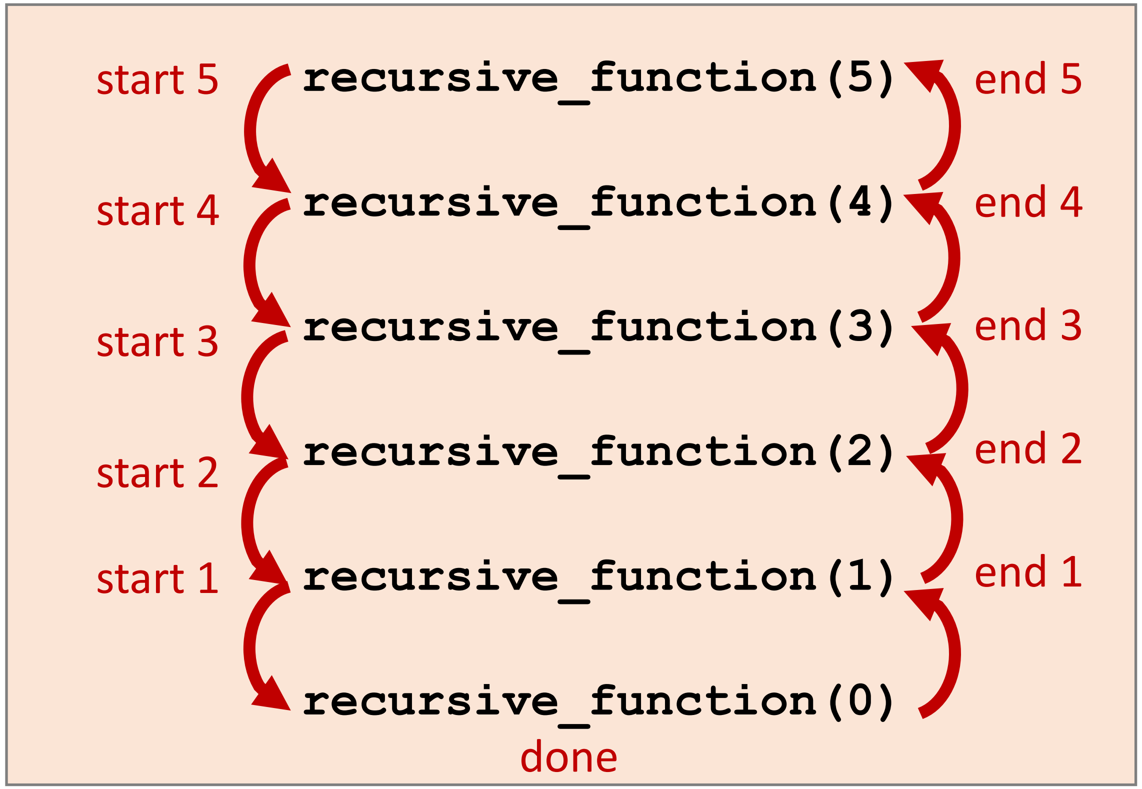
Recursion

- symbol table
- stack frames

Recursion

Recursive function
≡
"function that calls itself"



Python shell

```
> def recursive_function(x):
      if x > 0:
          print("start", x)
          recursive_function(x - 1)
          print("end", x)
      else:
          print("done")

> recursive_function(5)
| start 5
| start 4
| start 3
| start 2
| start 1
| done
| end 1
| end 2
| end 3
| end 4
| end 5
```

Recursion

x	0	stack frame
x	1	
x	2	
x	3	
x	4	
x	5	
recursive_function	<function>	global variables

