멀티쓰레드,
Turtle 드로잉

Seolyoung Jeong, Ph.D.

경북대학교 IT대학 컴퓨터학부

프로세스 & 쓰레드

Thinking…

- input() 함수 수행 시, 사용자의 입력을 기다리는 동안 아무 것도 못함
- 우리는 PC 사용 시 여러 개의 프로그램을 동시에 동작시킴
  예) 음악을 들으면서 + 워드 작업을 함
- PC의 운영체제에서는 멀티태스킹(multi-tasking) 기능을 제공함으로써 동시 수행이 가능하도록 함
- 태스크(task) : 일의 수행 단위 = 프로세스(process)라고도 함

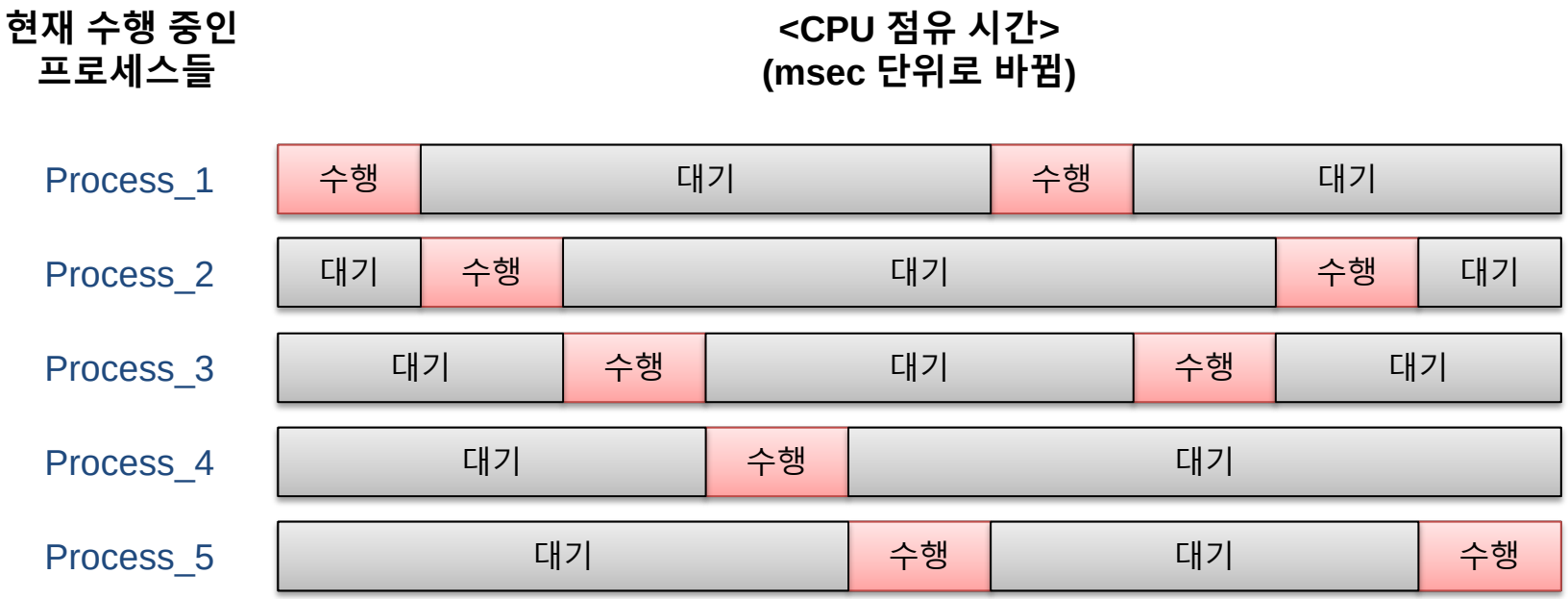
프로세스

- **프로세스 또는 태스크 : 시스템에서 일을 수행하는 단위**
- **GPOS(General Purpose OS, 예:윈도우)에서는 프로세스라는 용어를 많이 사용**
- **RTOS(Real-Time OS, 예:FreeRTOS, VxWorks 등)에서는 태스크라는 용어를 많이 사용**

- **실시간 운영체제(RTOS)**
  - 주로 임베디드 시스템에서 많이 사용됨
  - 정해진 시간(deadline time) 안에 수행이 완료되도록 보장해줌
    → 예측 가능함 (deterministic)
  - 우선순위에 따른 수행을 보장 (priority-based)
    → 우선순위가 높은 태스크를 먼저 수행
  - Hard Real-Time OS : 국방, 항공, 우주선, 원자력 등 처리 시간의 변동폭(jitter)이 아주 작아야 함
  - Soft Real-Time OS : 멀티미디어 시스템, 가전제품 등 처리 시간의 변동폭(jitter)이 좀 넓더라도 일을 처리하는데 문제가 발생하지 않음

CPU 프로세스 처리

- CPU는 한 번에 한 개의 프로세스만 처리 가능
- 여러 개의 프로세스를 짧은 시간으로 쪼개어 바꿔가며 수행 (Scheduling)
- 사용자 눈에는 동시에 동작하는 것으로 착각



멀티 프로세싱

- 멀티 + 프로세서 : 여러 개의 프로세서(CPU)가 일을 수행
  → 멀티 코어 CPU 기반

- CPU의 성능은 기하급수적으로 증가해 왔음
  (폴락의 법칙 : 성능은 면적(트랜지스터 수) 증가량의 제곱근과 비례)
- 하지만, CPU 하나만으로 처리할 수 있는 작업 속도에는 한계가 있음
  → AMD에서 최초로 데스크탑용 듀얼코어 CPU를 내놓음 (2005년)

- 멀티코어 : 두 개 이상의 독립적인 코어를 단일 직접 회로로 하나의 패키지로 통합
- 듀얼 코어부터 시작하여 인텔에서 도헵타콘타 코어(72 코어)까지 발표
  ➔ 일반 프로그래밍 방식으로는 하나의 코어만 주로 사용하게 됨
  병렬 프로세싱에 대한 이해와 할당 정책이 필요함
  (이를 위한 병행성 프로그래밍 언어도 존재. 예: Erlang)

쓰레드

- **하나의 프로세스에는 여러 개의 쓰레드가 존재 가능**
- **쓰레드 : 실제 최소 실행 단위 (별도의 실행 메모리 공간을 갖는다)**

- **왜 필요한가?**
- **예1) 워드 or 한글**
  - 사용자 입력 (버튼 클릭 or 타이핑)
  - 자동 오타 검사
  - 일정시간마다 자동저장

- **예2) 웹브라우저**
  - 동영상 재생
  - 서버로부터 스트리밍 데이터 수신
  - 사용자 입력 (댓글 달기)

- **예3) 웹서버 프로그램**
  - 여러 명의 클라이언트가 동시에 접속하여 개별적으로 필요한 기능을 수행

파이썬 쓰레드

non-Thread

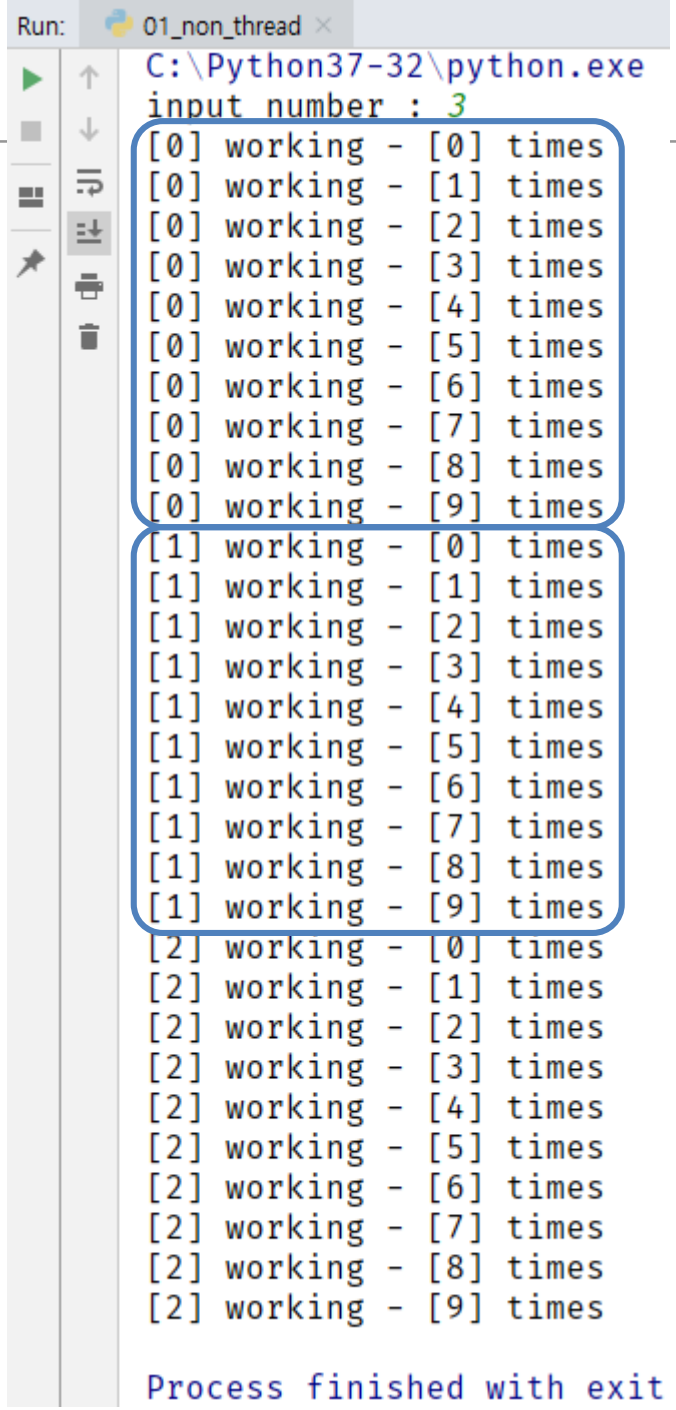
- worker가 몇 명인지 입력 받음
- 한 명의 worker마다 10번의 working 수행

```
1   # function
2   def working(worker_no):
3       for i in range(10):
4           print("[%d] working - [%d] times" % (worker_no, i))
5
6   # main code
7   num = int(input("input number : "))
8
9   for k in range(num):
10      working(k)
```

