



香港中文大學(深圳)
The Chinese University of Hong Kong, Shenzhen

Introduction to Computer Science: Programming Methodology

Lecture 3 Flow Control

Prof. Yunming XIAO
School of Data Science

Please help to make this class better!

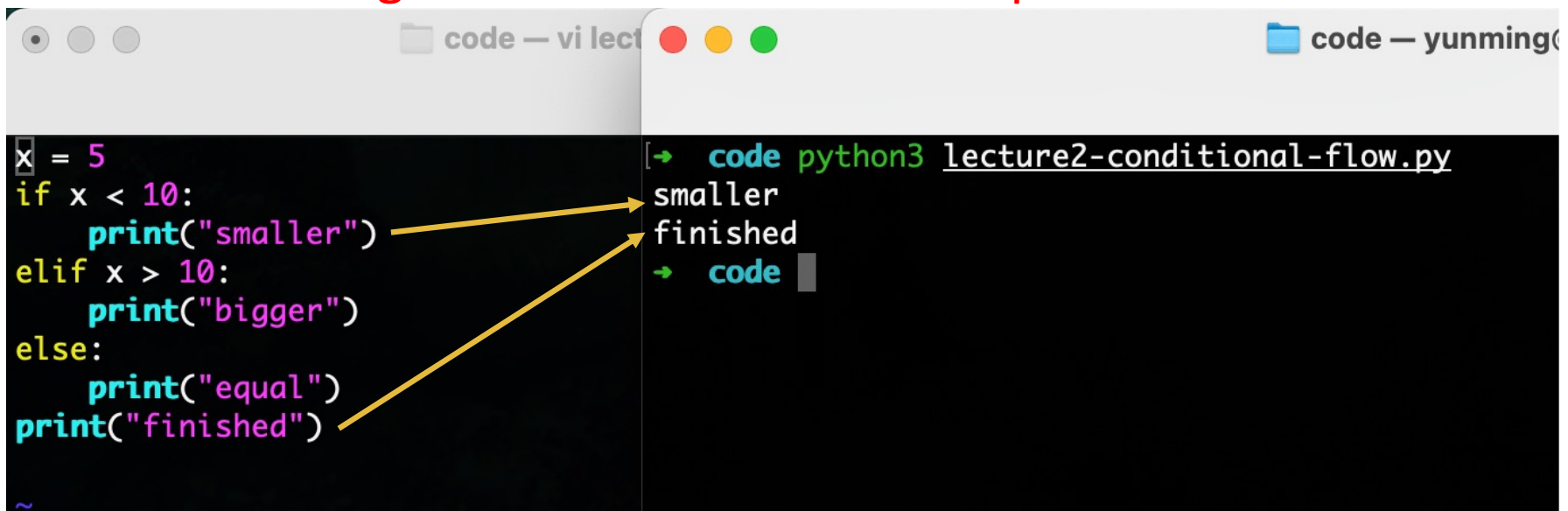


<https://forms.cloud.microsoft/r/euzTPwjksR>

Conditional flow

Program

Outputs



```
x = 5
if x < 10:
    print("smaller")
elif x > 10:
    print("bigger")
else:
    print("equal")
print("finished")
```

```
→ code python3 lecture2-conditional-flow.py
smaller
finished
→ code
```

The image shows a side-by-side comparison of a Python program and its execution output. The left window, titled 'code — vi lect', contains the source code. The right window, titled 'code — yunmingo', shows the output of running the program. Two yellow arrows point from the code to the output: one from the `print("smaller")` line to the output 'smaller', and another from the `print("finished")` line to the output 'finished'.

Comparison operators

- **Boolean expressions** ask a question and produce a **Yes/No** result, which we use to **control program flow**

- **Boolean expressions** use **comparison operators** to evaluate Yes/No or True/False

$x < y$	Is x less than y?
$x \leq y$	Is x less than or equal to y?
$x == y$	Is x equal to y?
$x \geq y$	Is x greater than or equal to y?
$x > y$	Is x greater than y?
$x \neq y$	Is x not equal to y?