Recursions stack when x = 0 is reached

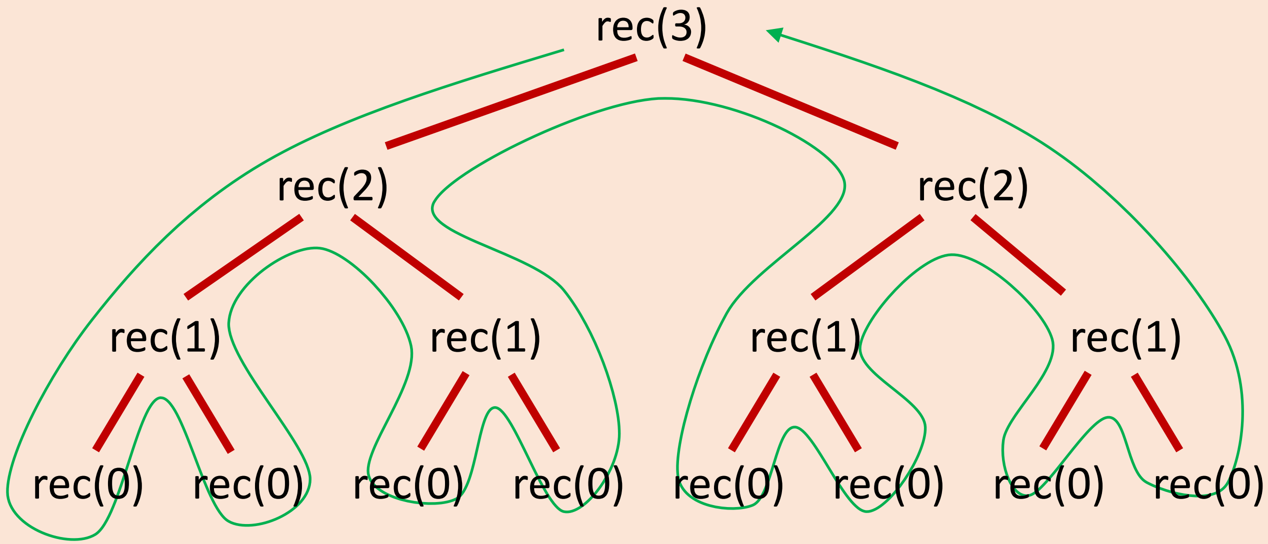
Python shell

```
> def recursive_function(x):
      if x > 0:
          print("start", x)
          recursive_function(x - 1)
          print("end", x)
      else:
          print("done")

> recursive_function(5)
| start 5
| start 4
| start 3
| start 2
| start 1
| done
| end 1
| end 2
| end 3
| end 4
| end 5
```

Python shell

```
> def rec(x):
      if x > 0:
          print("start", x)
          rec(x - 1)
          rec(x - 1)
          print("end", x)
      else:
          print("done")
```

Recursion tree



Python shell

```
> rec(3)
| start 3
| start 2
| start 1
| done
| done
| end 1
| start 1
| done
| done
| end 1
| end 2
| start 2
| start 1
| done
| done
| end 1
| start 1
| done
| done
| end 1
| end 2
| end 3
```

Question – How many times does `rec(5)` print "`done`"?

Python shell

```
> def rec(x):
      if x > 0:
          print("start", x)
          rec(x - 1)
          rec(x - 1)
          rec(x - 1)
          print("end", x)
      else:
          print("done")
```

a) 3

b) 5

c) 15

d) 81

e) 125

f) $243 = 3^5$

g) Don't know

Factorial

$$n\,! = n \cdot \underbrace{(n-1)\cdot(n-2)\cdots 3\cdot 2\cdot 1}_{(n-1)\,!}$$

Observation
(recursive definition)

$$1\,! = 1$$

$$n\,! = n \cdot (n-1)\,!$$

factorial.py

```
def factorial(n):
    if n <= 1:
        return 1
    return n * factorial(n - 1)
```

factorial_iterative.py

```
def factorial(n):
    result = 1
    for i in range(2, n + 1):
        result *= i
    return result
```

factorial.py

```
def factorial(n):
    return n * factorial(n - 1) if n > 1 else 1
```

