Recursion

Based on Materials by Thai Pangsakulyanont

Instructor: Paruj Ratanaworabhan

Recursion

- A technique in which a concept is defined, directly or indirectly, in terms of itself
- Allow repetition just like when we use loops for iteration

Iteration Example

```
def fac(n):
    if n == 0:
        return 1
    result = 1
    for i in range(1, n+1):
        result *= i
    return result

print(fac(5))
print(fac(0))
print(fac(10))
```

Recursive Factorial

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n-1)! \times n & \text{if } n > 0 \end{cases}$$

Factorial Example

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n-1)! \times n & \text{if } n > 0 \end{cases}$$

3!

Factorial Example

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n-1)! \times n & \text{if } n > 0 \end{cases}$$

$$3! = (3 - 1)! \times 3$$

Factorial Example

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n-1)! \times n & \text{if } n > 0 \end{cases}$$

$$\begin{aligned} 3! &= (3 - 1)! \times 3 \\ &= 2! \times 3 \end{aligned}$$

Factorial Example

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n-1)! \times n & \text{if } n > 0 \end{cases}$$

$$\begin{aligned} 3! &= (3 - 1)! \times 3 \\ &= 2! \times 3 \end{aligned}$$

We turn this problem into a
smaller problem of same kind.
This is called "decomposition."

Factorial Example

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n-1)! \times n & \text{if } n > 0 \end{cases}$$

3! = (3 - 1)! × 3

= <u>2!</u> × 3

2! = (2 - 1)! × 2

Factorial Example

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n-1)! \times n & \text{if } n > 0 \end{cases}$$

$$3! = (3 - 1)! \times 3$$
$$= \underline{2!} \times 3$$

$$2! = (2 - 1)! \times 2$$
$$= 1! \times 2$$

Factorial Example

$\begin{cases} 1 & \text{if } n = 0 \\ \ldots \times n & \text{if } n > 0 \end{cases}$

$1! = (1 - 1)! \times 1$

$\ldots \times 3$

$2! = (2 - 1)! \times 2$

$= \underline{1!} \times 2$

Factorial Example

$$\begin{cases} 1 & \text{if } n = 0 \\ \ldots \times n & \text{if } n > 0 \end{cases}$$

$\ldots \times 3$

$1! = (1 - 1)! \times 1$

$= 0! \times 1$

$\ldots = (2 - 1)! \times 2$

$= \underline{1!} \times 2$

Factorial … ple

$$0! = 1$$

$$1! = (1 - 1)! \times 1$$
$$= \underline{0!} \times 1$$

$$… = (2 - 1)! \times 2$$
$$= \underline{1!} \times 2$$

… × 3

Factorial ple

0! **= 1**

0

1! = (1 - 1

= 0! × 1

3

This is called the "base case" where the function does not call itself anymore.

(2 - 1)! × 2

= 1! × 2

Factorial

$0! = 1$

$1! = (1 - 1)! \times 1$

$= \underline{0!} \times 1$

$(2 - 1)! \times 2$

$= \underline{1!} \times 2$

Factorial Example

$\begin{cases} 1 & \text{if } n = 0 \\ \ldots \times n & \text{if } n > 0 \end{cases}$

$\times\ 3$

$$1! = (1 - 1)! \times 1$$
$$= \underline{0!} \times 1$$
$$= \underline{\mathbf{1}} \times 1 = \mathbf{1}$$

$$= (2 - 1)! \times 2$$
$$= \underline{1!} \times 2$$

Factorial Example

We use the result of smaller problem
to find the result of larger problem.
This is called "composition".

$1! = (1 - 1)! \times 1$

$= \underline{0!} \times 1$

$= \underline{\mathbf{1}} \times 1 = \mathbf{1}$

3

$2! = (2 - 1)! \times 2$

$= \underline{1!} \times 2$

Factorial Example

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n-1)! \times n & \text{if } n > 0 \end{cases}$$

$$\begin{aligned} 3! &= (3 - 1)! \times 3 \\ &= \underline{2!} \times 3 \end{aligned}$$

$$\begin{aligned} 2! &= (2 - 1)! \times 2 \\ &= \underline{1!} \times 2 \\ &= \underline{\mathbf{1}} \times 2 \mathbf{= 2} \end{aligned}$$

Factorial Example

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n-1)! \times n & \text{if } n > 0 \end{cases}$$

$$\begin{aligned} 3! &= (3 - 1)! \times 3 \\ &= \underline{2!} \times 3 \\ &= 2 \times 3 \\ &= \mathbf{6.} \end{aligned}$$

Factorial Example

```
def fac(n):
    if n == 0:
        return 1
    else:
        return fac(n - 1) * n

print fac(12)
```

Recursive Function

- A function that calls itself!