- **Comparison operators** check variables but do not change the values of variables
- **Careful!!** “=” is used for assignment

Examples of comparison

```
[>>> 5 > 7
False
[>>> x, y = 45, -3.0
[>>> x > y
True
[>>> result = x > y + 50
[>>> result
False
[...     print("I think this should print.")
[...
I think this should print.
[>>> "hello" > "Bye"
True
[>>> "AAB" > "AAC"
False
[>>>
```

- Python 3 uses the **lexicographic** (dictionary) order for strings
- Capital letters are **always before** lower case letters

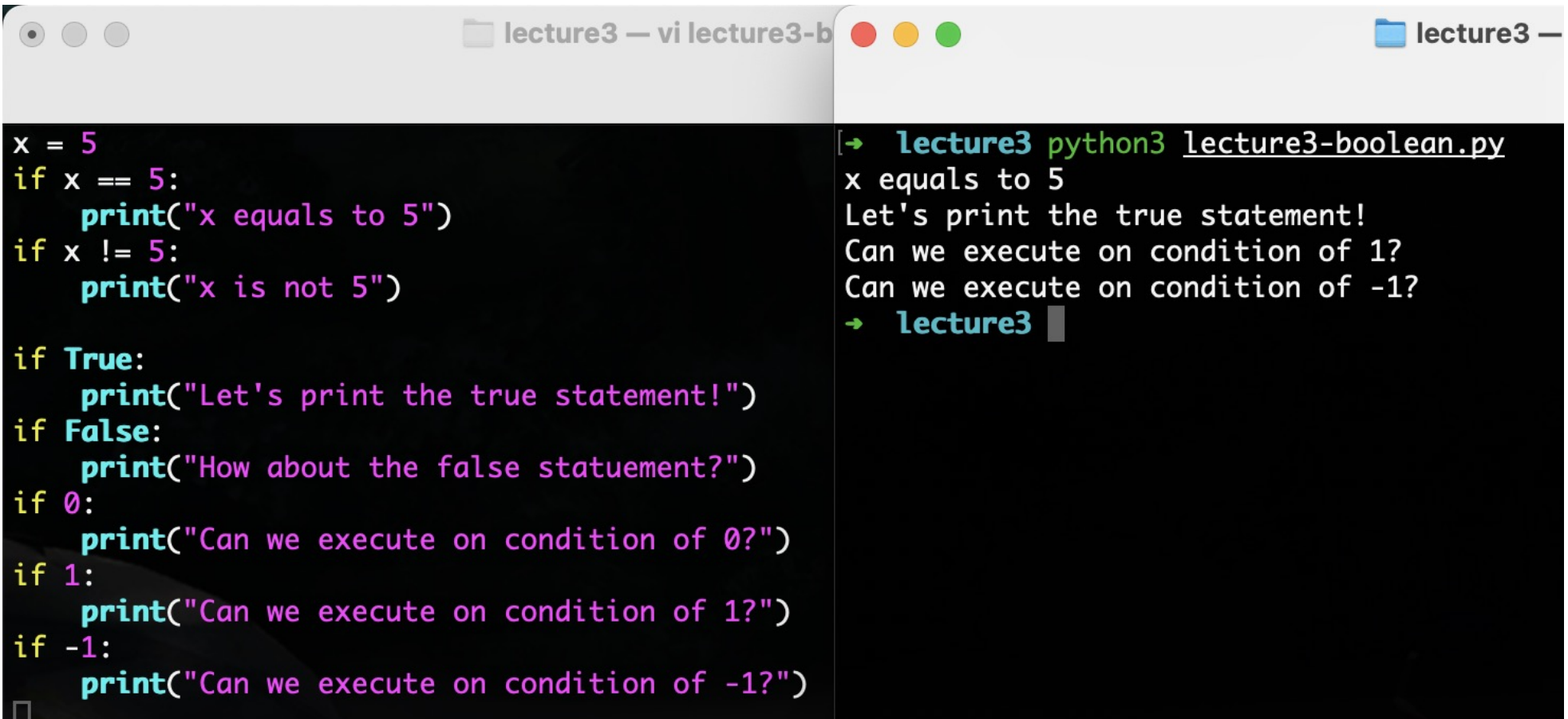
Examples of comparison

```
[>>> 7 == 7.0
True
[>>> x = 0.1
[>>> 1 == 10 * x
True
[>>> 1 == x + x + x + x + x + x + x + x + x + x + x
False
[>>> x + x + x + x + x + x + x + x + x + x + x
0.9999999999999999
[>>> 7 != "7"
True
[>>> "A" == 65
False
[>>>
```