non-Thread 결과

첫 번째 worker의 working 함수 호출
→ 10번의 working 결과 출력

두 번째 worker의 working 함수 호출
→ 10번의 working 결과 출력



함수 이용 쓰레드 생성

- **working 함수를 쓰레드로 만들어 사용**

```
1   import threading    # import threading module
2
3   # function
4   def working(worker_no):
5       for i in range(10):
6           print("[%d] working - [%d] times" % (worker_no, i))
7
8   # main code
9   num = int(input("input number : "))
10
11  for k in range(num):
12      # create thread
13      t = threading.Thread(target=working, args=(k,))
14      # start thread
15      t.start()
```

- threading모듈의 Thread 클래스 생성자 인자
  - target = 쓰레드로 수행할 함수명
  - args = 함수에 전달할 인자들
- start() : thread 시작(thread마다 한번씩 실행) → run() 함수 실행
- run() : thread 실행 구문

클래스 기반 쓰레드 생성

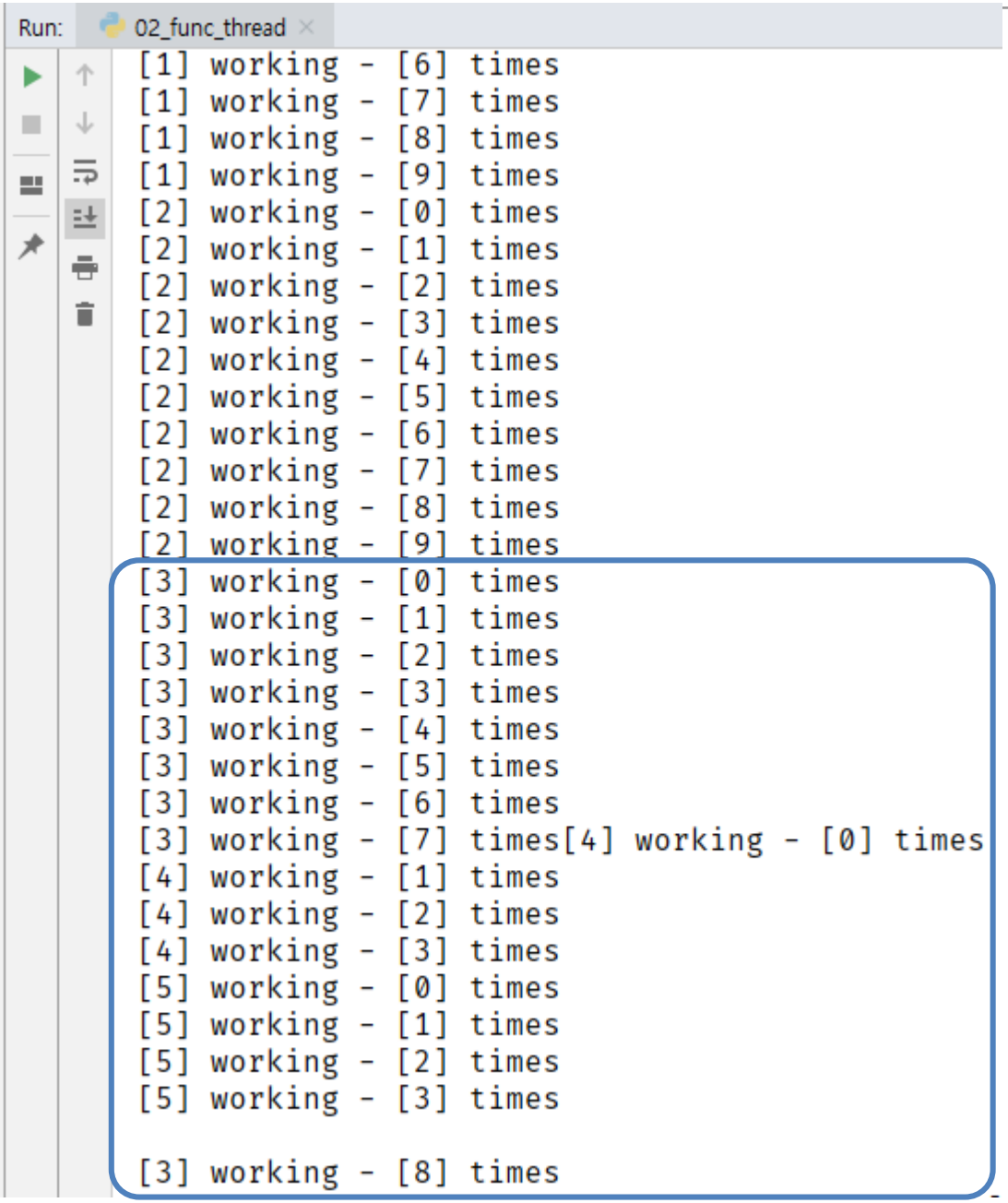
- 사용자 쓰레드 클래스를 생성하여 수행
  (threading 모듈의 Thread 클래스로부터 상속)

```
1   import threading    # import threading module
2
3   # create new_class from Thread class of threading module
4   class my_thread(threading.Thread):
5       # constructor
6       def __init__(self, no):
7           # call the constructor of parent(threading.Thread)
8           threading.Thread.__init__(self)
9           self.worker_no = no
10
11      def run(self):
12          for i in range(10):
13              print("[%d] working - [%d] times" % (self.worker_no, i))
14
15  # main code
16  num = int(input("input number : "))
17
18  for k in range(num):
19      # create thread
20      t = my_thread(k)
21      # start thread
22      t.start()
```

multi-threading 결과

생성하는 쓰레드 개수가 많을 수록...
수행 순서가 일정하지 않음

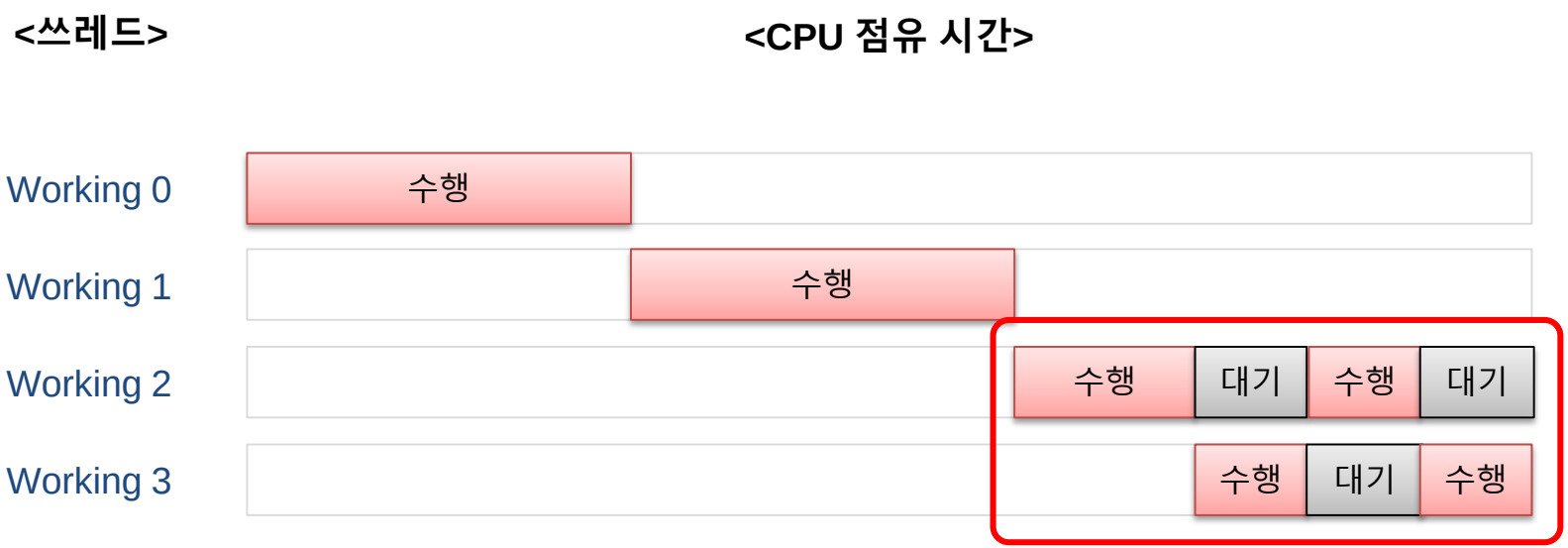
worker[3], [4], [5] 쓰레드
뒤섞여서 수행됨



CPU를 누가 점유하는가?