Count Down

```
countdown(5)
print "happy recursion day"
```

```
5
4
3
2
1
happy recursion day
```

Countdown Algorithm

- If I want to countdown from 0,
  then it's over! Don't do anything.
- If I want to countdown from N (> 0),
  then count N, and countdown from N-1.

Countdown Example

```
def countdown(n):
    if n == 0:
        return
    print n
    countdown(n - 1)

countdown(5)
print "happy recursion day"
```

Example:
Euclidean Algorithm

- Fast way to find a GCD of two numbers.

Example:
Euclidean Algorithm

If you have 2 numbers,
then you subtract the smaller number
from the larger number, the GCD of these
two number stays the same.

Example:
Euclidean Algorithm

68 119

GCD(68, 119) is 17

Example:
Euclidean Algorithm

68 119

68 51

GCD(68, 51) is still 17

Example:
Euclidean Algorithm

If you have 2 numbers,
then you subtract the smaller number
from the larger number, the GCD of these
two number stays the same.
Keep doing that until the two numbers equal each other, then that number is the GCD.

Example: Euclidean Algorithm

GCD(68, 119)

Example:
Euclidean Algorithm

GCD(68, 119) = GCD(68, 51)

Example:
Euclidean Algorithm

GCD(68, 119) = GCD(68, 51)

= **GCD(17, 51)**

Example: Euclidean Algorithm

$$
\begin{aligned}
\mathbf{GCD(68,\ 119)} &= \mathrm{GCD}(68,\ 51)\\
&= \mathrm{GCD}(17,\ 51)\\
&= \mathbf{GCD(17,\ 34)}
\end{aligned}
$$

Example: Euclidean Algorithm

$$
\begin{aligned}
\mathbf{GCD(68,\ 119)} &= \mathrm{GCD}(68,\ 51)\\
&= \mathrm{GCD}(17,\ 51)\\
&= \mathrm{GCD}(17,\ 34)\\
&= \mathbf{GCD(17,\ 17)}
\end{aligned}
$$

Example:
Euclidean Algorithm

GCD(68, 119) = GCD(68, 51)

= GCD(17, 51)

= GCD(17, 34)

= GCD(17, 17)

= **17**

Example: Euclidean Algorithm

```
def gcd(a, b):
    if a < b:
        return gcd(a, b - a)
    elif b < a:
        return gcd(a - b, b)
    else:
        return a

print gcd(68, 119)
```

Why Recursion?

- Sometimes it's easier to write and read code in recursive form.

Why Recursion?

- Sometimes it's easier to write and read code in recursive form.
- You can always convert an iterative algorithm to recursive and *vice versa*.

Why Recursion?

- Sometimes it's easier to write and read code in recursive form.
- You can always convert an iterative algorithm to recursive and *vice versa*.

  (does not mean it's easy)

Iterative Euclidean Algorithm

```
def gcd(a, b):
    while a != b:
        if a < b:
            b = b - a
        elif b < a:
            a = a - b
    return a

print gcd(68, 119)
```

Remember?

- Decomposition
- Base Case
- Composition

String Reversal

- Given a string, I want a reversed version of that string.

```
print reverse("reenigne")
          # => “engineer”
```

String Reversal

- Let’s think recursively!