Boolean type

- Python contains a built-in **Boolean type**, which takes two values **True/False**
- Number 0 can also be used to represent **False**. All other numbers represent **True**
- Reversely, **True** is considered as **1** if involved in numerical calculation, and **False** is considered as **0**

Boolean type in conditional flow



The image shows two side-by-side terminal windows. The left window, titled 'lecture3 — vi lecture3-b', contains Python code for conditional execution. The right window, titled 'lecture3 —', shows the output of running the code.

```
x = 5
if x == 5:
    print("x equals to 5")
if x != 5:
    print("x is not 5")

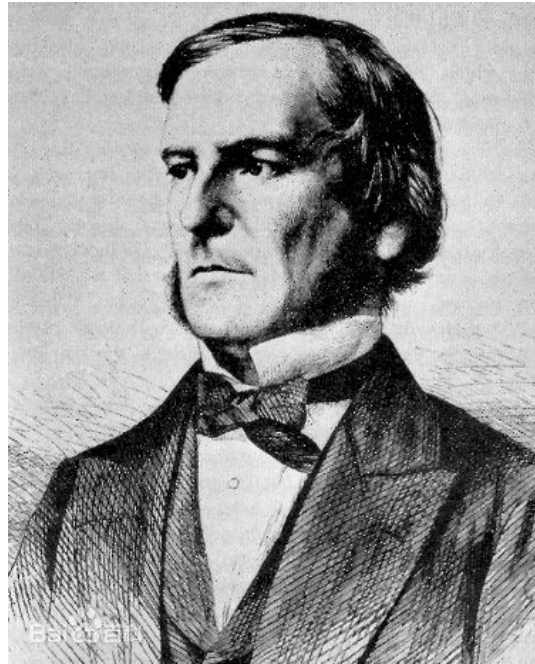
if True:
    print("Let's print the true statement!")
if False:
    print("How about the false statuemnt?")
if 0:
    print("Can we execute on condition of 0?")
if 1:
    print("Can we execute on condition of 1?")
if -1:
    print("Can we execute on condition of -1?")
```

→ lecture3 python3 lecture3-boolean.py
x equals to 5
Let's print the true statement!
Can we execute on condition of 1?
Can we execute on condition of -1?
→ lecture3

Bool()

