Control structure: *Repetition - Part 1*

01204111 Computers and Programming

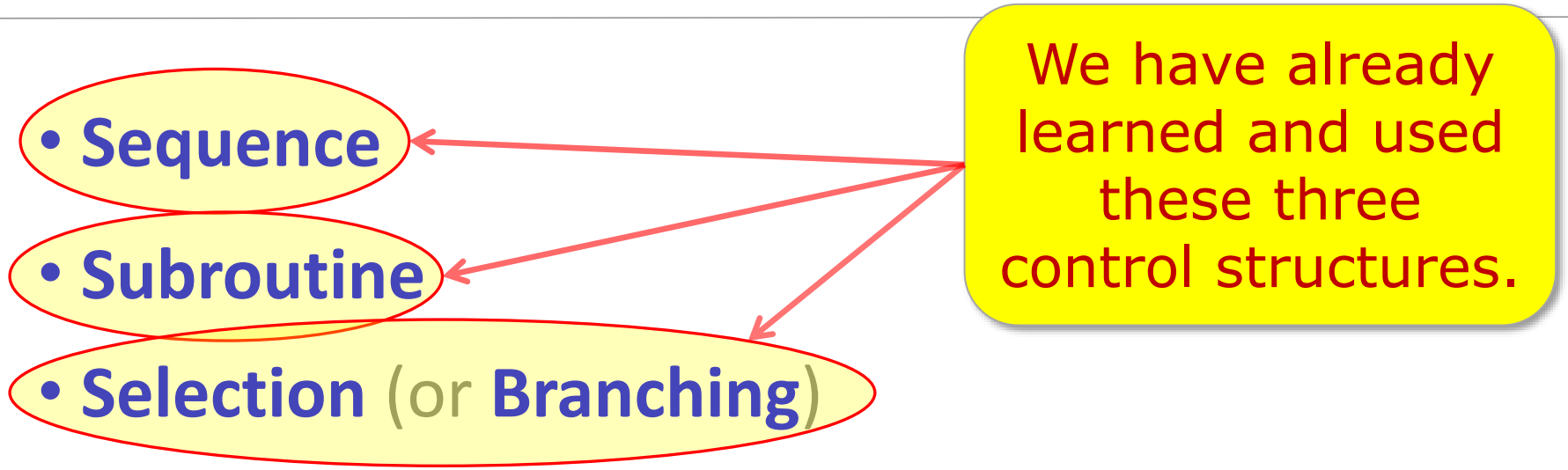
Chalermsak Chatdokmaiprai

Department of Computer Engineering
Kasetsart University

Outline

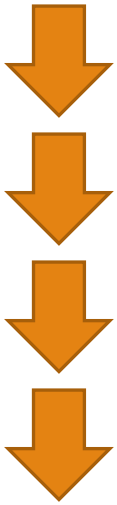
- **Repetition Control Flow**
  - Task : Hello world n times
  - Definite loops – the `for` statement
  - The `range()` function
- **Programming Examples**
  - A conversion table : Fahrenheit to Celsius
  - The factorial function
- **More About Strings**
  - String indexing
  - The `len()` function
  - Strings are immutable
  - For-loops with strings : String traversals
- **A Numerical Example : Average of Numbers**

Fundamental Flow Controls

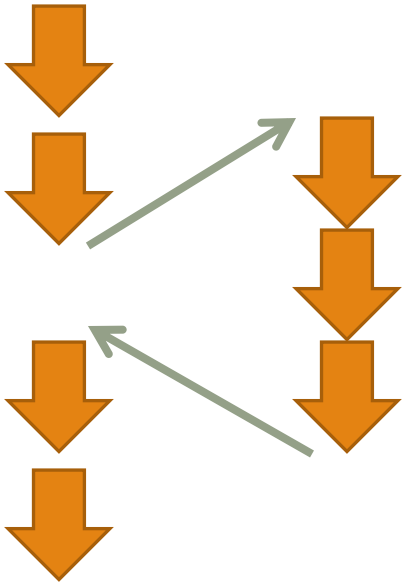


- **Sequence**
- **Subroutine**
- **Selection** (or **Branching**)
- **Repetition** (or **Iteration** or **Loop**)

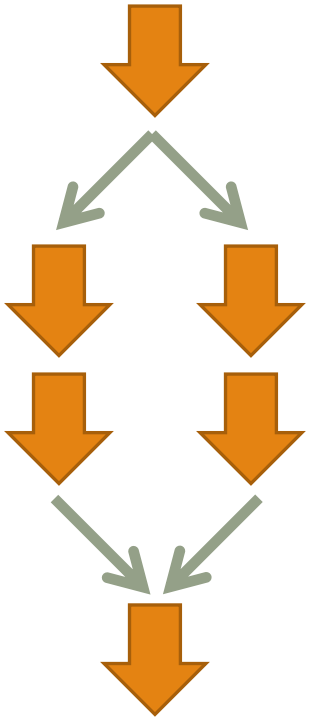
Schematic View of Flow Controls



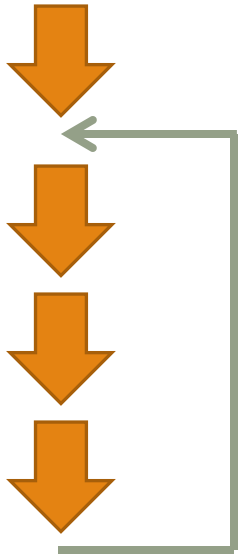
Sequence



Subroutine



Selection



Repetition

Repetition Flow Control

- Computers are often used to do *repetitive tasks* because humans don't like to do the same thing over and over again.
- In computer programs, **repetition flow control** is used to execute a group of instructions repeatedly.
- **Repetition flow control** is also called **iteration** or **loop**.
- In Python, repetition flow control can be expressed by a `for-statement` or a `while-statement` that allow us to execute a code block repeatedly.