Binomial coefficient $\binom{n}{k}$

- $\binom{n}{k}$ = number of ways to pick k elements from a set of size n
- $\binom{n}{k} = \begin{cases} 1 & \text{if } k = 0 \text{ or } k = n \\ \binom{n-1}{k} + \binom{n-1}{k-1} & \text{otherwise} \end{cases}$

binomial_recursive.py

```python
def binomial(n, k):
    if k == 0 or k == n:
        return 1
    return binomial(n - 1, k) + binomial(n - 1, k - 1)
```

- Unfolding computation shows $\binom{n}{k}$ 1's are added → slow

Readable functions ? – return early / bail out fast

- Treat simple cases first and return
- Do not put `else` after `if` ending with `return`
- Avoid unnecessary nesting of code
- 1-liners are not always the most readable code

binomial_return_early.py

```
def binomial(n, k):  # Ugly, nested indentations and redundant else
    if k == 0:
        return 1
    else:
        if k == n:
            return 1
        else:
            return binomial(n - 1, k) + binomial(n - 1, k - 1)

def binomial(n, k):  # Treat each special case first and return
    if k == 0:
        return 1
    if k == n:
        return 1
    return binomial(n - 1, k) + binomial(n - 1, k - 1)

def binomial(n, k):   # Several cases simultaneously - is test obvious?
    if k == 0 or k == n:
        return 1
    return binomial(n - 1, k) + binomial(n - 1, k - 1)

def binomial(n, k):  # 1-liner, but is this the easiest to read?
    return binomial(n - 1, k) + binomial(n - 1, k - 1) if 0 < k < n else 1
```

See also *Clean Code - Handbook of Agile Software Craftsmanship*, by Robert C Martin (2009)

Tracing the recursion

- At beginning of function call, print arguments
- Before returning, print return value
- Keep track of recursion depth in a argument to print indentation

binomial_trace.py

```
def binomial(n, k, indent=0):
    print('    ' * indent + f'binomial({n}, {k})')
    if k == 0 or k == n:
        result = 1
    else:
        result = binomial(n - 1, k, indent=indent + 1) + \
                 binomial(n - 1, k - 1, indent=indent + 1)
    print('    ' * indent + f'return {result}')
    return result
```

Python shell

```
> binomial(4, 2)
| binomial(4, 2)
|     binomial(3, 2)
|         binomial(2, 2)
|         return 1
|         binomial(2, 1)
|             binomial(1, 1)
|             return 1
|             binomial(1, 0)
|             return 1
|         return 2
|     return 3
|     binomial(3, 1)
|         binomial(2, 1)
|             binomial(1, 1)
|             return 1
|             binomial(1, 0)
|             return 1
|         return 2
|         binomial(2, 0)
|         return 1
|     return 3
| return 6
| 6
```

Binomial coefficient $\binom{n}{k}$

Observation $\binom{n}{k} = \dfrac{n\,!}{(n-k)\,!\cdot k\,!}$

binomial_factorial.py

```
def binomial(n, k):
    return factorial(n) // factorial(k) // factorial(n - k)
```

- Unfolding computation shows 2n - 2 multiplications and 2 divisions → fast
- Intermediate value n ! can have significantly more digits than result (bad)

Binomial coefficient $\binom{n}{k}$

Observation $$\binom{n}{k} = \frac{n \cdot (n-1) \cdot (n-2) \cdots (n-k+1)}{k \cdot (k-1) \cdot (k-2) \cdots 1} = \binom{n-1}{k-1} \cdot \frac{n}{k}$$

binomial_recursive_product.py

```
def binomial(n, k):
    if k == 0:
        return 1
    else:
        return binomial(n - 1, k - 1) * n // k
```

- Unfolding computation shows k multiplications and divisions → fast
- Multiplication with fractions ≥ 1 → intermediate numbers limited size

Questions – Which correctly computes $\binom{n}{k}$?