- 여러 개의 Thread는 경쟁적으로 CPU 사용

- CPU를 사용하게 되면 결과 출력
- 사용권이 뺏기면 다른 Thread가 결과 출력



동기화

- 병렬로 수행되는 멀티쓰레딩 환경에서는 언제 어느 쓰레드가 수행될지 모름

- 모든 쓰레드 수행 종료를 기다릴 필요 발생
  예) 모든 쓰레드 종료 후 메시지 출력

- 여러 개의 쓰레드 간 동기를 맞출 필요 발생
  예) 한 명의 worker가 일을 끝낸 후, 다음 worker가 일을 수행

쓰레드 수행 종료 대기

```python
import threading    # import threading module

# create new_class from Thread class of threading module
class my_thread(threading.Thread):
    # constructor
    def __init__(self, no):
        # call the constructor of parent(threading.Thread)
        threading.Thread.__init__(self)
        self.worker_no = no

    def run(self):
        for i in range(10):
            print("[%d] working - [%d] times" % (self.worker_no, i))

# main code
num = int(input("input number : "))

for k in range(num):
    # create thread
    t = my_thread(k)
    # start thread
    t.start()

print("---------- finish all threads ----------")
```

```
[96] working - [0] times[63] working - [1] times[97] working - [0] times[48] working - [2] times[98] working - [0] times
[99] working - [0] times[97] working - [1] times---------- finish all threads ----------


[69] working - [1] times[52] working - [1] times[50] working - [1] times[67] working - [1] times
```

쓰레드 수행 종료 대기

- **join() : 쓰레드 종료 대기 함수**

```python
import threading    # import threading module

# create new_class from Thread class of threading module
class my_thread(threading.Thread):
    # constructor
    def __init__(self, no):
        # call the constructor of parent(threading.Thread)
        threading.Thread.__init__(self)
        self.worker_no = no

    def run(self):
        for i in range(10):
            print("[%d] working - [%d] times" % (self.worker_no, i))


# main code
ths = []        # threads list

num = int(input("input number : "))

for k in range(num):
    # create thread
    t = my_thread(k)
    # start thread
    t.start()
    # add thread to list
    ths.append(t)

# wait for all threads to complete
for t in ths:
    t.join()

print("---------- finish all threads ----------")
```

쓰레드 간 동기화

- **Lock** : lock이 걸린 동안 다른 쓰레드가 그 변수에 접근하지 못함 (일종의 임계영역: Critical Section)
- **threading.Lock()** : lock 객체 생성
- **acquire()** : lock 시작
- **release()** : lock 해제
- **locked()** : lock 상태에 있으면 1, 아니면 0

```python
import threading    # import threading module

# create new_class from Thread class of threading module
class my_thread(threading.Thread):
    # constructor
    def __init__(self, no):
        # call the constructor of parent(threading.Thread)
        threading.Thread.__init__(self)
        self.worker_no = no

    def run(self):
        # set lock (start of critical_section)
        t_lock.acquire()

        for i in range(10):
            print("[%d] working - [%d] times" % (self.worker_no, i))

        # release lock (end of critical_section)
        t_lock.release()

# main code
t_lock = threading.Lock()    # thread lock
ths = []        # threads list

num = int(input("input number : "))

for k in range(num):
    # create thread
    t = my_thread(k)
    # start thread
    t.start()
    # add thread to list
    ths.append(t)

# wait for all threads to complete
for t in ths:
    t.join()

print("---------- finish all threads ----------")
```

보호 구역

그 외 동기화를 위한 threading 모듈 함수

- **threading.Condition() : Condition Variable, 내부에 하나의 쓰레드 대기큐 가짐**
  - wait() : 대기큐에 들어가며 sleep 상태에 들어감
  - notify() : 대기큐에서 하나의 쓰레드를 깨움

- **threading.Event() : 내부에 하나의 이벤트 플래그를 가짐 (초기값 0)**
  - set() : 내부 플래그 1로 set
  - clear() : 0으로 set
  - wait() : 내부 플래그가 1이면 즉시 리턴, 0이면 다른 쓰레드에 의해서 1이 될 때까지 대기 (내부 플래그 값 변경 안함)

- **threading.RLock() : Lock 객체와 같으나, lock을 소유하고 있는 쓰레드가 두 번 이상 acquire와 release 호출 가능 (acquire 한 만큼 release 해야 lock 해제됨)**

그 외 동기화를 위한 threading 모듈 함수

- **threading.Semaphore([value])**
  - 세마포어는 카운터로 관리
  - acquire() : 카운터 -1
  - release() 카운터 +1

  - 만약, acquire() 실행 시 카운터 값이 0이면 쓰레드는 대기큐에서 대기
  - release() 함수는 우선 대기 쓰레드 검사.
    대기 쓰레드 중 가장 오래된 대기 쓰레드 하나 해제.
    만약 대기큐에 쓰레드가 없다면, 카운터 값만 +1

- **threading.BoundedSemaphore([value])**
  - 현재 값이 초기 값을 초과하지 않는지 검사하는 기능 추가
  - 초기 값 초과할 경우 ValueError 예외 발생
  - 주로 제한된 자원 관리 시 사용

파이썬 멀티쓰레드

쓰레드를 여러 개 사용하면 빨라질까?

- 간단한 덧셈 프로그램
- 입력한 수까지 더한 값 출력
- 소요된 시간 출력

```python
from threading import Thread     # import threading module
import time                      # import time module

def do_work(start_no, end_no, result):
    sum = 0
    for i in range(start_no, end_no):
        sum += i
    result.append(sum)

# main code
while True:
    num = int(input("input number : "))

    start = 0
    end = num
    result = []

    # start time
    t1 = time.time()

    # create single thread
    th1 = Thread(target=do_work, args=(start, end, result))
    th1.start()
    th1.join()

    # result
    print("Result : %d" % sum(result))

    # stop time
    t2 = time.time()
    print("================= [%.3f sec]" % (t2-t1))

    t1, t2 = 0.0, 0.0
```

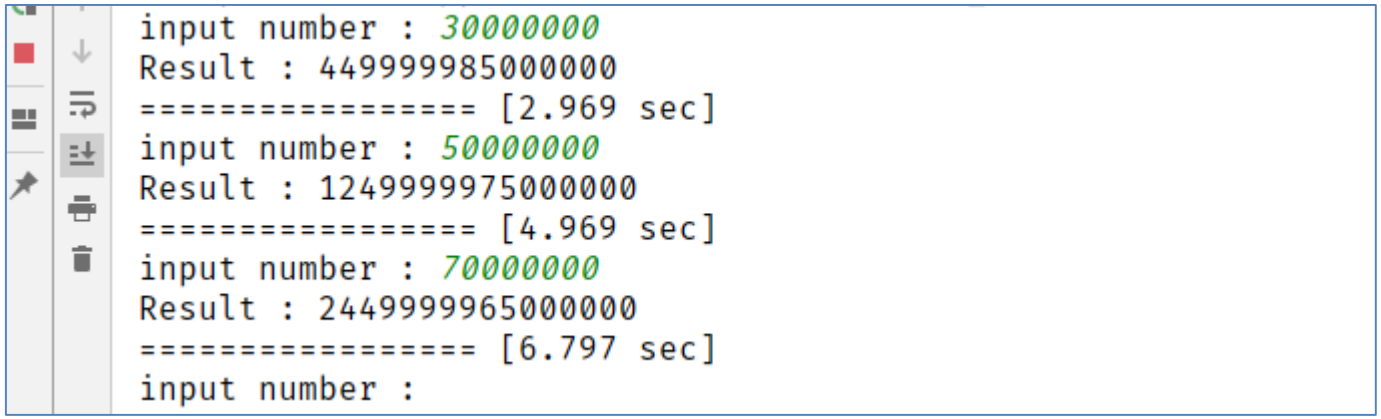
쓰레드를 여러 개 사용하면 빨라질까?

- 2개의 쓰레드로 나누어서 계산

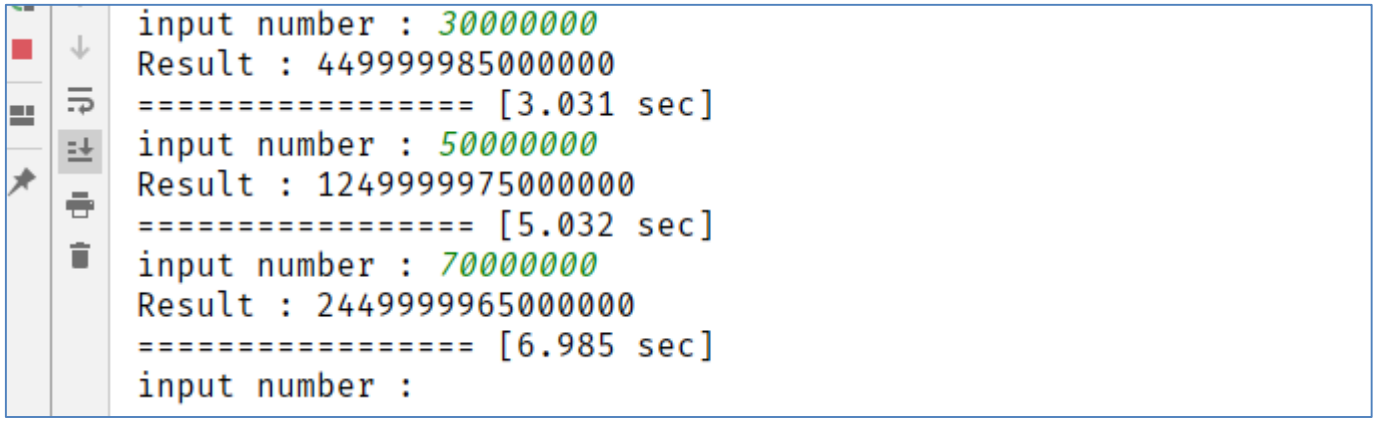
```python
from threading import Thread     # import threading module
import time                      # import time module

def do_work(start_no, end_no, result):
    sum = 0
    for i in range(start_no, end_no):
        sum += i
    result.append(sum)

# main code
while True:
    num = int(input("input number : "))

    start = 0
    end = num
    result = []

    # start time
    t1 = time.time()

    # create two threads
    th1 = Thread(target=do_work, args=(start, int(end/2), result))
    th2 = Thread(target=do_work, args=(int(end/2), end, result))

    th1.start()
    th2.start()

    th1.join()
    th2.join()

    # result
    print("Result : %d" % sum(result))

    # stop time
    t2 = time.time()
    print("================= [%.3f sec]" % (t2 - t1))

    t1, t2 = 0.0, 0.0
```

쓰레드를 여러 개 사용하면 빨라질까?