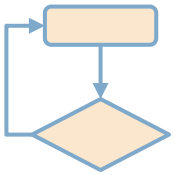
Task: Hello World *n* times

- Write a function **`hello(n)`** to write **Hello World!** **n** times, where **n** ≥ 0 is the input. After that, write **Goodbye!** once.

```
>>> hello(3)
Hello World!
Hello World!
Hello World!
Goodbye!
>>> hello(0)
Goodbye!
>>> hello(1)
Hello World!
Goodbye!
```

```
>>> hello(10)
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Goodbye!
```

The function hello(n) – Steps



- **hello(n):**
  - receive *n* as its parameter.
  - repeat *n* times:
    - write 'Hello World!'
  - write 'Goodbye!'

How can we do this repetition in Python?

```
def hello(n):
    for i in range(n):
        print('Hello World!')
    print('Goodbye!')
```

And it's all done !

Definite Loops: the *for* Statement

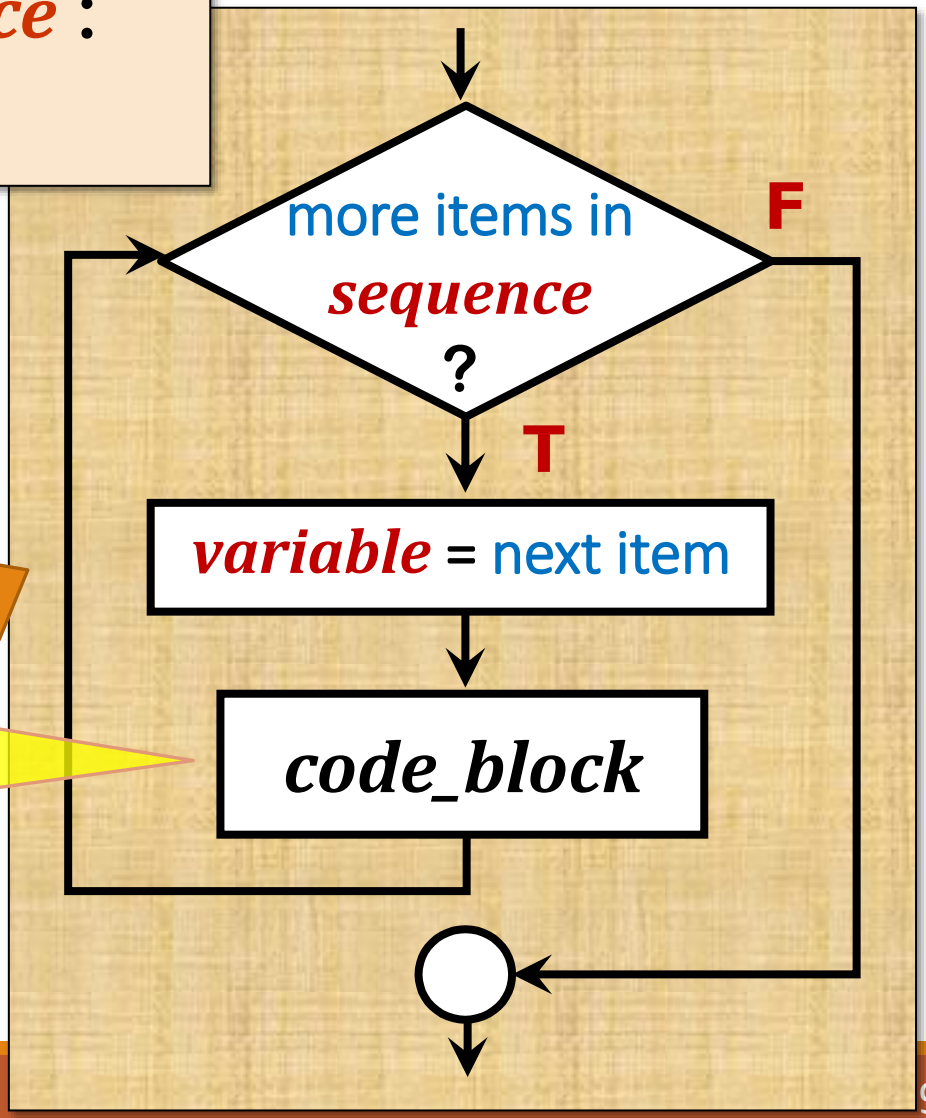
Python Syntax

```
for variable in sequence :
    code_block
```

- ***variable*** after **`for`** is called the *loop index*. It takes on each successive value in ***sequence*** in the order they appear in the sequence.
- The ***sequence*** can be one of the Python sequence objects such as *a string*, *a list*, *a tuple*, or *a range of integers* from the built-in function **`range()`**.

How the *for* statement works

```
for variable in sequence:
    code_block
```



The number of times the ***code_block*** is executed is precisely the number of items in the ***sequence***.

Hands-on Example

```
>>> my_string = 'python'
>>> k = 0
>>> for c in my_string:
        k = k + 1
        print(f'round {k} : c is {c}')

round 1 : c is p
round 2 : c is y
round 3 : c is t
round 4 : c is h
round 5 : c is o
round 6 : c is n
```

The variable **c** is the loop index.

The variable **my_string** produces the sequence **'p', 'y', 't', 'h', 'o', 'n'**.

Output

The **code block** to be repeated for each value of **c**.

Hands-on Example

The variable `i` is the loop index.