Observation $\binom{n}{k} = \dfrac{n \cdot (n-1) \cdot (n-2) \cdots (n-k+1)}{k \cdot (k-1) \cdot (k-2) \cdots 1}$

a) binomial_A
b) binomial_B
c) both
d) none
e) Don’t know

binomial_iterative.py

```python
def binomial_A(n, k):
    result = 1
    for i in range(k):
        result = result * (n - i) // (k - i)
    return result

def binomial_B(n, k):
    result = 1
    for i in range(k)[::-1]:
        result = result * (n - i) // (k - i)
    return result
```

Python shell

```
> binomial_A(5, 2)
| 8
> binomial_B(5, 2)
| 10
```

Recursively print all leaves of a tree

- Assume a recursively nested tuple represents a tree with strings as leaves

Python shell

```
> def print_leaves(tree):
      if isinstance(tree, str):
          print("Leaf:", tree)
      else:
          for child in tree:
              print_leaves(child)

> print_leaves(('a',('b','c')))
| Leaf: a
| Leaf: b
| Leaf: c
```

```
     /\
    /  \
  'a'   /\
       /  \
     'b'  'c'
```

Question – How many times is `print_leaves` function called in the example?

```
   /\
  /  \
'a'  /\
    /  \
  'b'  'c'
```

Python shell

```
> def print_leaves(tree):
      if isinstance(tree, str):
          print("Leaf:", tree)
      else:
          for child in tree:
              print_leaves(child)

> print_leaves(('a',('b','c')))
| Leaf: a
| Leaf: b
| Leaf: c
```

a) 3

b) 4

c) 5

d) 6

e) Don’t know

Python shell

```
> def collect_leaves_wrong(tree, leaves = set()):
      if isinstance(tree, str):
          leaves.add(tree)
      else:
          for child in tree:
              collect_leaves_wrong(child, leaves)
      return leaves

> def collect_leaves_right(tree, leaves = None):
      if leaves == None:
          leaves = set()
      if isinstance(tree, str):
          leaves.add(tree)
      else:
          for child in tree:
              collect_leaves_right(child, leaves)
      return leaves
```

```
> collect_leaves_wrong(('a',('b','c')))
| {'a', 'c', 'b'}
> collect_leaves_wrong(('d',('e','f')))
| {'b', 'e', 'a', 'f', 'c', 'd'}

> collect_leaves_right(('a',('b','c')))
| {'b', 'a', 'c'}
> collect_leaves_right(('d',('e','f')))
| {'f', 'd', 'e'}
```

Python shell

```
> def collect_leaves(tree):
      leaves = set()

      def traverse(tree):
          nonlocal leaves  # can be omitted
          if isinstance(tree, str):
              leaves.add(tree)
          else:
              for child in tree:
                  traverse(child)

      traverse(tree)
      return leaves

> collect_leaves(('a', ('b', 'c')))
| {'b', 'a', 'c'}
> collect_leaves(('d', ('e', 'f')))
| {'f', 'd', 'e'}
```

Maximum recursion depth ?