- 쓰레드 1개 수행 결과



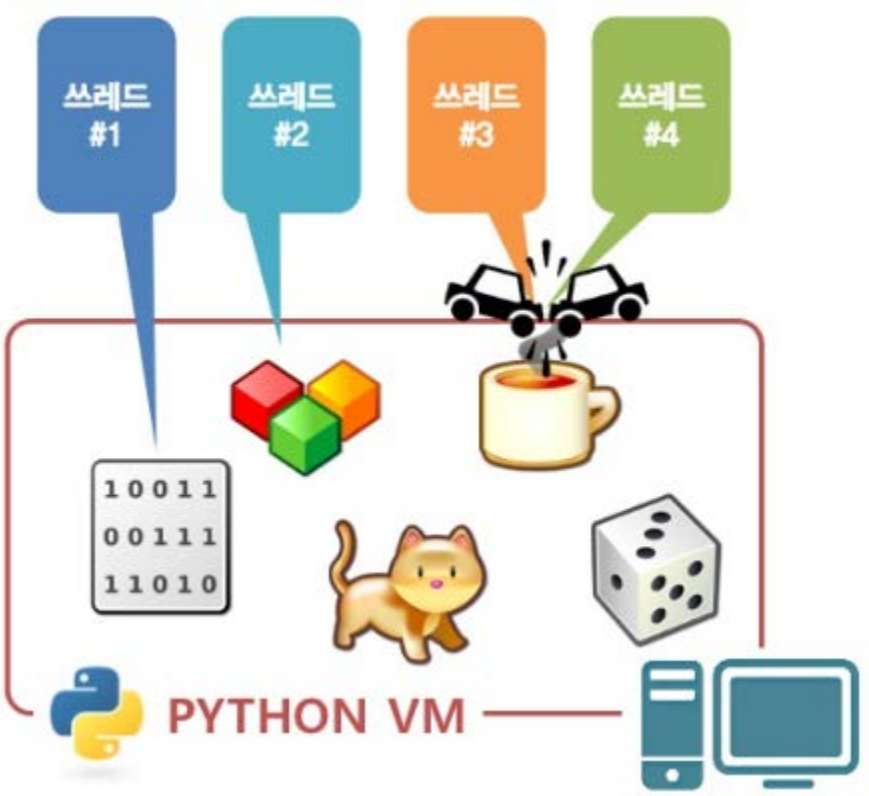
- 쓰레드 2개 수행 결과



➔ 더 느림

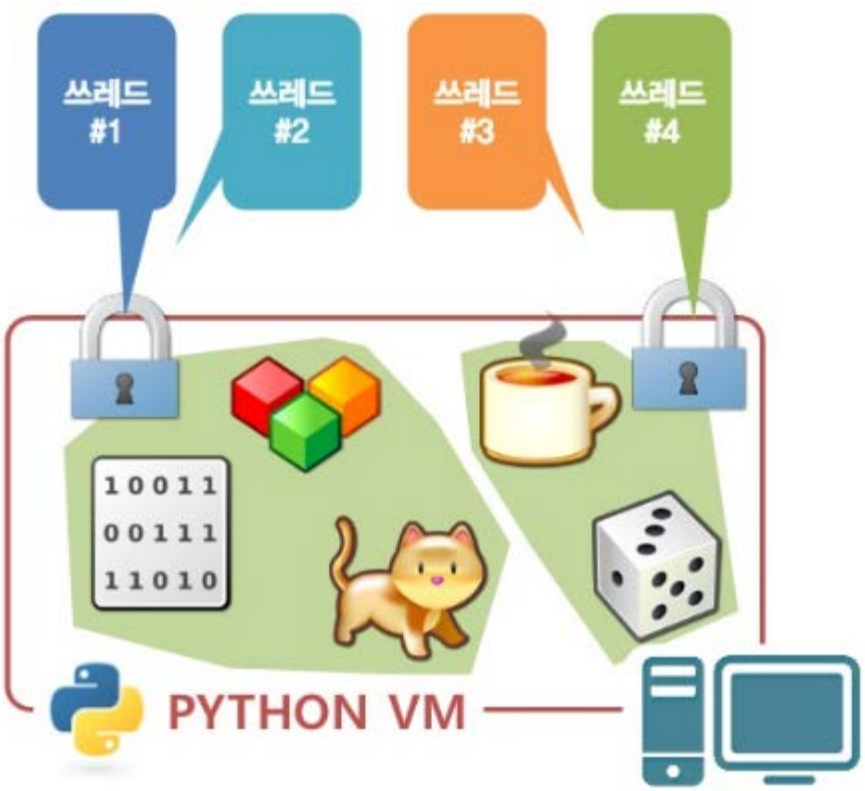
쓰레드 메커니즘

- 인터프리터에서도 자원 보호 필요



쓰레드 메커니즘

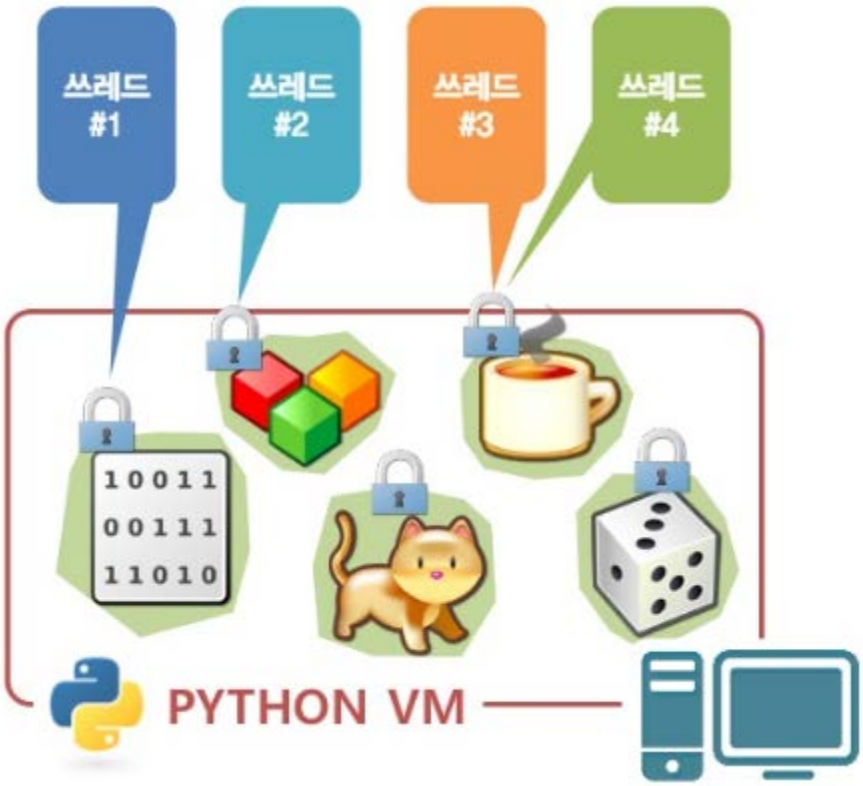
- **Coarse-Grained Lock : 대충 크게 묶어서 lock 설정**



- **예) disk가 여러 개 있을 때 여러 개의 disk 사용에 대해 전체 lock**

쓰레드 메커니즘

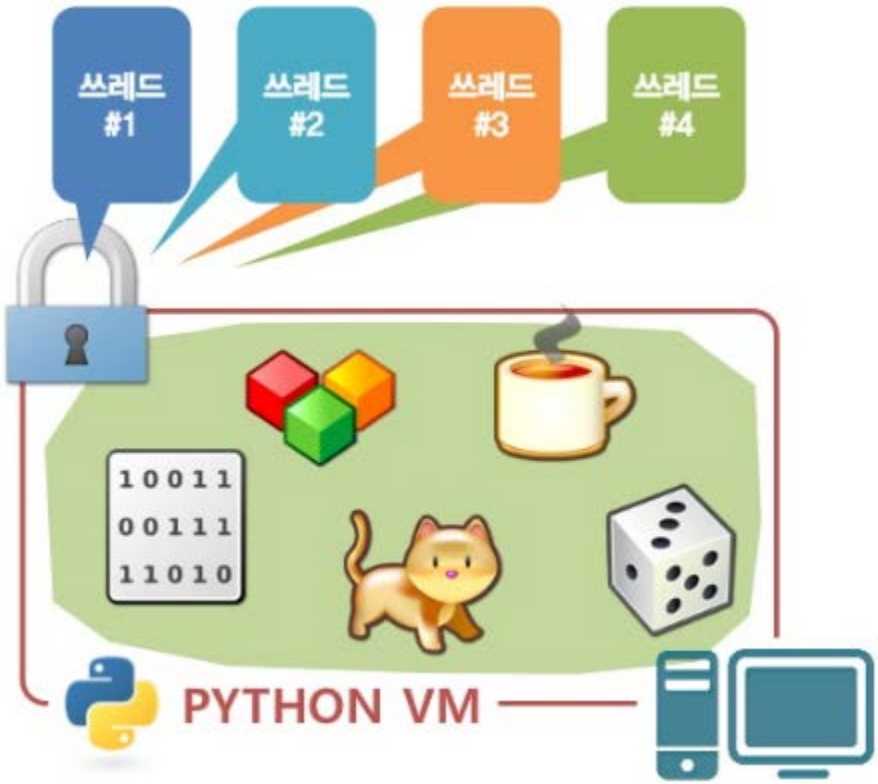
- **Fine-Grained Lock : 아주 세밀하게 lock 설정**