This list object produces the sequence 10, -3.5, 'py', True

```
>>> k = 0
>>> for i in [10,-3.5,'py',True]:
        k = k + 1
        print(f'round {k} : i is {i}')

round 1 : i is 10
round 2 : i is -3.5
round 3 : i is py
round 4 : i is True
```

Output

The code block to be repeated for each value of `i`.

Don't worry about list objects now.
We'll study them in detail in the near future.

Hands-on Example

```
>>> k = 0
>>> for i in range(4):
        k = k + 1
        print(f'round {k} : i is {i}')

round 1 : i is 0
round 2 : i is 1
round 3 : i is 2
round 4 : i is 3
```

The variable `i` is the loop index.

The object `range(4)` generates the sequence 0, 1, 2, 3

Output

The code block to be repeated for each value of `i`.

The `range()` function

Python Syntax

```
range(start, stop, step)
```

- In its most general form, the `range()` function takes three *integer* arguments: *start*, *stop*, and *step*.
- `range(start, stop, step)` produces the sequence of integers:

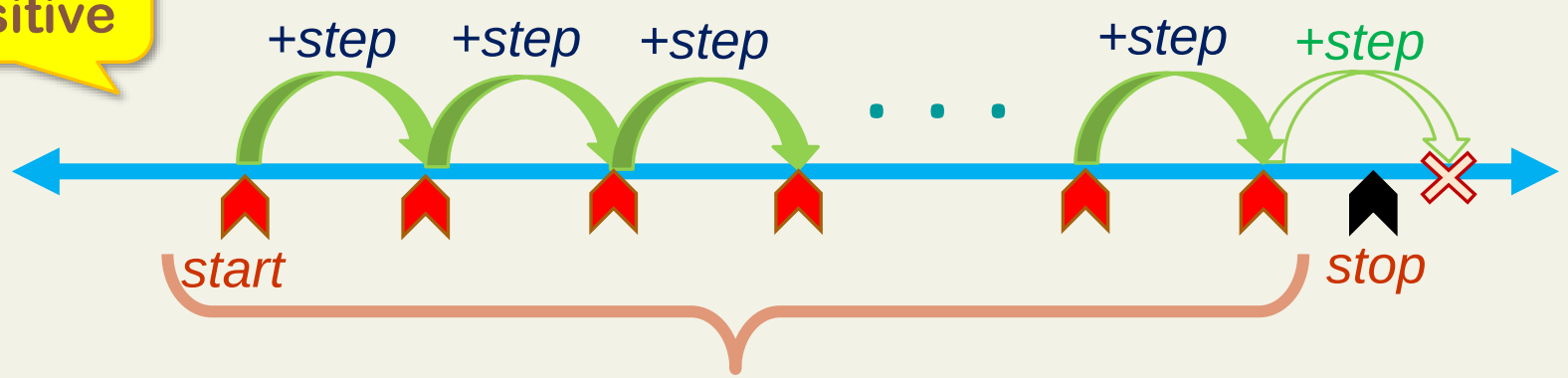
  start, *start + step*, *start + 2*step*, *start + 3*step*, …

If *step* is positive, the last element is the largest integer less than *stop*.

If *step* is negative, the last element is the smallest integer greater than *stop*.

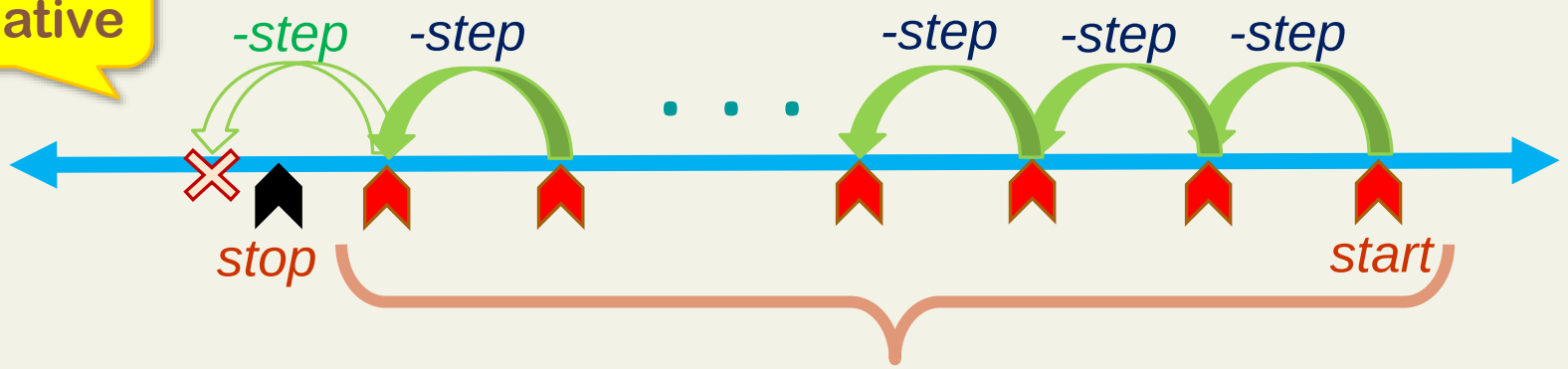
range(start, stop, step)

When step is positive



values in the sequence
(in left-to-right order)

When step is negative



values in the sequence
(in right-to-left order)

Hands-on Example

```
>>> type(range(-4,14,3))
<class 'range'>
>>> range(-4,14,3)
range(-4, 14, 3)
>>> list(range(-4,14,3))
[-4, -1, 2, 5, 8, 11]
>>> list(range(14,-4,-3))
[14, 11, 8, 5, 2, -1]
>>> list(range(5,3,1))
[]
>>> list(range(3,5,-1))
[]
>>> list(range(3,3,2))
[]
>>> list(range(3,3,-2))
[]
>>>
```

To see the type of object **range()**

This doesn't show the sequence generated by **range()**.