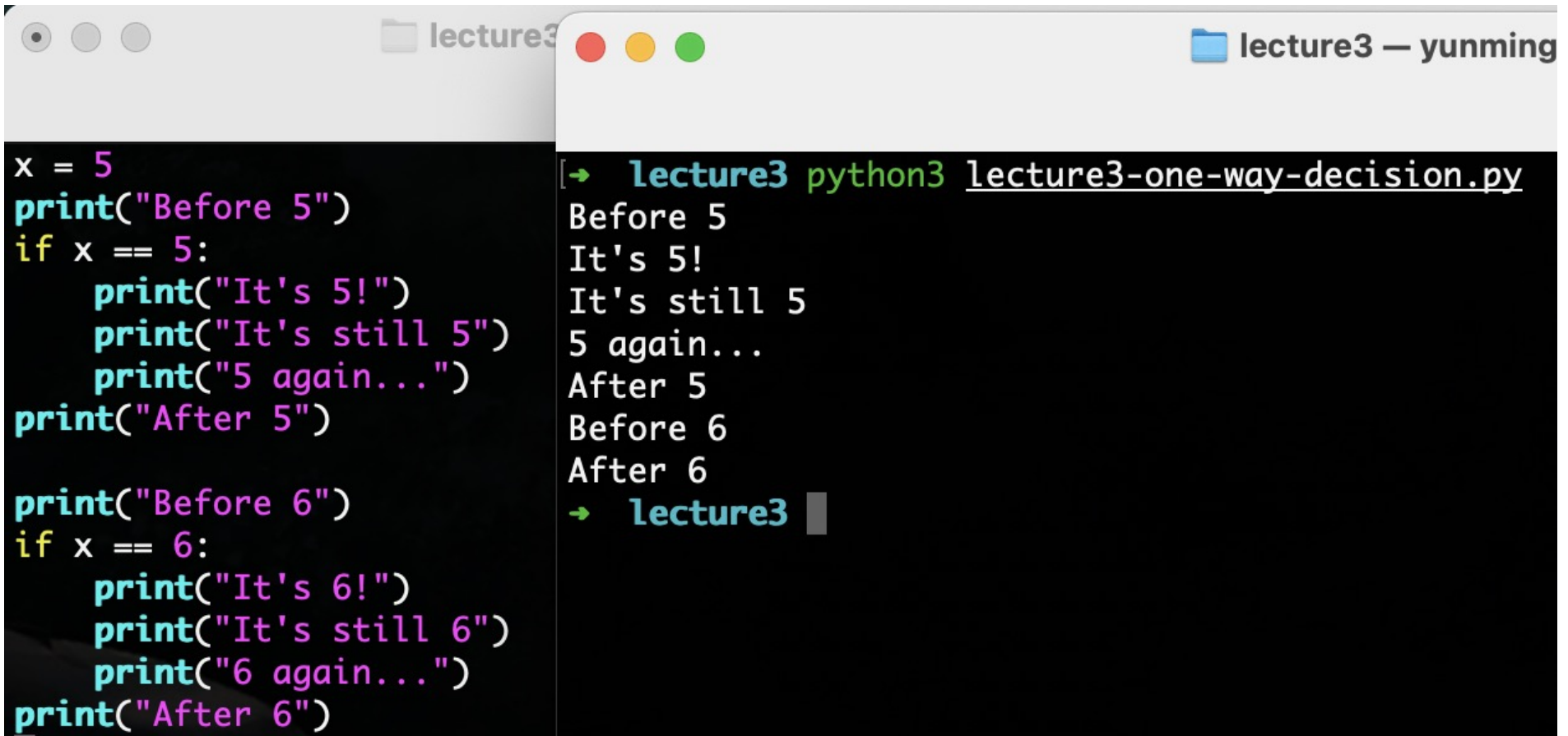
```
[>>> x = 0; y = 0.0; z = 0 + 0j
[>>> bool(x), bool(y), bool(z)
(False, False, False)
[>>> x = -1; y = 1.e-10; z = 0 + 1j
[>>> bool(x), bool(y), bool(z)
(True, True, True)
[>>> x = []; y = [0]; z = "0"
[>>> bool(x), bool(y), bool(z)
(False, True, True)
[>>>
[>>> 42 + True
43
[>>> 42 - False
42
```

Boolean type



George Boole (1815 - 1864): Mathematician, inventor of mathematical logic, significant contributions to differential and difference equations

One-way decisions



The image shows two overlapping terminal windows. The background window, titled 'lecture3', contains Python code for a one-way decision. The foreground window, titled 'lecture3 — yunming', shows the output of running the code with the command 'python3 lecture3-one-way-decision.py'.

```
x = 5
print("Before 5")
if x == 5:
    print("It's 5!")
    print("It's still 5")
    print("5 again...")
print("After 5")

print("Before 6")
if x == 6:
    print("It's 6!")
    print("It's still 6")
    print("6 again...")
print("After 6")
```

→ lecture3 python3 lecture3-one-way-decision.py

```
Before 5
It's 5!
It's still 5
5 again...
After 5
Before 6
After 6
→ lecture3
```

Indentation

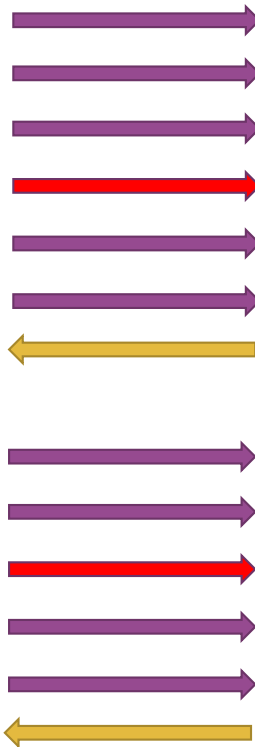
- **Increase indent**: indent after an **if** or **for** statement (after :)
- **Maintain indent**: to indicate the **scope** of the block (which lines are affected by the **if/for**)
- **Decrease indent**: to **back to** the level of the if statement or for statement to indicate the end of the block
- **Blank lines** are ignored – they **do not affect indentation**
- **Comments** on a line by themselves are **ignored** w.r.t. indentation