- **예) disk가 여러 개 있을 때 하나의 disk 사용에 대해 lock**

쓰레드 메커니즘

- 파이썬에서 사용하는 쓰레드 메커니즘
- Global Interpreter Lock (GIL)



➔ 파이썬은 사실상 Single Thread

➔ 멀티쓰레드로 구현했음에도 불구하고, 사실상 GIL에 의해 예외적인 경우를 제외하고, 한 시점에는 하나의 thread만 동작하는 것처럼 보임

Global Interpreter Lock

- **인터프리터 구현 쉬움**
- **Garbage Collector 만들기 좋음**
- **C/C++ 확장 모듈 만들기 쉬움 : c library의 thread-safety 고려하지 않아도 됨**
- **I/O Bound 동작이 많은 경우에는 이득**
  - CPU bound 동작 : 주로 CPU에서 수행하는 일(계산)이 많은 동작
    예) 압축, 정렬, 인코딩 등
  - I/O bound 동작 : CPU보다 외부 장치에서 수행하는 일이 많은 동작
    실제로 수행 시간이 더 많이 소요되며, CPU보다 느림
    예) 네트워크, 디스크 사용 (파일 쓰기, 읽기), 키보드, 마우스 사용 등

➔ 우리가 작성하는 일반적 프로그래밍은 대부분 I/O bound가 많음

Global Interpreter Lock

https://docs.python.org/3.7/library/threading.html?highlight=thread#module-threading

CPython implementation detail: In CPython, due to the Global Interpreter Lock, only one thread can execute Python code at once (even though certain performance-oriented libraries might overcome this limitation). If you want your application to make better use of the computational resources of multi-core machines, you are advised to use `multiprocessing` or `concurrent.futures.ProcessPoolExecutor`. However, threading is still an appropriate model if you want to run multiple I/O-bound tasks simultaneously.

파이썬 멀티프로세싱

멀티프로세싱

- 쓰레드를 여러 개 생성하는 것이 아니라, 여러 개의 프로세스를 멀티 코어에 할당하여 연산 수행
- 프로세스 간 객체 공유 : 큐 / 파이프

```
1   from multiprocessing import Process, Queue  # import multiprocessing module
2   import time                                 # import time module
3
4   def do_work(start_no, end_no, result):
5       sum = 0
6       for i in range(start_no, end_no):
7           sum += i
8       result.put(sum)
9
10  if __name__ == "__main__":
11      # main code
12      while True:
13          num = int(input("input number : "))
14
15          start = 0
16          end = num
17          result = Queue()
18
19          # start time
20          t1 = time.time()
21
22          # create two processes
23          pr1 = Process(target=do_work, args=(start, int(end/2), result))
24          pr2 = Process(target=do_work, args=(int(end/2), end, result))
25
26          pr1.start()
27          pr2.start()
28
29          pr1.join()
30          pr2.join()
```

멀티프로세싱

```
31
32          # result
33          result.put("STOP")
34          sum = 0
35          while True:
36              tmp = result.get()
37              if tmp == "STOP": break
38              else: sum += tmp
39
40          print("Result : %d" % sum)
41
42          # stop time
43          t2 = time.time()
44          print("================== [%.3f sec]" % (t2 - t1))
45
46          t1, t2 = 0.0, 0.0
```

➔ 멀티프로세싱 결과

```
input number : 30000000
Result : 449999985000000
================== [1.719 sec]
input number : 50000000
Result : 1249999975000000
================== [2.875 sec]
input number : 70000000
Result : 2449999965000000
================== [4.078 sec]
input number : 100000000
Result : 4999999950000000
================== [5.391 sec]
```

Turtle Module

Turtle

- **간단한 그림을 그릴 수 있는 그래픽 모듈**

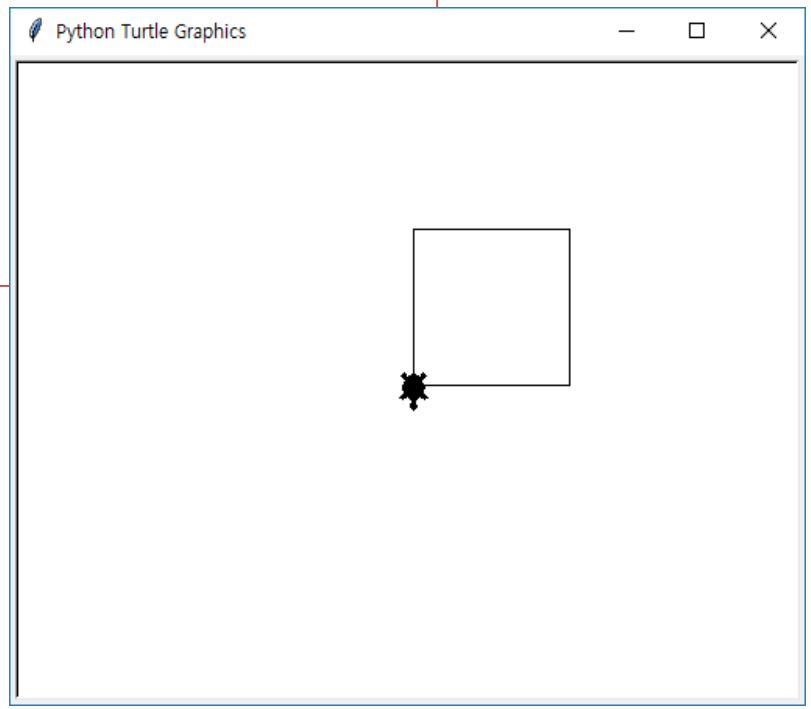
- **기본 명령 예제**
  - tt = Turtle('turtle') : 거북이 모양 아이콘을 가진 turtle 객체 생성

  - tt.speed(속도값) : 0 : 최고속도
1 : 가장 느린 속도
10 : 빠른 속도

  - tt.forward(값) : 픽셀값만큼 앞으로 이동
  - tt.backward(값) : 픽셀값만큼 뒤로 이동
  - tt.left(값) : 각도값만큼 왼쪽으로 회전
  - tt.right(값) : 각도값만큼 오른쪽으로 회전

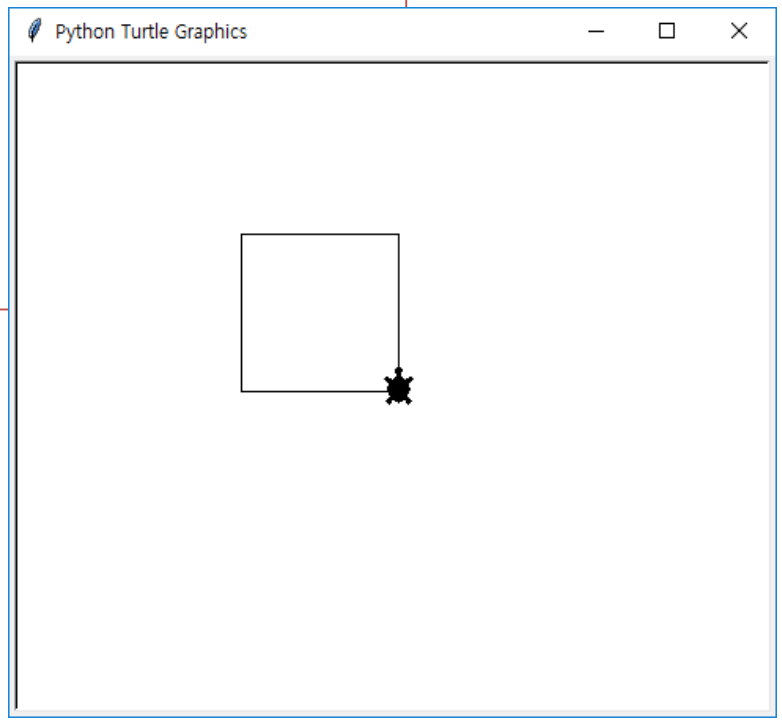
사각형 그리기

```
1   # import module
2   import turtle
3
4   # create turtle object
5   tt = turtle.Turtle("turtle")
6
7   tt.forward(100)      # forward
8   tt.left(90)          # left turn 90
9   tt.forward(100)      # forward
10  tt.left(90)          # left turn 90
11  tt.forward(100)      # forward
12  tt.left(90)          # left turn 90
13  tt.forward(100)      # forward
14
15  # turtle done
16  turtle.done()
```