Use the built-in function **list()** to show the sequence.

Try a negative step.

produces *the empty sequence* because *step* is positive and *start* is not less than *stop*.

produces *the empty sequence* because *step* is negative and *start* is not greater than *stop*.

Notice that in all cases, the *stop* value will never appear in the generated sequence.

Hands-on Example : *start* and *step* can be omitted

```
>>> list(range(3,8,1))
[3, 4, 5, 6, 7]
>>> list(range(3,8))
[3, 4, 5, 6, 7]
>>> list(range(0,5))
[0, 1, 2, 3, 4]
>>> list(range(5))
[0, 1, 2, 3, 4]
>>> for i in range(4):
        print('i is', i)

i is 0
i is 1
i is 2
i is 3
```

If *step* is omitted, the *default step* is 1.

If *start* is also omitted, the *default start* is 0.

So **range(4)** is the same as **range(0,4)** and **range(0,4,1)**.

Programming Example on For-Loops: A Conversion Table

Task: Print a Fahrenheit-to-Celsius Table

- Write a function to print a Fahrenheit-to-Celsius conversion table from 212 F to 32 F, decremented in each step by 20 F.

```
>>> fah_to_cel()
  Fahrenheit     Celsius
  ----------     -------
         212       100.0
         192        88.9
         172        77.8
         152        66.7
         132        55.6
         112        44.4
          92        33.3
          72        22.2
          52        11.1
          32         0.0
  ----------     -------
```

Print a Fahrenheit-to-Celsius Table - Ideas

- The formula to convert fahrenheit to celsius:

```
celsius = (5/9)*(fahrenheit-32)
```

- We'll write a function **`fah_to_cel()`** to do the task.
- We'll use a ***for****-loop* in which the *loop index* holds *the fahrenheit values*.
- The ***for****-loop* will iterate over a sequence from 212 downto 32 generated by **`range()`**.

First, let's experiment with the range of Fahrenheit values:

```
>>> list(range(212,32,-20))
[212, 192, 172, 152, 132, 112, 92, 72, 52]
>>> list(range(212,31,-20))
[212, 192, 172, 152, 132, 112, 92, 72, 52, 32]
>>>
```

32 is missing!

We should use *one less than 32* so that 32 can be included in the sequence

So now we've got a correct sequence running from 212 downto 32.

The function fah_to_cel()

```
def fah_to_cel():
    print(f"{'Fahrenheit':>12}{'Celsius':>12}")
    print(f"{'----------':>12}{'-------':>12}")

    for fah in range(212,31,-20):
        cel = (5/9)*(fah-32)
        print(f"{fah:12}{cel:12.1f}")

    print(f"{'----------':>12}{'-------':>12}")
```

```
>>> fah_to_cel()
  Fahrenheit     Celsius
  ----------     -------
         212       100.0
         192        88.9
         172        77.8
         152        66.7
         132        55.6
         112        44.4
          92        33.3
          72        22.2
          52        11.1
          32         0.0
  ----------     -------
```

Do those print statements above still puzzle you?

If so, let's do some experiments to demystify them:

```
def test_print():
    print('12345678901234567890')
    print(f"{'python':>10}{'idle':>10}")
    print(f"{'python':<10}{'idle':<10}")
    print(f"{'python':^10}{'idle':^10}")
    cel = 32/3
    print('12345678901234567890')
    print(f"{cel:>10.1f}{cel:>10.3f}")
    print(f"{cel:<10.1f}{cel:<10.3f}")
    print(f"{cel:^10.1f}{cel:^10.3f}")
    print('12345678901234567890')
```

```
>>> test_print()
12345678901234567890
    python      idle
python    idle
  python      idle
12345678901234567890
      10.7    10.667
10.7      10.667
   10.7     10.667
12345678901234567890
>>>
```

Left-justification is the default for strings, so the symbol **<** can be omitted here.

Right-justification is the default for numbers, so the symbol **>** can be omitted here.

Next: Make `fah_to_cel()` more general

- Let's add three parameters: start, end, and step to control the Fahrenheit values to be printed.

```
>>> fah_to_cel(step=-20,start=100,end=0)
  Fahrenheit     Celsius
  ----------     -------
         100        37.8
          80        26.7
          60        15.6
          40         4.4
          20        -6.7
  ----------     -------
>>>
```

```
>>> fah_to_cel(32,100,20)
  Fahrenheit     Celsius
  ----------     -------
          32         0.0
          52        11.1
          72        22.2
          92        33.3
  ----------     -------
>>> fah_to_cel(100,32,-20)
  Fahrenheit     Celsius
  ----------     -------
         100        37.8
          80        26.7
          60        15.6
          40         4.4
  ----------     -------
>>>
```

The generalized fah_to_cel()

```
def fah_to_cel(start, end, step):
    print(f"{'Fahrenheit':>12}{'Celsius':>12}")
    print(f"{'----------':>12}{'-------':>12}")

    for fah in range(start, end, step):
        cel = (5/9)*(fah-32)
        print(f"{fah:12}{cel:12.1f}")

    print(f"{'----------':>12}{'-------':>12}")
```

Programming Example on For-Loops: The Factorial Function

Task: Computing the factorial

- Suppose you have *five pens* of different colors to give to *five kids*. How many ways are there to give those five pens to those kids?
  - **Answer:** 5*4*3*2*1 = 120 ways

This value is called the factorial of 5, or simply 5!