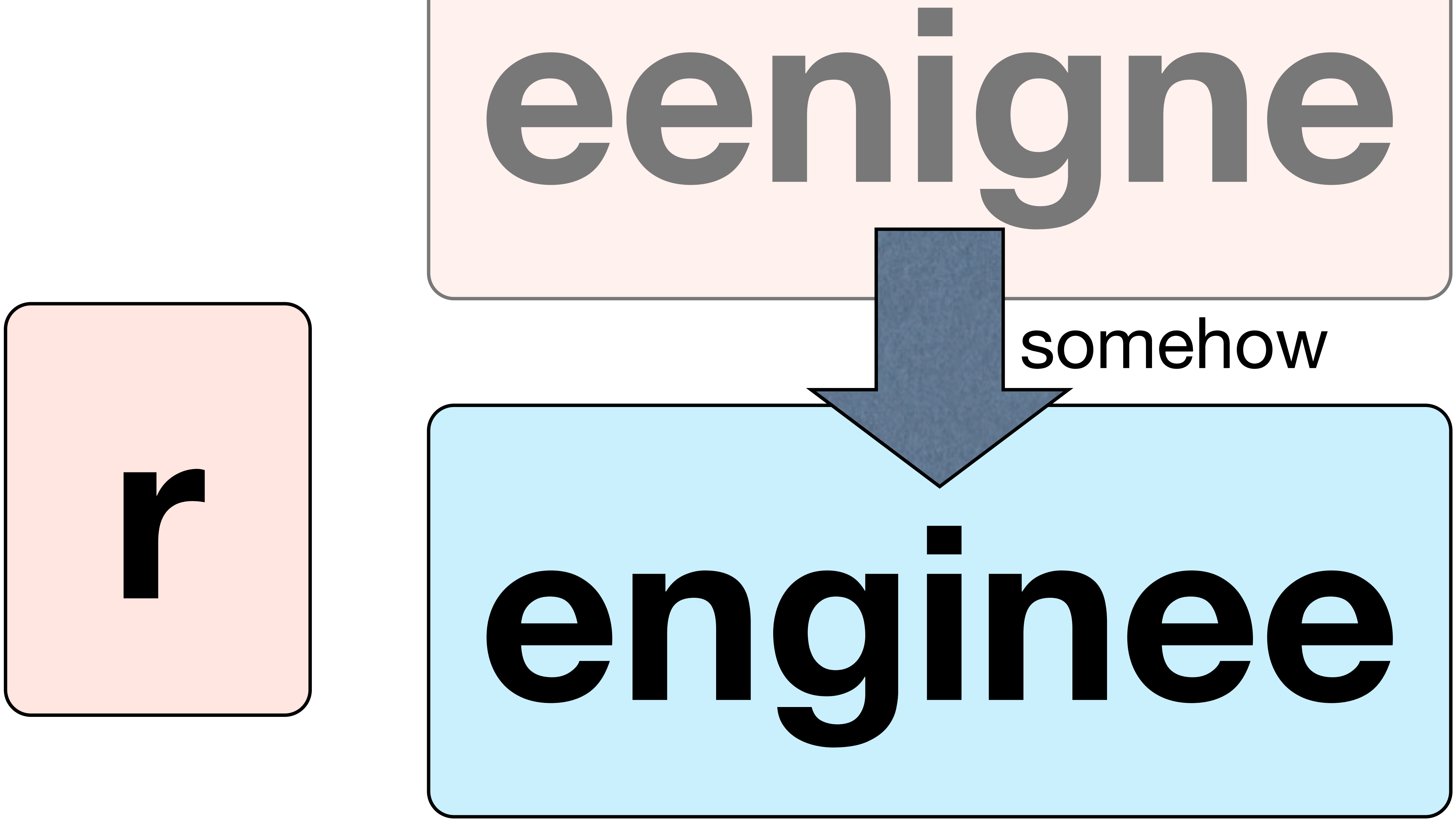
reenigne

Decomposition

r

eenigne

Recursive Case



Composition

enginee	r

Composition

engineer

eenigne

?

enginee

eenigne

e enigne

engine e

enginee

Base Case

- The reverse of an **empty string** is an **empty string**.