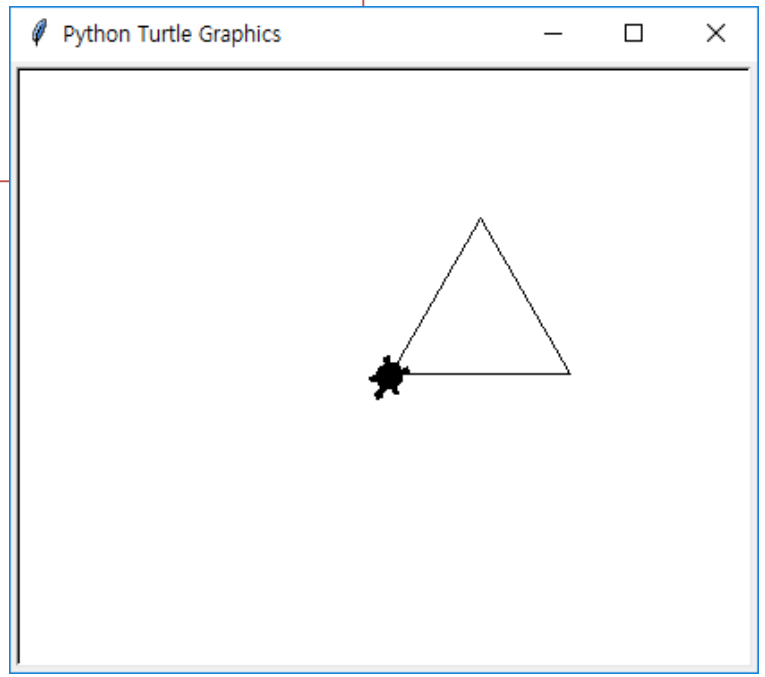
사각형 그리기

```
1   # import module
2   import turtle
3
4   # create turtle object
5   tt = turtle.Turtle("turtle")
6
7   tt.backward(100)      # backward
8   tt.right(90)          # right turn 90
9   tt.backward(100)      # backward
10  tt.right(90)          # right turn 90
11  tt.backward(100)      # backward
12  tt.right(90)          # right turn 90
13  tt.backward(100)      # backward
14
15  # turtle done
16  turtle.done()
```



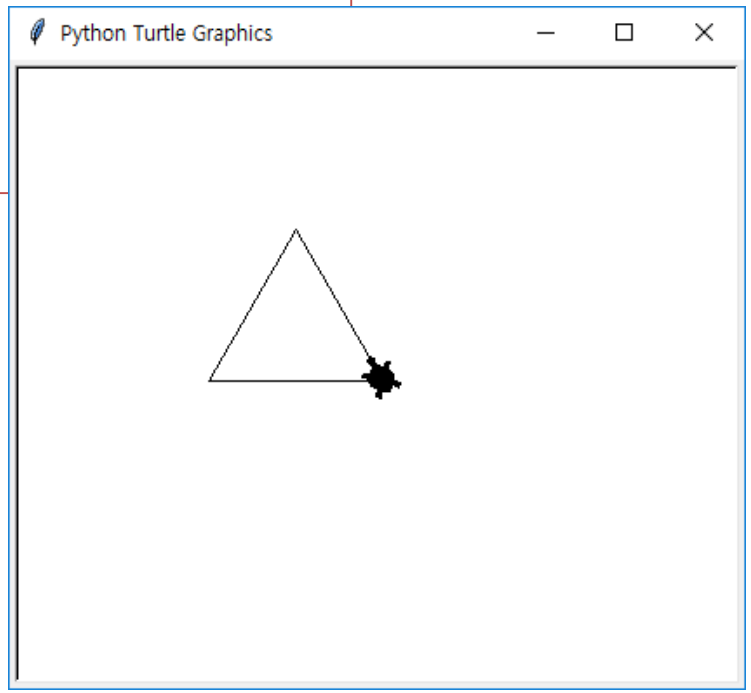
삼각형 그리기

```
1   # import module
2   import turtle
3
4   # create turtle object
5   tt = turtle.Turtle("turtle")
6
7   tt.forward(100)      # forward
8   tt.left(120)         # left turn 120
9   tt.forward(100)      # forward
10  tt.left(120)         # left turn 120
11  tt.forward(100)      # forward
12
13  # turtle done
14  turtle.done()
```



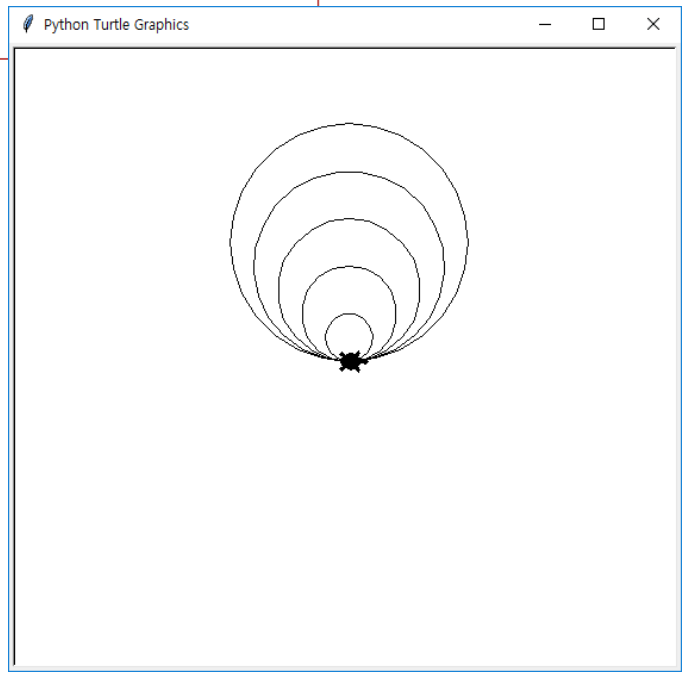
삼각형 그리기

```
1   # import module
2   import turtle
3
4   # create turtle object
5   tt = turtle.Turtle("turtle")
6
7   tt.backward(100)      # backward
8   tt.right(120)         # right turn 120
9   tt.backward(100)      # backward
10  tt.right(120)         # right turn 120
11  tt.backward(100)      # backward
12
13  # turtle done
14  turtle.done()
```



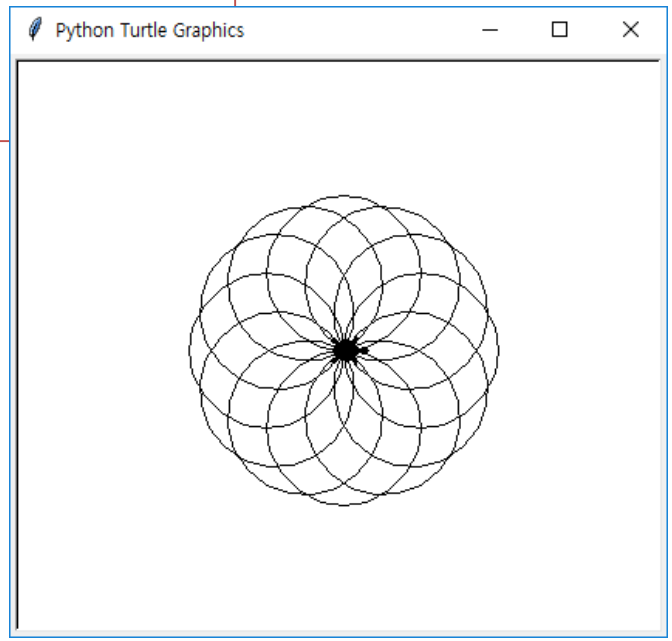
원 그리기

```
1   # import module
2   import turtle
3
4   # create turtle object
5   tt = turtle.Turtle("turtle")
6
7   tt.circle(100)  # circle radius 100
8   tt.circle(80)  # circle radius 70
9   tt.circle(60)  # circle radius 40
10  tt.circle(40)  # circle radius 10
11  tt.circle(20)  # circle radius 20
```



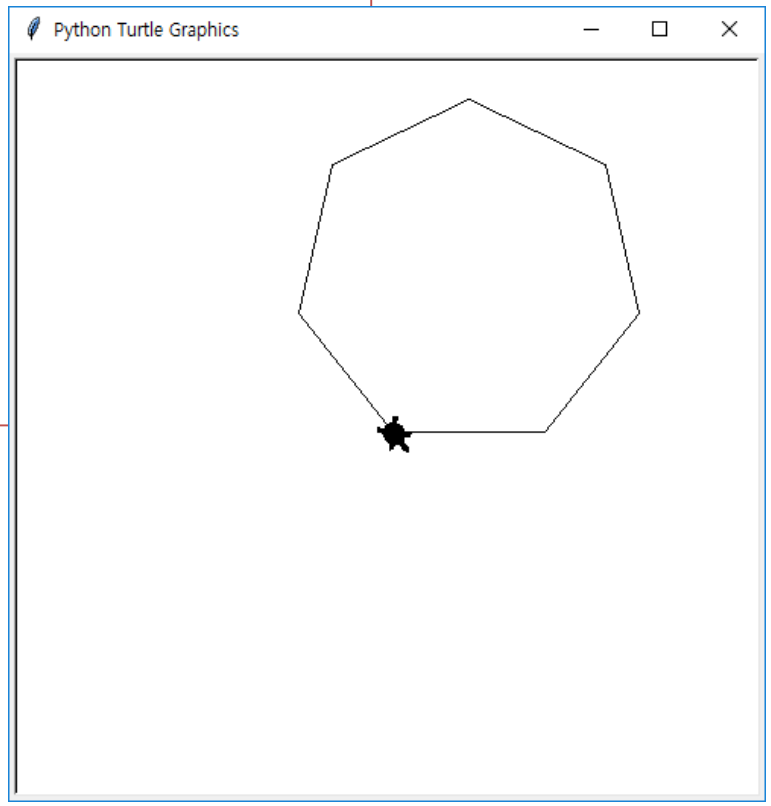
다중 원 그리기

```
1   # import module
2   import turtle
3
4   # create turtle object
5   tt = turtle.Turtle("turtle")
6
7   theta = 0
8   while theta < 360:
9       tt.circle(50)  # circle
10      tt.left(30)     # turn left
11      theta += 30
12
13  # turtle done
14  turtle.done()
```