- More generally, the **factorial** is defined as a function of nonnegative integers (0, 1, 2, …) such that:

n! = n(n-1)(n-2)…(2)(1) when n > 0,

and 0! = 1

Task: Computing the factorial

- Let's write a Python function **`factorial(n)`** to compute and return the value of **`n!`**

```
>>> factorial(5)
120
>>> factorial(3)
6
>>> factorial(1)
1
>>> factorial(0)
1
>>> factorial(10)
3628800
>>>
```

factorial(n): An Accumulating Algorithm

- How do we calculate 3! and 5! ?
  - 3! = 3*2*1 = 6
  - 5! = 5*4*3*2*1 = 120
- But the function **`factorial(n)`** must work for every value of n, so we'll devise *an accumulating algorithm* that works for every value of n.

factorial(n): An Accumulating Algorithm

- How can we compute 4! by accumulating results?

```
Start at 1
Take that 1,   then   1*4 = 4
Take that 4,   then   4*3 = 12
Take that 12,  then  12*2 = 24
Done!
```

Let's use a variable **result** to hold the accumulating result.

```
result = 1
result = result*4
result = result*3
result = result*2
return result
```

Then, translate our algorithm into Python statements

factorial(n): An Accumulating Algorithm

```
result = 1
result = result*4
result = result*3
result = result*2
return result
```

Notice that for **4!**
this calculation is repeated
3 times through the sequence
4, 3, 2

```
result = 1
result = result*n
result = result*(n-1)
result = result*(n-2)
...
result = result*2
return result
```

Therefore for **n!**
this calculation is repeated
n-1 times through the sequence
n, n-1, n-2 …, 3, 2

This repetition is exactly the for-loop:

```
for i in range(n,1,-1):
    result = result*i
```

factorial(n): from algorithm to code

```
result = 1
result = result*n
result = result*(n-1)
result = result*(n-2)
...
result = result*2
return result
```


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

Wait a minute!

Does it work when n = 0 or 1 ?

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

When n = 0 or 1, the range() in the for-statement becomes range(0,1,-1) or range(1,1,-1), respectively.

What are range(0,1,-1) and range(1,1,-1)?

Can you explain why?

```
>>> list(range(0,1,-1))
[]
>>> list(range(1,1,-1))
[]
```

And this is what happens when looping through *the empty sequence*.

```
>>> for i in []:
        print("Yappadapadoo")

>>>
```

Statements inside the loop don't get executed at all.

Now, can you explain why our `factorial(n)` returns correct results when n = 0 or 1 ?

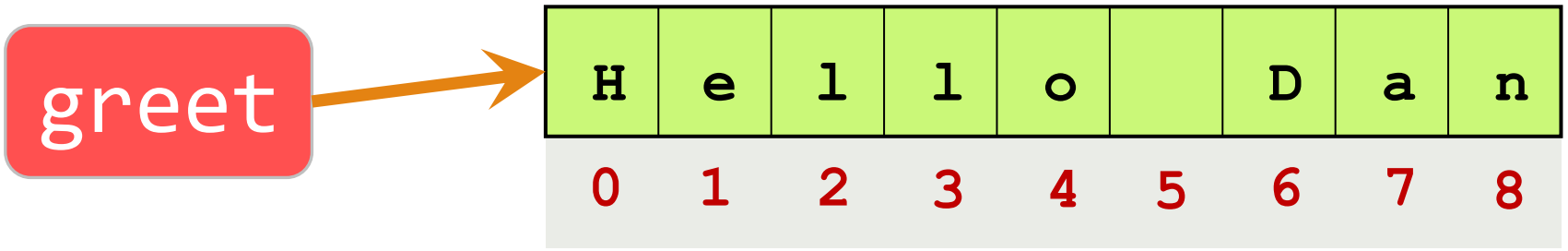
More About
Strings

String Indexing:
Accessing Characters in a String

- A Python string is an object of type `str`, which represents a sequence of characters.
- We can access each individual character in a string with *an expression (type `int`) in the bracket operator* [ ]. The expression in the bracket is called a string index.

String Indexing:
Accessing Characters in a String

- The indexes of characters in a string are numbered from the left, starting with 0.

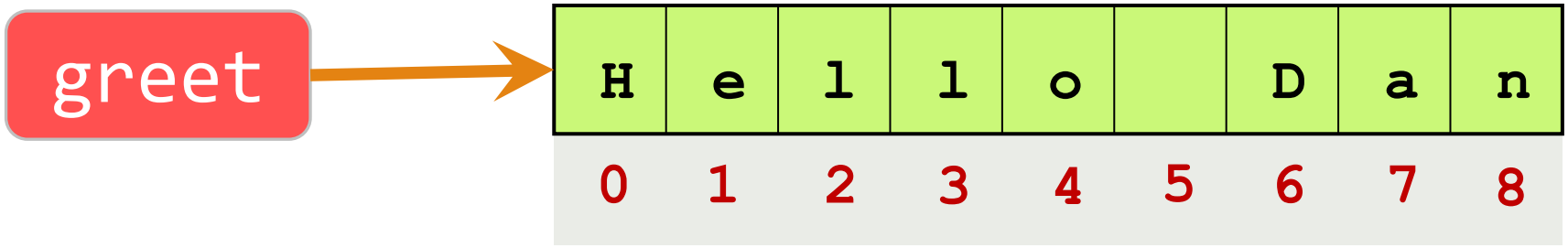



```
>>> greet = 'Hello Dan'
>>> greet
'Hello Dan'
>>> greet[0]
'H'
>>> greet[4]
'o'
>>> print(greet[1], greet[4]+greet[8])
e on
```