Increase/maintain/decrease

- **Increase/maintain** after if/for statements

- **Decrease** to indicate the end of a block

- **Blank lines and comments** are ignored



```
x = 5
print("Before 5")
if x == 5:
    print("It's 5!")
    print("It's still 5")
    print("5 again...")
print("After 5")

print("Before 6")
if x == 6:
    print("It's 6!")
    print("It's still 6")
    print("6 again...")
print("After 6")
```

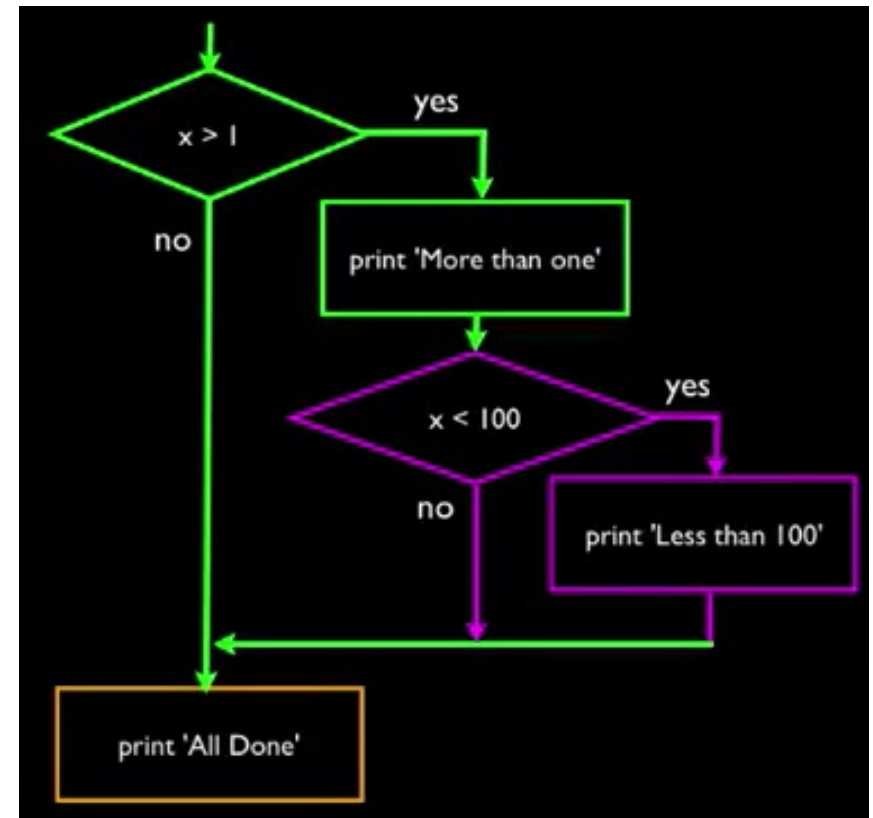
Nested decisions

Example

```
x = 42 # What about 101? -1?
if x > 1:
    print("More than 1")

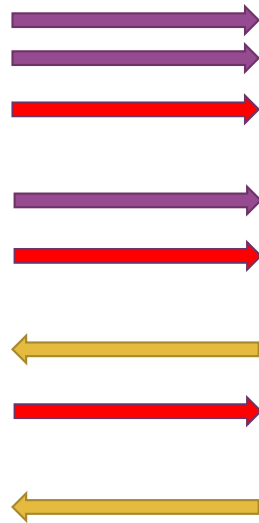
    if x < 100:
        print("Less than 100")

print("Finished")
```



Flow chart

Mental begin/end



```
x=10
if x>5:
    print('Greater than 5')

    if x>8:
        print('Greater than 8')

        if x>10:
            print('Greater than 10')

print('Finished')
```

Too many nested decisions will be a disaster...