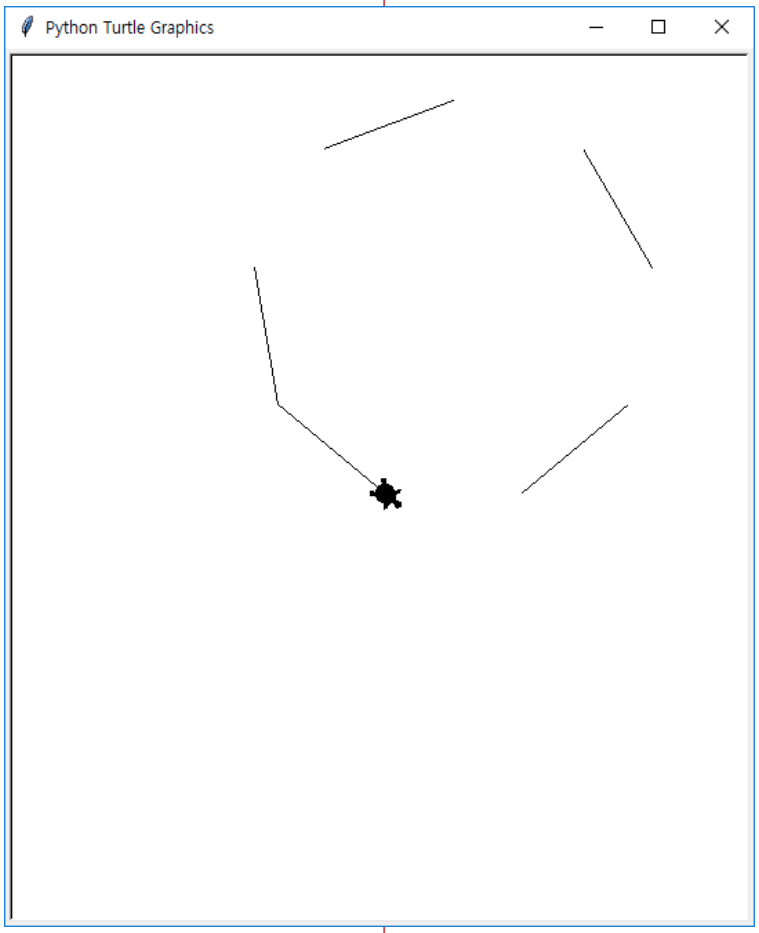
다각형 함수

```
1   # import module
2   import turtle
3
4   # function
5   def polygon(tt, size, angle):
6       for i in range(0, angle-1):
7           tt.forward(size)        # forward
8           tt.left(360/angle)      # turn left
9       tt.forward(size)  # forward
10
11  # create turtle object
12  tt = turtle.Turtle("turtle")
13
14  polygon(tt, 100, 7)
15
16  # turtle done
17  turtle.done()
```



penup(), pendown()

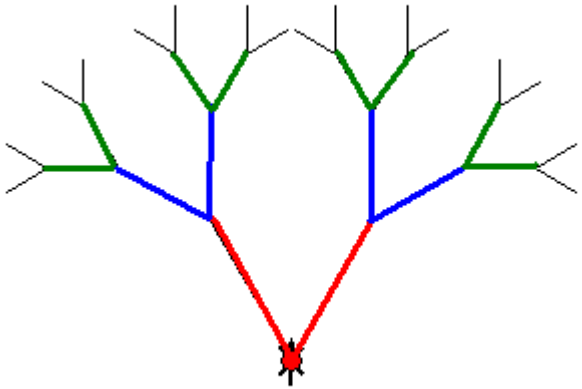
```
1  # import module
2  import turtle
3
4  # function
5  def polygon(tt, size, angle):
6      flag = 0
7      for i in range(0, angle-1):
8          if flag == 1:
9              tt.pendown()     # drawing
10             flag = 0
11         elif flag == 0:
12             tt.penup()       # no drawing
13             flag = 1
14
15         tt.forward(size)         # forward
16         tt.left(360/angle)       # turn left
17
18     tt.forward(size)  # forward
19
20 # create turtle object
21 tt = turtle.Turtle("turtle")
22
23 polygon(tt, 100, 9)
24
25 # turtle done
26 turtle.done()
```



나무 그리기

- **나무 기본 구조**
  - turn left 30도
  - forward 길이
  - backward 돌아감
  - turn right 60도
  - forward 길이
  - backward 돌아감

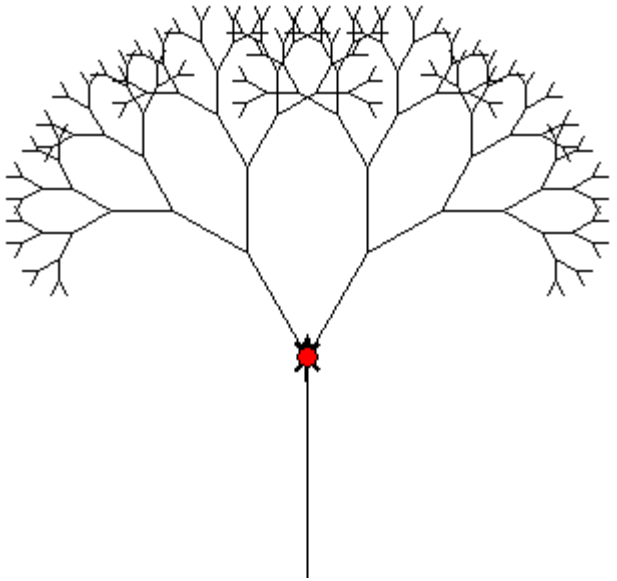
- **나무들의 크기만 작아질 뿐, 같은 모양**
  - 빨간색, 파란색, 초록색 나무 모양 같음
  - 크기는 점점 작아지는 형태



나무 그리기 재귀 함수

- **나무 그리기 반복**
  - 왼쪽 가지 그리고, 거기에 달린 작은 나무 그리고,
  - 오른쪽 가지 그리고, 거기에 달린 작은 나무 그리고…

- **나무 그리기 재귀 함수 (tree)**
  - 왼쪽 가지
  - tree (좀 더 작은 길이)
  - 제자리로
  - 오른쪽 가지
  - tree (좀 더 작은 길이)
  - 제자리로