Recursive Algorithm

reverse(“Hello”)

Recursive Algorithm

reverse("Hello") = reverse("ello") + "H"

Recursive Algorithm

reverse(“Hello”) = reverse(“ello”) + “H”

reverse(“ello”) = reverse(“llo”) + “e”

Recursive Algorithm

reverse(“Hello”) = reverse(“ello”) + “H”

reverse(“ello”) = reverse(“llo”) + “e”

reverse(“llo”) = reverse(“lo”) + “l”

Recursive Algorithm

reverse(“Hello”) = reverse(“ello”) + “H”

reverse(“ello”) = reverse(“llo”) + “e”

reverse(“llo”) = reverse(“lo”) + “l”

reverse(“lo”) = reverse(“o”) + “l”

Recursive Algorithm

reverse(“Hello”) = reverse(“ello”) + “H”

reverse(“ello”) = reverse(“llo”) + “e”

reverse(“llo”) = reverse(“lo”) + “l”

reverse(“lo”) = reverse(“o”) + “l”

reverse(“o”) = reverse(“”) + “o”

Recursive Algorithm

reverse(“Hello”) = reverse(“ello”) + “H”

reverse(“ello”) = reverse(“llo”) + “e”

reverse(“llo”) = reverse(“lo”) + “l”

reverse(“lo”) = reverse(“o”) + “l”

reverse(“o”) = reverse(“”) + “o”

reverse(“”) = “”

Recursive Algorithm

reverse(“Hello”) = reverse(“ello”) + “H”

reverse(“ello”) = reverse(“llo”) + “e”

reverse(“llo”) = reverse(“lo”) + “l”

reverse(“lo”) = reverse(“o”) + “l”

reverse(“o”) = reverse(“”) + “o”
= “” + “o” = **“o”**

Recursive Algorithm

reverse("Hello") = reverse("ello") + "H"

reverse("ello") = reverse("llo") + "e"

reverse("llo") = reverse("lo") + "l"

reverse("lo") = reverse("o") + "l"

= "o" + "l" = **"ol"**

Recursive Algorithm

reverse("Hello") = reverse("ello") + "H"

reverse("ello") = reverse("llo") + "e"

reverse("llo") = reverse("lo") + "l"

= "ol" + "l" = **"oll"**

Recursive Algorithm

reverse(“Hello”) = reverse(“ello”) + “H”

reverse(“ello”) = <u>reverse(“llo”)</u> + “e”

= <u>“oll”</u> + “e” = **“olle”**

Recursive Algorithm

reverse("Hello") = <u>reverse("ello")</u> + "H"

= <u>"olle"</u> + "H"

= **"olleH"**

Reverse Example

```
def reverse(str):
    if str == "":
        return str    # or return “”
    else:
        return reverse(str[1:]) + str[0]

print reverse("reenigne")
```

Sum of Digits

```
print sum_digits(314159265) # => 36
```

Sum of Digits

```
print sum_digits(314159265) # => 36
```

- **NO LOOPS!**
- **NO STRINGS!**

Recursive Algorithm

- **Sum of digits of 0 is 0.**
- **Sum of digits of N > 0:**
  Find last digit + sum of digits except last.

Recursive Algorithm

- **Sum of digits of 0 is 0.**
- **Sum of digits of N > 0:**

  Find last digit + sum of digits except last.

N % 10

N // 10

Sum of Digits

```
def sum_digits(n):
    if n == 0:
        return 0
    else:
        return (n % 10) + sum_digits(n // 10)

print sum_digits(314159265)
```

Palindrome

Civic
Level
Madam
Malayalam
Radar
Reviver
Rotator
Terret

```
print is_palindrome("Refer")    # => True
print is_palindrome("Referrer") # => False
```

Recursive Algorithm

- Empty string **is** a palindrome.
- String with 1 character **is** a palindrome.
- String that has a **different** first character and last character **is not** a palindrome.

Recursive Algorithm

- Empty string **is** a palindrome.
- String with 1 character **is** a palindrome.
- String that has a **different** first character and last character **is not** a palindrome.
- String that has a **same** first and last character **is a** palindrome **only if** the string without first and last character is a palindrome.

```python
def is_palindrome(s):
    if len(s) < 2:
        return True
    if s[0].lower() != s[-1].lower():
        return False
    return is_palindrome(s[1:-1])

print is_palindrome("Refer")    # => True
print is_palindrome("Referrer") # => False
```

```python
def reverse(str):
    if str == "":
        return str
    else:
        return reverse(str[1:]) + str[0]


def is_palindrome(s):
    return reverse(s.lower()) == s.lower()

print is_palindrome("Refer")    # => True
print is_palindrome("Referrer") # => False
```

In Summary

- We have learned recursion as a programming technique that allows us to repeat operations
- Those repeatable operations must be defined in terms of themselves
- We have looked at some examples of recursion