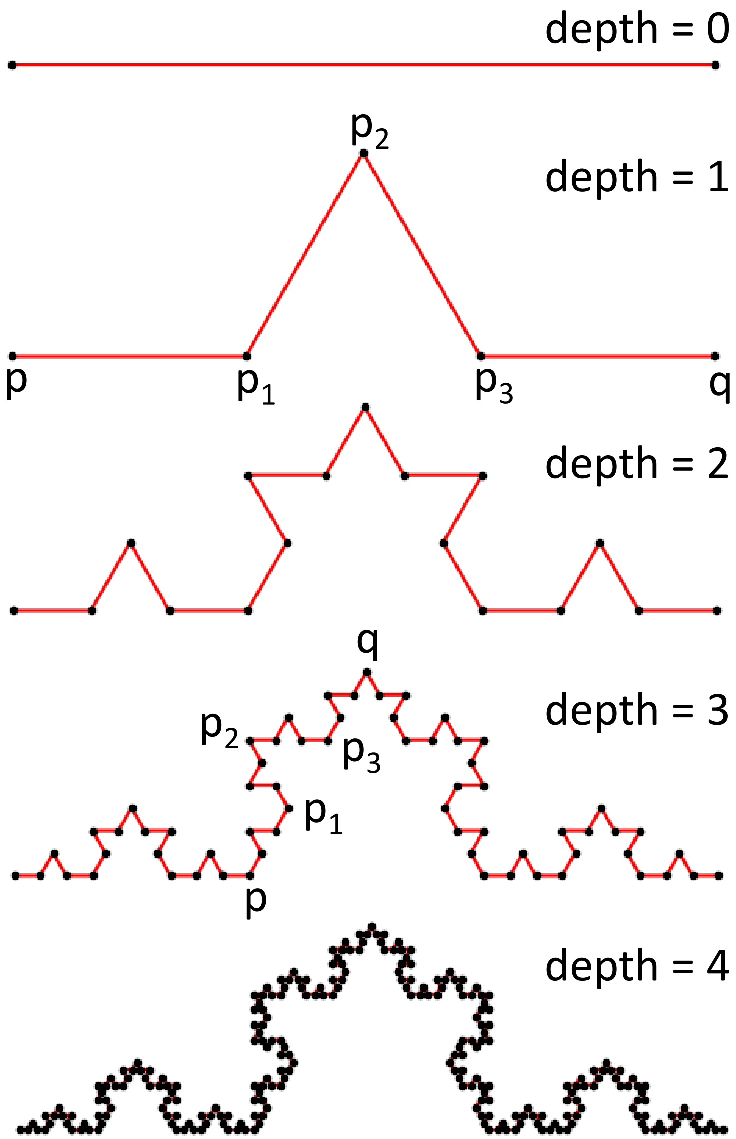
- Pythons maximum allowed recursion depth can be increased by

```
import sys
sys.setrecursionlimit(1500)
```

Python shell

```
> def f(x):
      print("#", x)
      f(x + 1)

> f(1)
| # 1
| # 2
| # 3
| ...
| # 975
| # 976
| # 977
| # 978
| RecursionError: maximum
  recursion depth exceeded
  while pickling an object
```