Both of them are of type str.

```
>>> type(greet)
<class 'str'>
>>> type(greet[4])
<class 'str'>
```

String Indexing:
Accessing Characters Within a String




```
>>> greet = 'Hello Dan'
>>> letter = greet[1]
>>> letter
'e'
>>> k = 1
>>> greet[k]
'e'
>>> greet[k+5]
'D'
>>> greet[2*k+5]
'a'
>>> greet[1.5]
TypeError: string indexes must be integers
```

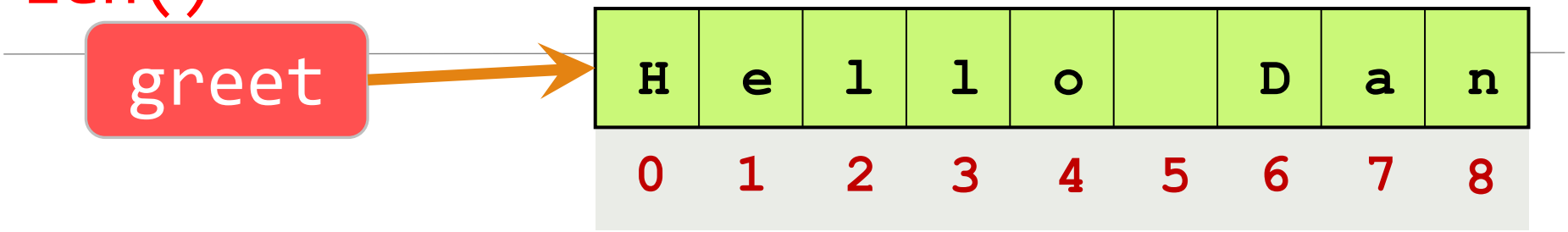
greet[1] is a string object so it can be assigned to a variable.

String indexes can be any expressions of type int.

Indexing also works on string literals.

```
>>> 'python'[0]
'p'
>>> "python"[3]
'h'
>>> 'python'[k+2]
'h'
```

Length of a String: The Built-in Function len()




```
>>> greet = 'Hello Dan'
>>> len(greet)
9
>>> len("I'm happy.")
10
>>> greet + 'ny!'
'Hello Danny!'
>>> len(greet + 'ny!')
12
```

The built-in function len() returns the length of its string argument.

The last index of greet is 8, not 9.

```
>>> length = len(greet)
>>> length
9
>>> lastchar = greet[length]
IndexError: string index out of range
>>> lastchar = greet[length-1]
>>> lastchar
'n'
```

Correct index of the last character.

Indexing from the right side: Negative indexes

greet

H	e	l	l	o		D	a	n
0	1	2	3	4	5	6	7	8
-9	-8	-7	-6	-5	-4	-3	-2	-1

You can use negative indexes to index backward from the end of the string.

Notice the positive indexes that refer to the same positions as their negative counterparts.

```
>>> greet = 'Hello Dan'
>>> greet[-1]
'n'
>>> greet[-2]
'a'
>>> greet[-9]
'H'
>>> greet[-len(greet)]
'H'
```

last character

second-last character

first character

Also first character since `len(greet)` is 9.

```
>>> length = len(greet)
>>> greet[length-1]
'n'
>>> greet[length-2]
'a'
>>> greet[0]
'H'
>>> greet[length-len(greet)]
'H'
```

last character

second-last character

first character

Also first character since `length-len(greet)` is 0.

String Objects Are Immutable.

- Python string objects are immutable. That means the characters within a string object *cannot* be changed.

```
>>> greet = 'Hello Dan'
>>> greet[6] = 'J'
TypeError: 'str' object does not support item assignment
>>> 'Hello Dan'[6] = 'J'
TypeError: 'str' object does not support item assignment
>>> greet
'Hello Dan'
>>> greet = greet + 'ny'
>>> greet
'Hello Danny'
```

A new string object 'Hello Dan' is created and then assigned to the variable greet.

Expect to change Dan to Jan

This doesn't work either.

greet and the string 'Hello Dan' remain unchanged.

A new string object 'Hello Danny' is created and then assigned to the variable greet.

Note that the variable **greet** has changed its binding twice by assignment, but the two string objects, 'Hello Dan' and 'Hello Danny', can never be changed.

For-Loops with Strings

String Traversal

- Lots of computations involve processing a string one character at a time in this pattern:

1. Get a string
2. Repeat from the beginning until the end of the string:
   - *Get next character*
   - *Do something with the character*

- This pattern of processing is called a ***traversal***.

- In Python, one way to implement a string traversal is by using a **for** loop.

Traversal Example: Spreading Out a String

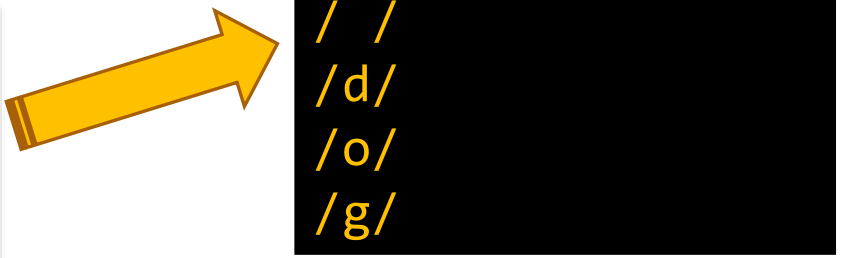
- We want to write a program that *prints each character in a string, one per line, enclosed in ()*.