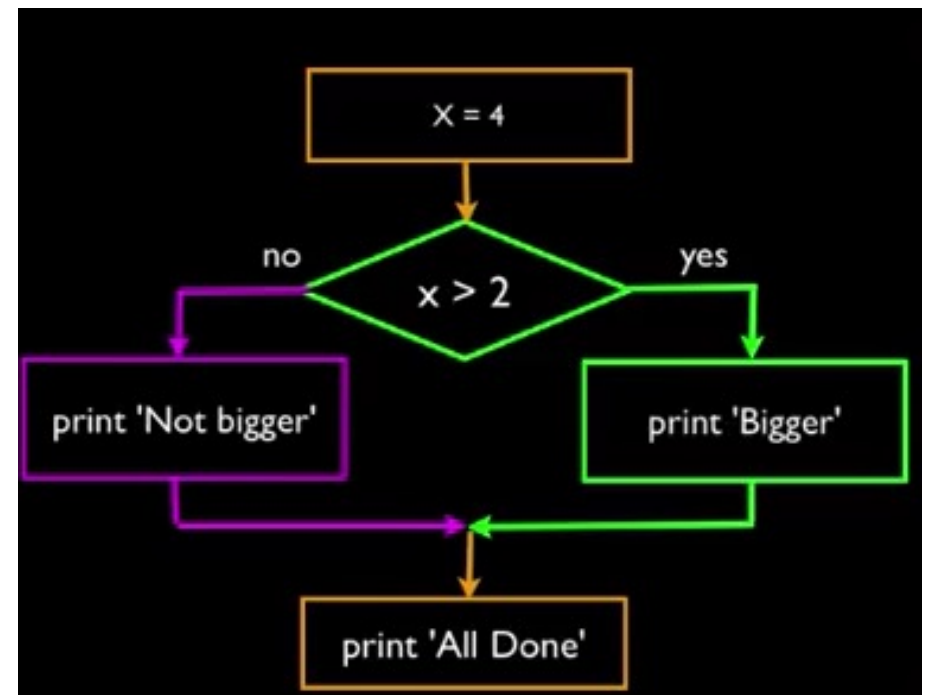
```
function register()
{
    if (!empty($_POST)) {
        $msg = '';
        if ($_POST['user_name']) {
            if ($_POST['user_password_new']) {
                if ($_POST['user_password_new'] === $_POST['user_password_repeat']) {
                    if (strlen($_POST['user_password_new']) > 5) {
                        if (strlen($_POST['user_name']) < 65 && strlen($_POST['user_name']) > 1) {
                            if (preg_match('/^[a-z\d]{2,64}$/i', $_POST['user_name'])) {
                                $user = read_user($_POST['user_name']);
                                if (!isset($user['user_name'])) {
                                    if ($_POST['user_email']) {
                                        if (strlen($_POST['user_email']) < 65) {
                                            if (filter_var($_POST['user_email'], FILTER_VALIDATE_EMAIL)) {
                                                create_user();
                                                $_SESSION['msg'] = 'You are now registered so please login';
                                                header('Location: ' . $_SERVER['PHP_SELF']);
                                                exit();
                                            } else $msg = 'You must provide a valid email address';
                                        } else $msg = 'Email must be less than 64 characters';
                                    } else $msg = 'Email cannot be empty';
                                } else $msg = 'Username already exists';
                            } else $msg = 'Username must be only a-z, A-Z, 0-9';
                        } else $msg = 'Username must be between 2 and 64 characters';
                    } else $msg = 'Password must be at least 6 characters';
                } else $msg = 'Passwords do not match';
            } else $msg = 'Empty Password';
        } else $msg = 'Empty Username';
        $_SESSION['msg'] = $msg;
    }
    return register_form();
}
```



