-filling curves).
- Koch curve
- Sierpinski triangle
- Mandelbrot set
- Pythagorean tree
- Newton fractal
- Julia set

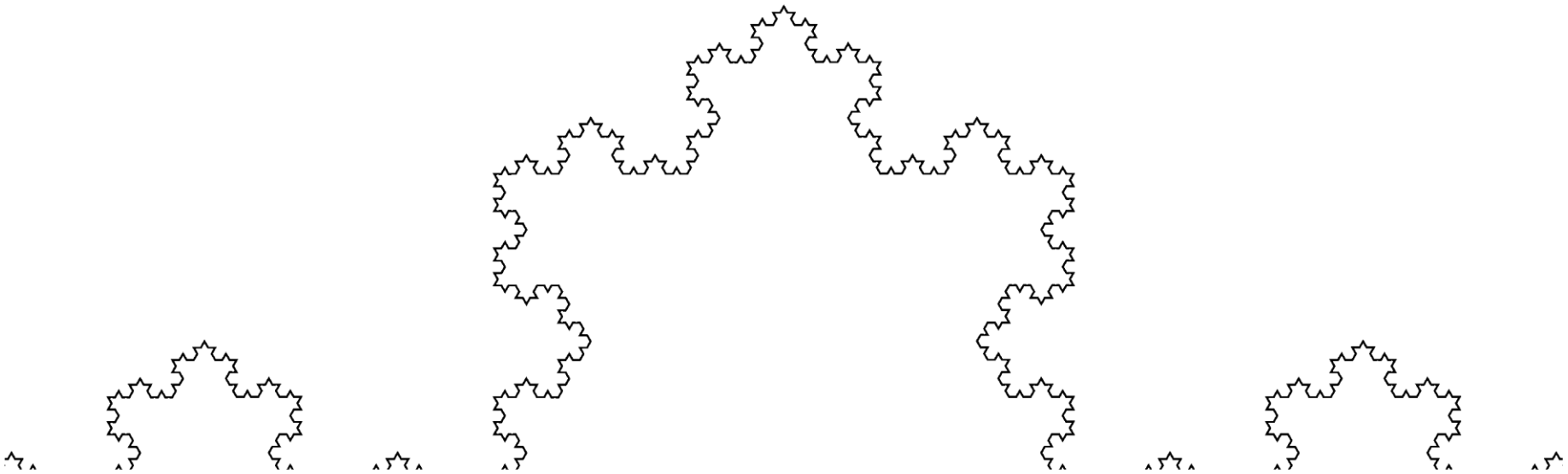






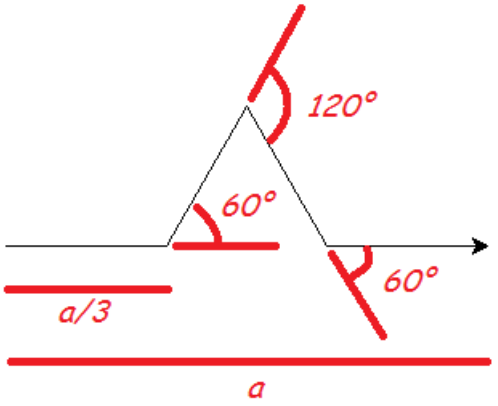


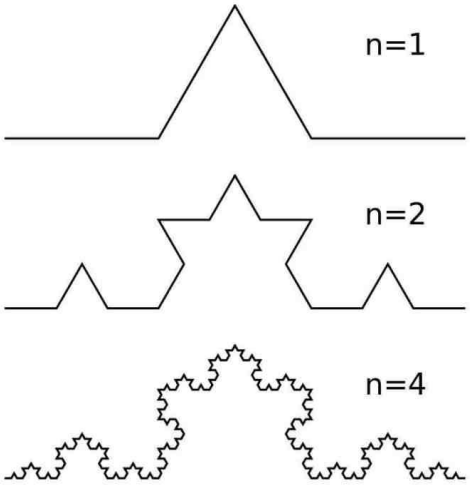
Koch curve

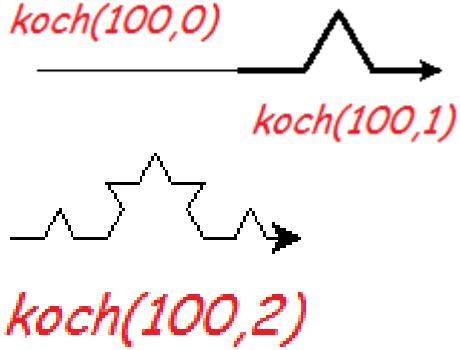


Koch curve

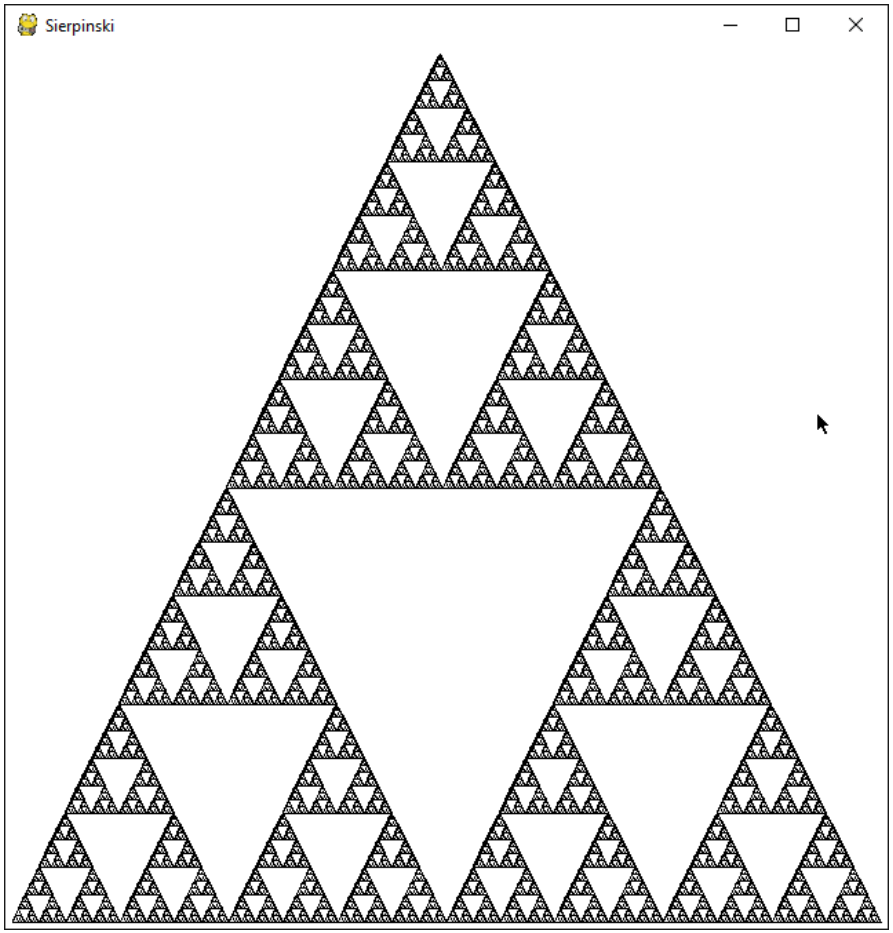
```
from turtle import *

def koch(a, order):
    if order > 0:
        for t in [60, -120, 60, 0]:
            forward(a/3)
            left(t)
    else:
        forward(a)

def koch2(a, order):
    if order > 0:
        for t in [60, -120, 60, 0]:
            koch(a/3,order-1)
            left(t)

koch(100, 0)
pensize(3)
#koch(100,1)
koch2(100, 2)
```