Without using a loop, we may write a python program to traverse the string assigned to a variable text like this:

```
text = 'a dog'
c = text[0]; print(f'/{c}/')
c = text[1]; print(f'/{c}/')
c = text[2]; print(f'/{c}/')
c = text[3]; print(f'/{c}/')
c = text[4]; print(f'/{c}/')
```

The symbol ; allows two or more statements on the same line

Output



Traversal Example: Spreading Out a String

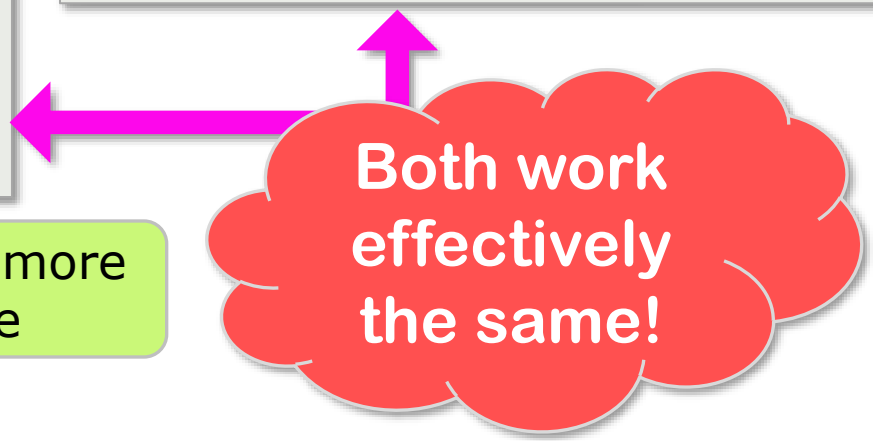
Without using a loop, we may write a python program to traverse the string assigned to a variable text like this:

```
text = 'a dog'
c = text[0]; print(f'/{c}/')
c = text[1]; print(f'/{c}/')
c = text[2]; print(f'/{c}/')
c = text[3]; print(f'/{c}/')
c = text[4]; print(f'/{c}/')
```

The symbol ; allows two or more statements on the same line

But since a string is a *sequence type* like a range() object, we can use **for** statements with strings in a similar way:

```
text = 'a dog'
for c in text:
    print(f'/{c}/')
```



Spreading Out a String : Generalization

```
text = 'a dog'
for c in text:
    print(f'/{c}/')
```

Test it.

```
>>> text = 'a dog'
>>> for c in text:
        print(f'/{c}/')

/a/
/ /
/d/
/o/
/g/
```

Generalize and encapsulate it into a function.

```
def spread_str(text):
    """print text,
    one char per line within ()"""
    for c in text:
        print(f'/{c}/')
```

Test it again.

```
>>> spread_str('a')
/a/
>>> spread_str('')
>>>
>>> dino = 'Rex'
>>> spread_str("T'" + dino)
/T/
/'/
/R/
/e/
/x/
```

No output Why?

String Traversal: Looping Over String Indexes

These two blocks of code work effectively the same.

```
text = 'a dog'
c = text[0]; print(f'/{c}/')
c = text[1]; print(f'/{c}/')
c = text[2]; print(f'/{c}/')
c = text[3]; print(f'/{c}/')
c = text[4]; print(f'/{c}/')
```

```
text = 'a dog'
print(f'/{text[0]}/')
print(f'/{text[1]}/')
print(f'/{text[2]}/')
print(f'/{text[3]}/')
print(f'/{text[4]}/')
```

Thus another version of spread_str(text)

```
def spread_str(text):
    """print text,
    one char per line within ()"""
    for i in range(len(text)):
        print(f'/{text[i]}/')
```

This suggests that we can also do this task by looping over the string indexes.

Recall that the indexes of any string **s** are in the range **0, 1, 2, …, len(s)-1**

```
>>> s = 'a dog'
>>> list(range(len(s)))
[0, 1, 2, 3, 4]
```

Two equivalent implementations

```
def spread_str(text):
    """print text, one char per line within ()"""
    for c in text:
        print(f'/{c}/')
```

Which one do you prefer?

```
def spread_str(text):
    """print text, one char per line within ()"""
    for i in range(len(text)):
        print(f'/{text[i]}/')
```

Traversal Example: Counting a Character

- We want to write a program that *counts the number of times a character appears in a string*.

Suppose we want to count the number of 'e' in a string 'pete', we may write a program like this:

```
text = 'pete'
count = 0
c=text[0]; if c=='e': count = count+1
c=text[1]; if c=='e': count = count+1
c=text[2]; if c=='e': count = count+1
c=text[3]; if c=='e': count = count+1
print(count)
```

This box *traverses* the string from left to right. If the current character is 'e', the variable count will be incremented by 1.

This repetition is exactly the **for** loop:

```
for c in text:
    if c == 'e':
        count = count+1
```

Counting a Character : Generalization

```
text = 'pete'
count = 0
for c in text:
    if c == 'e':
        count = count+1
print(count)
```

Generalize and encapsulate it into a function.

```
def count_char(char, text):
    """counts the no. of times
    'char' appears in 'text'"""
    count = 0
    for c in text:
        if c == char:
            count = count+1
    return count
```

Test it.

```
>>> count_char('a', 'anaconda')
3
>>> count_char('x', 'python')
0
>>> count_char(' ', 'I am happy')
2
>>> count_char(text='python', char='y')
1
```

Counting a Character : An Alternative

Alternatively, we may *loop over string indexes* with the same result.

```
def count_char(char, text):
    """counts the no. of times
    'char' appears in 'text'"""
    count = 0
    for c in text:
        if c == char:
            count = count+1
    return count
```

```
def count_char(char, text): #version 2
    """counts the no. of times
    'char' appears in 'text'"""
    count = 0
    for i in range(len(text)):
        if text[i] == char:
            count = count+1
    return count
```

Which one, do you think, is simpler?