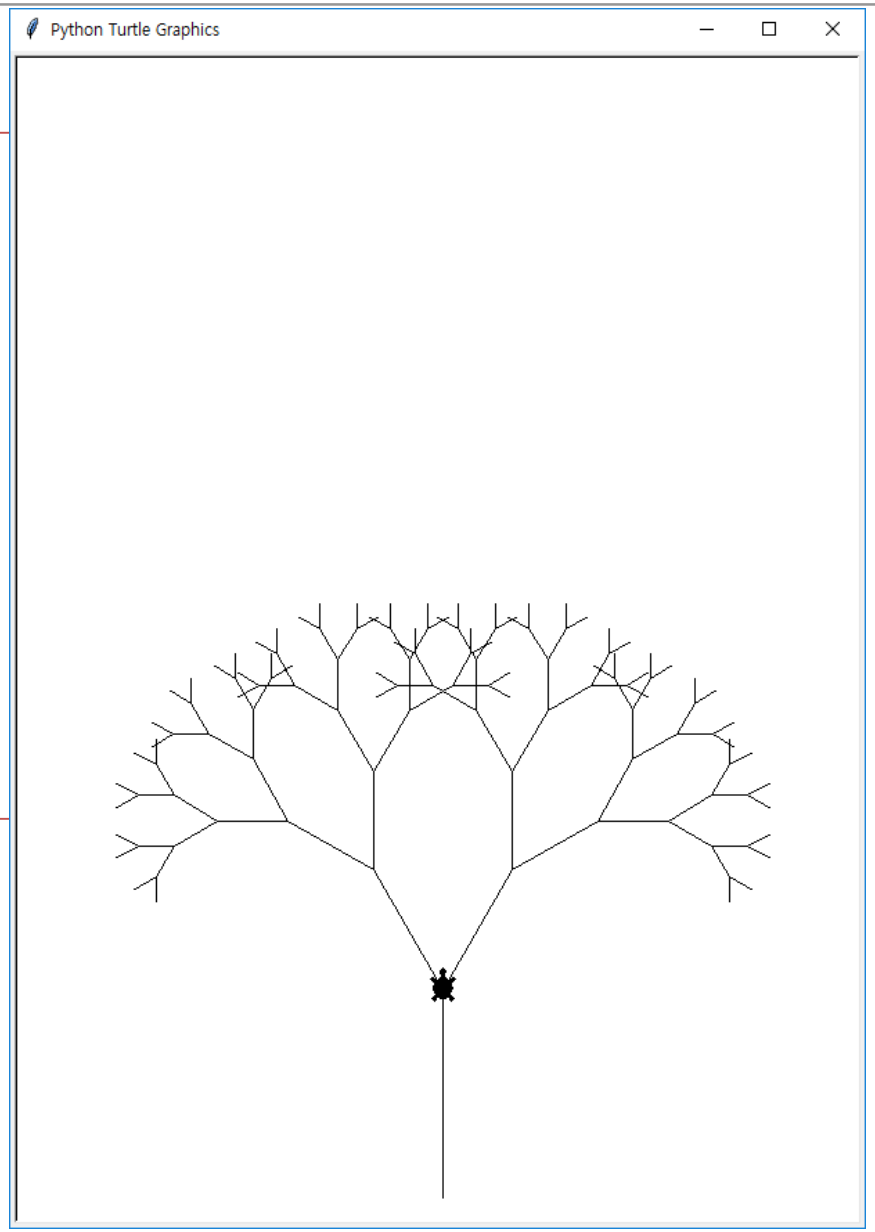


나무 그리기

```
# import module
import turtle

# function
def tree(tt, length, n):
    if n >= 0:
        # left branch
        tt.left(30)
        tt.forward(length)

        # small branch
        tree(tt, length/1.4, n-1)

        # return
        tt.penup()
        tt.backward(length)

        # right branch
        tt.right(60)
        tt.pendown()
        tt.forward(length)

        # small branch
        tree(tt, length/1.4, n-1)

        # return
        tt.penup()
        tt.backward(length)
        tt.left(30)

```

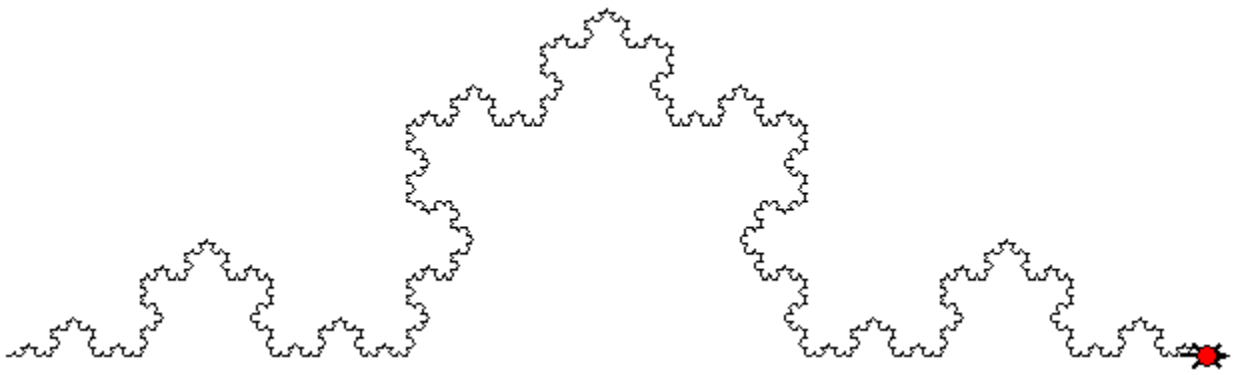
나무 그리기

```
30
31  # create turtle object
32  tt = turtle.Turtle("turtle")
33
34  # start point
35  tt.penup()
36  tt.goto(0, -400)
37  tt.left(90)
38
39  tt.pendown()
40  tt.forward(150)
41
42  # drawing tree
43  tree(tt, 100, 5)
44
45  # turtle done
46  turtle.done()
```



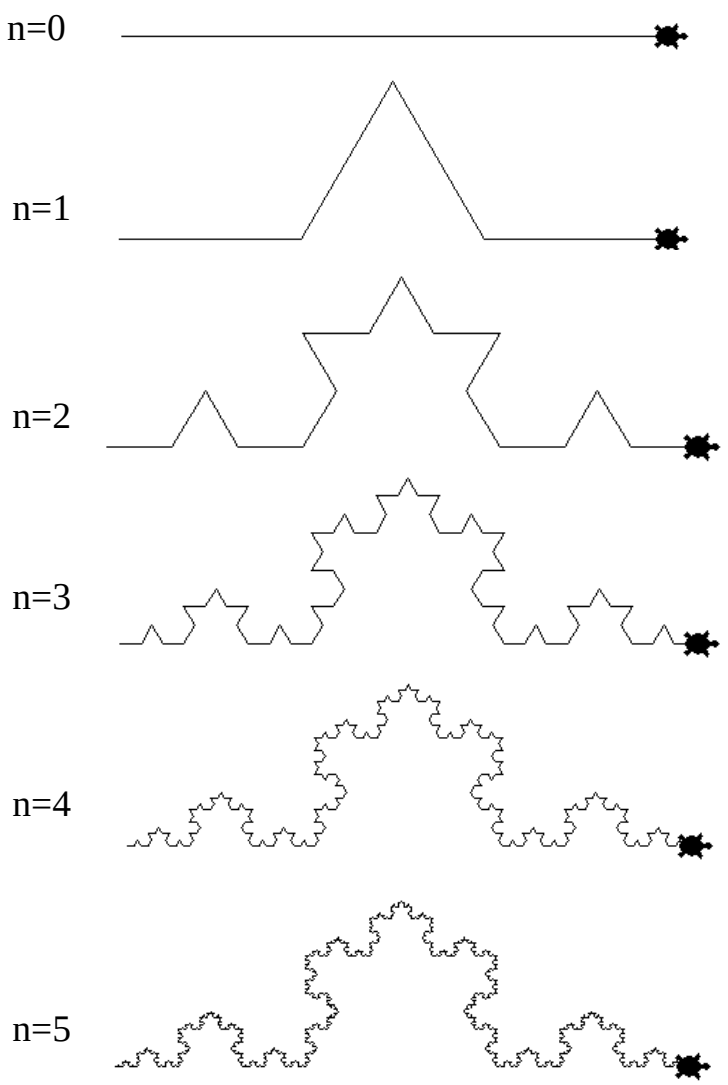
Koch 곡선

- **Koch 곡선 생성 법칙 (n != 0 동안)**
  - 작은 Koch
  - 왼쪽으로 60도 회전
  - 작은 Koch
  - 오른쪽으로 120도 회전
  - 작은 Koch
  - 왼쪽으로 60도 회전
  - 작은 Koch
- **n == 0 이면 forward()**

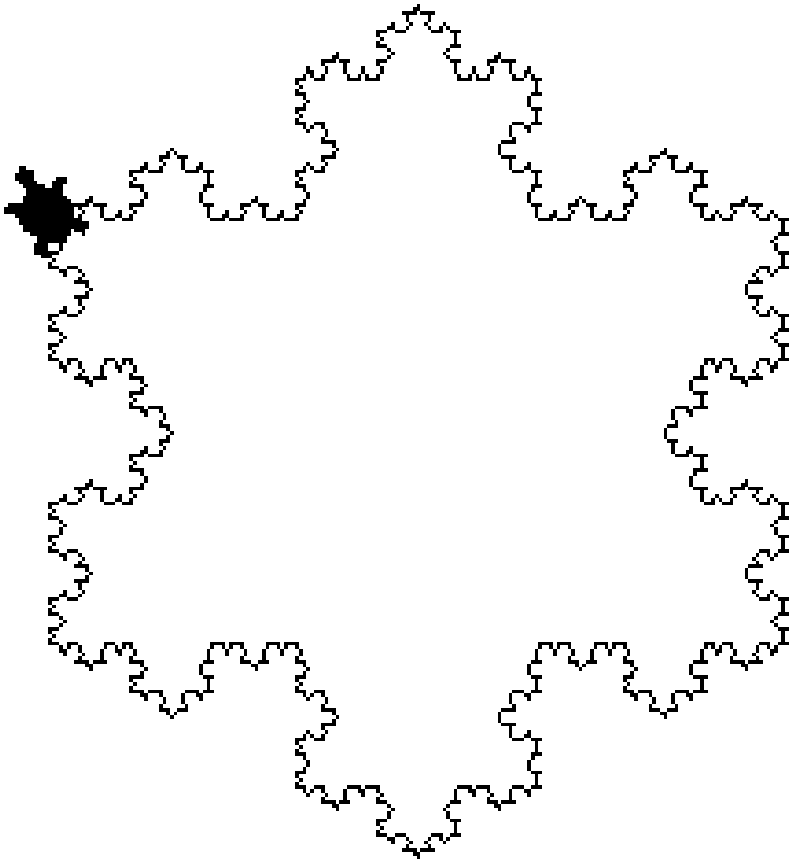


Koch 곡선

```python
# import module
import turtle

# function
def koch(tt, length, n):
    if n > 0:
        # small koch
        koch(tt, length/3.0, n-1)
        # turn left
        tt.left(60)
        # small koch
        koch(tt, length/3.0, n-1)
        # turn right
        tt.right(120)
        # small koch
        koch(tt, length/3.0, n-1)
        # turn left
        tt.left(60)
        # small koch
        koch(tt, length/3.0, n-1)

    elif n == 0:
        tt.forward(length)

# create turtle object
tt = turtle.Turtle("turtle")

# start point
tt.penup()
tt.goto(-200, 0)
tt.pendown()

# drawing koch
koch(tt, 400, 5)

# turtle done
turtle.done()
```



코흐의 눈송이



Any Questions...
Just Ask!