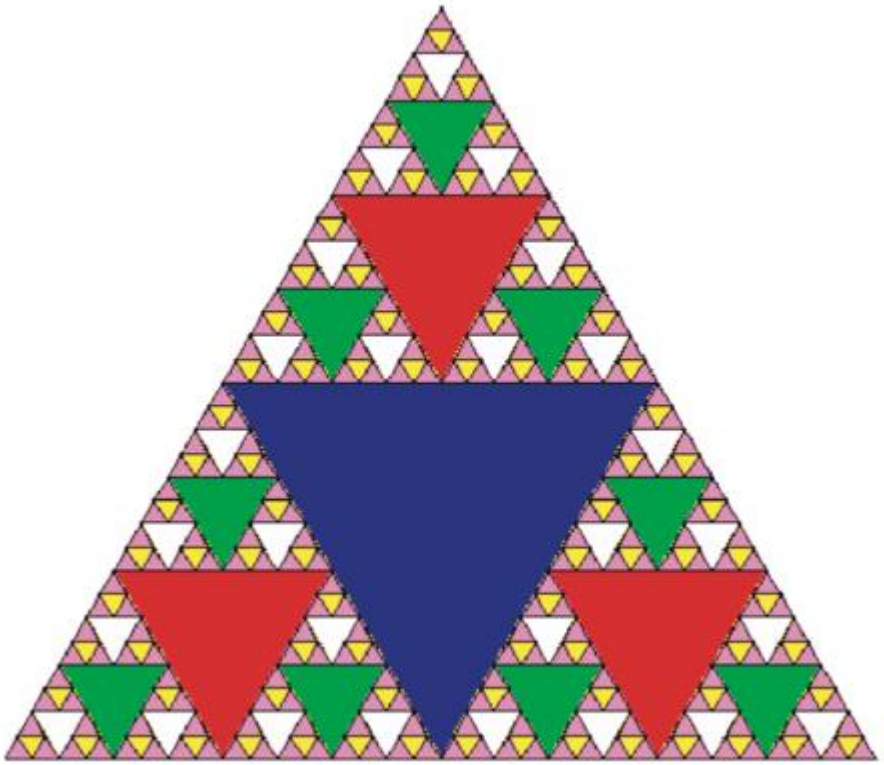




Sierpinski triangle



Sierpinski triangle



```
import turtle

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

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

def sierpinski(points,degree,myTurtle):
    colormap = ['blue','red','green','white','yellow',
                'violet','orange']
    drawTriangle(points,colormap[degree],myTurtle)
    if degree > 0:
        sierpinski([points[0],
                    getMid(points[0], points[1]),
                    getMid(points[0], points[2])],
                   degree-1, myTurtle)
        sierpinski([points[1],
                    getMid(points[0], points[1]),
                    getMid(points[1], points[2])],
                   degree-1, myTurtle)
        sierpinski([points[2],
                    getMid(points[2], points[1]),
                    getMid(points[0], points[2])],
                   degree-1, myTurtle)


myTurtle = turtle.Turtle()
myWin = turtle.Screen()
myPoints = [[-100,-50],[0,100],[100,-50]]
sierpinski(myPoints,3,myTurtle)
myWin.exitonclick()
```

Fractals in nature

Read lists – Until EOF

```
import sys

for line in sys.stdin:
    numbers = line.split(" ")
    lists=[]
    for number in numbers:
        lists.append(int(number))
    print(lists)
```

Read lists – Until EOF

```
import sys

for line in sys.stdin:
    lists=[int(number) for number in line.split(" ")]
    print(lists)
```

Read lists – N test cases

```
n = int(input())
for i in range(n):
    numbers = input().split(" ")
    lists=[]
    for number in numbers:
        lists.append(int(number))
    print(lists)
```

Read lists – N test cases

```
n = int(input())
for i in range(n):
    line = input()
    lists=[int(number) for number in line.split(" ")]
    print(lists)
```

Read lists – Until an empty line

```
import sys
for line in sys.stdin:
    if line.strip() == "":
        break
    numbers = line.split(" ")
    lists=[]
    for number in numbers:
        lists.append(int(number))
    print(lists)
```

Read lists – Until an empty line

```
import sys

for line in sys.stdin:
    if line.strip() == "":
        break
    numbers = line.split(" ")
    lists = [int(number) for number in numbers]
    print(lists)
```

Homework 1

https://viskillz.inf.unideb.hu/prog/#/2022?week=P1041

- Write a function named *count_of_squares()* that returns the number of squares of the elements in the list given as a parameter.

- In case of the *main()* function, read lists elements until the given condition, and call the *count_of_squares()* function and print out the number of squares elements.
  - Reads lists until EOF
  - Reads n test cases (lists)
  - Reads until an empty line

Homework 2

https://viskillz.inf.unideb.hu/prog/#/2022?week=P1051

- Write a function named ***delete_even_digits()*** that returns the string returned by deleting the even digits of the string given as a parameter.

- In case of the *main()* function, read strings until the given condition, and call the ***delete_even_digits()*** function and print out the new string without even digits.
  - Reads strings until EOF
  - Reads n test cases
  - Reads until an empty string

Homework3

https://viskillz.inf.unideb.hu/prog/#/?week=P1032

- Write a *Pythagorean()* function that returns the hypotenuse of a triangle given the two side (a and b).

- In case of the *main()* function, read datas until the given condition (a b ->in one line, seperated by the space character), and call the ***Pythagorean()*** function and print out the hypotenuse value.
  - Reads datas until EOF
  - Reads n test cases
  - Reads until an empty line