Koch Curves



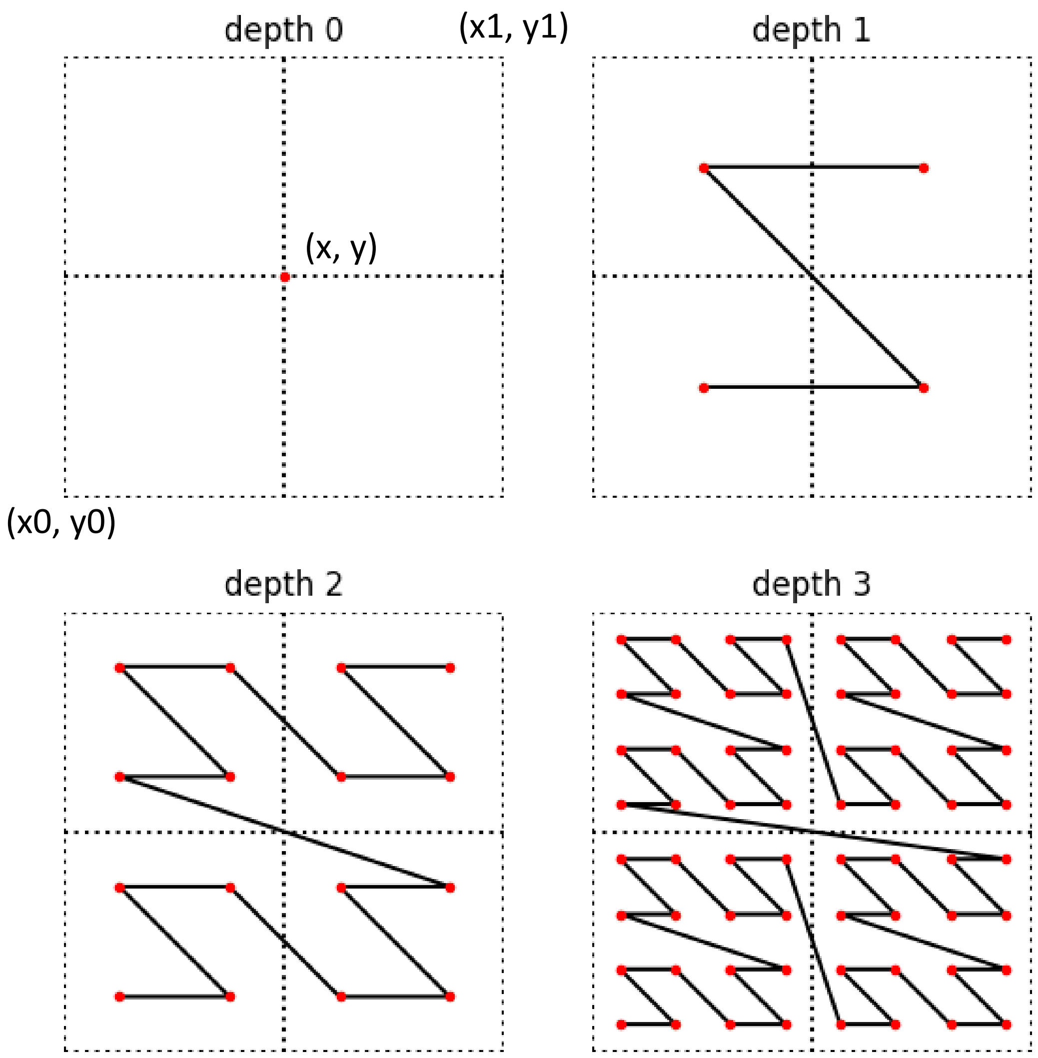
koch_curve.py

```
import matplotlib.pyplot as plt
from math import sqrt

def koch(p, q, depth=3):
    if depth == 0:
        return [p, q]

    (px, py), (qx, qy) = p, q
    dx, dy = qx - px, qy - py
    h = 1 / sqrt(12)
    p1 = px + dx / 3, py + dy / 3
    p2 = px + dx / 2 - h * dy, py + dy / 2 + h * dx
    p3 = px + dx * 2 / 3, py + dy * 2 / 3
    return (koch(p,  p1, depth - 1)[:-1]
          + koch(p1, p2, depth - 1)[:-1]
          + koch(p2, p3, depth - 1)[:-1]
          + koch(p3, q,  depth - 1))

points = koch((0, 0), (1, 0), depth=3)
X, Y = zip(*points)

plt.subplot(aspect='equal')
plt.plot(X, Y, 'r-')
plt.plot(X, Y, 'k.')
plt.show()
```

remove last point (equal to first point in next recursive call)

en.wikipedia.org/wiki/Koch_snowflake

Z-curves



z_curve.py

```python
import matplotlib.pyplot as plt

def z_curve(depth, x0=0, y0=0, x1=1, y1=1):
    x, y = (x0 + x1) / 2, (y0 + y1) / 2
    if depth == 0:
        return [(x, y)]
    return [
        *z_curve(depth - 1, x0, y0, x, y),
        *z_curve(depth - 1, x, y0, x1, y),
        *z_curve(depth - 1, x0, y, x, y1),
        *z_curve(depth - 1, x, y, x1, y1)
    ]

for depth in range(4):
    X, Y = zip(*z_curve(depth))
    plt.subplot(2, 2, 1 + depth, aspect='equal')
    plt.title(f'depth {depth}')
    plt.axis('off')
    plt.axis([0, 1, 0, 1])
    plt.plot(
        [0,1,1,0,0], [0,0,1,1,0], 'k:',  # dash box
        [0.5,0.5], [0,1], 'k:',  # dash vertical
        [0,1], [0.5,0.5], 'k:',  # dash horizontal
        X, Y, 'k-',  # Z-curve
        X, Y, 'r.',  # Z-curve points
    )
plt.show()
```

en.wikipedia.org/wiki/Z-order_curve