One More Numerical Example

Task: Average of Numbers

- Write a program to read **n** numbers and calculate their **average**.
- The inputs of the program are the values of **n** and all of the **n** numbers. **n** is a positive integer and each number is a `float` number.

Sample Run

```
How many numbers? 4
Enter number #1: 12
Enter number #2: 11.5
Enter number #3: 13
Enter number #4: 10.5
The average of 4 number(s) is 11.75
```

Sample Run

```
How many numbers? 0
Nothing to do. Bye!
```

Average of Numbers – Topmost Steps

- Algorithm of the Main Routine:
  - *Read the value of n, making sure that n > 0*
  - *Read each of the n numbers and calculate the average.*
  - *Output the average.*

Translate it into Python

```
import sys
# ---- main ---- #
n = int(input('How many numbers? '))
if n <= 0:
    print('Nothing to do. Bye!')
    sys.exit()
avg = average(n)
print(f'The average of {n} number(s) is {avg}')
```

We'll write the function `average()` to do the read/calculate task.

The function average(n)

- Algorithm of average(n):
  - *Gets the count of numbers n as its parameter.*
  - *Reads each of the n numbers and calculate the sum:*
    `sum = number1 + number2 + … + numbern`
  - *Calculates the average with:*
    `average = sum / n`
  - *Returns the average.*

Translate it into Python

```
def average(n):
    ?
    return sum/n
```

The hard part to be figured out next

A new variable **average** is not needed

Accumulating Algorithm Once Again

- *Reads each of the n numbers and calculate the sum:*

$$sum = number_1 + number_2 + \ldots + number_n$$

- sum = 0
- Read $number_1$; sum = sum + $number_1$
- Read $number_2$; sum = sum + $number_2$
- ...
- Read $number_n$; sum = sum + $number_n$

Transform the sum formula into an accumulating algorithm.

- sum = 0
- Repeat n times:
  - Read a new number into the variable number
  - sum = sum + number

Then make it a loop of n iterations.

Notice that only one variable number is enough for all numbers.

Then translate it into Python

```
sum = 0
for i in range(n):
    number = float(input(f'Enter number #{i+1}: '))
    sum = sum + number
```

average() - Complete

- Algorithm of average():
  - *Gets the count of numbers n as its parameter.*
  - *Reads each of the n numbers and calculate the sum:*
    sum = $number_1$ + $number_2$ + … + $number_n$
  - *Calculates the average with:*
    average = sum / n
  - *Returns the average.*

Translate it into Python

```
def average(n):
    sum = 0
    for i in range(n):
        number = float(input(f'Enter number #{i+1}: '))
        sum = sum + number
    return sum/n
```

Average of Numbers – Complete Program

```
import sys

def average(n):
    sum = 0
    for i in range(n):
        number = float(input(f'Enter number #{i+1}: '))
        sum = sum + number
    return sum/n


# ---- main ---- #
n = int(input('How many numbers? '))
if n <= 0:
    print('Nothing to do. Bye!')
    sys.exit()
avg = average(n)
print(f'The average of {n} number(s) is {avg}')
```

```
How many numbers? 4
Enter number #1: 12
Enter number #2: 11.5
Enter number #3: 13
Enter number #4: 10.5
The average of 4 number(s) is 11.75
```

```
How many numbers? 0
Nothing to do. Bye!
```

Conclusion

- In computer programs, repetition flow control (also called iteration or loop) is used to execute a group of instructions repeatedly.
- In Python, a repetition flow control can be expressed by a `for`-statement which allows us to execute a code block a definite number of times.
- We can use a `for`-statement to iterate over any of the Python sequence objects such as a range of integers, a string, a list, etc.

References

- Think Python
  - http://greenteapress.com/thinkpython2/thinkpython2.pdf
- Official reference on the *for* statement:
  - https://docs.python.org/3/reference/compound_stmts.html#the-for-statement
- Good tutorials for *for-loops, range(), and string indexing*:
  - https://docs.python.org/3/tutorial/controlflow.html#for-statements
  - https://docs.python.org/3/tutorial/controlflow.html#the-range-function
  - https://docs.python.org/3/tutorial/introduction.html#strings
  - http://www.python-course.eu/python3_for_loop.php

Syntax Summary I

for statement

```
for variable in sequence:
    code_block
```

sequence may be any Python sequence object such as a string, a range of integers, a list, etc.

The range() function

`range(start, stop)`

exactly the same as `range(start, stop, 1)`

`range(stop)`

exactly the same as `range(0, stop, 1)`

`range(start, stop, step)`

generates successive integers:
*start, start + step, start + 2*step, …*

If *step* is positive, the last element is the largest integer less than *stop*.

If *step* is negative, the last element is the smallest integer greater than *stop*.

Syntax Summary II

The list() function

```
list(range(start, stop, step))
list(range(start, stop))
list(range(stop))
```

Returns a list object defined by the successive integers generated by `range()`

The len() function

```
len(string)
```

Returns the length of its string argument

string indexing

`str[0]`	first character
`str[i]`	(i^{th}-1) character
`str[len(str)-1]`	last character
`str[-1]`	last character
`str[-2]`	second-last character
`str[-len(str)]`	first character

Major Revision History

- September, 2017 – Chalermsak Chatdokmaiprai
  - First release
- March 16, 2019 – Chalermsak Chatdokmaiprai
  - Fixed minor typos
- February 2, 2020 – Chalermsak Chatdokmaiprai
  - improved explanations here and there

Constructive comments or error reports on this set of slides would be welcome and highly appreciated. Please contact Chalermsak.c@ku.ac.th