C#论坛
www.ibcibc.com

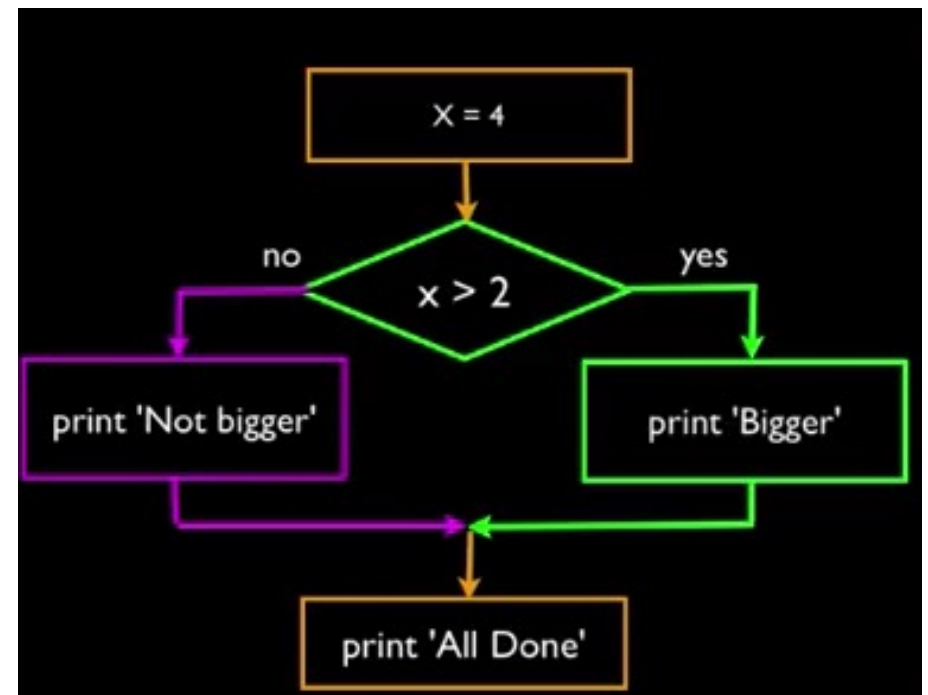
Two-way decisions

- Sometimes we want to do one thing when the logical expression is true, and another thing when it is false
- It is like a fork in the road, we need to choose **one or the other path**, but **not both**



Two-way decisions

```
x = 4
if x > 2:
    print("Larger")
else:
    print("Not larger")
```



Pay attention to indentation!

```
x = 4
if x > 2:
    print("Larger")
else:
    print("Not larger")
```



```
x = 4
if x > 2:
    print("Larger")
else:
    print("Not larger")
```



- Else must come after if
- Use **indentation** to match if and else

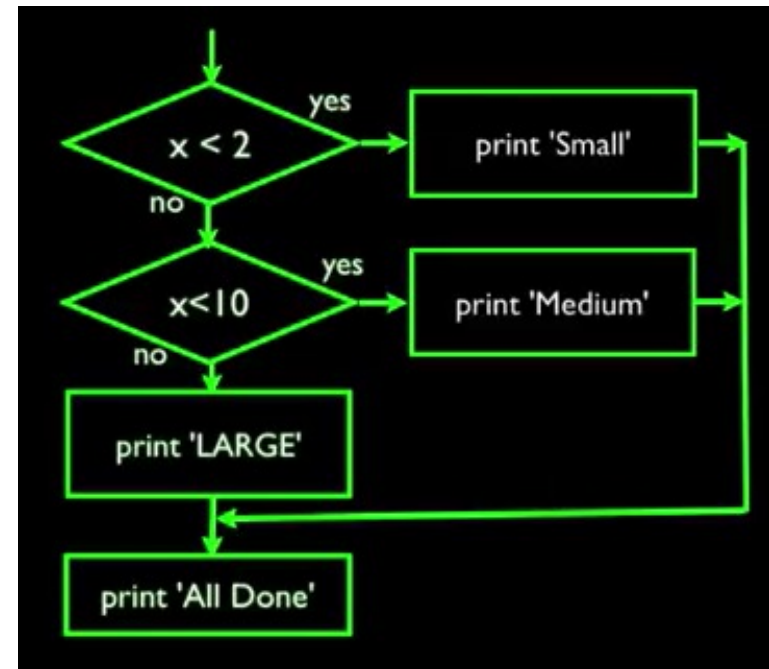
More example

```
x = 3
if x > 2:
    if x > 4:
        print("Larger")
    else:
        print("Smaller")
print("Finished")
```

```
x = 3
if x > 2:
    if x > 4:
        print("Larger")
else:
    print("Smaller")
print("Finished")
```

Multi-way decisions

```
x = 5
if x < 10:
    print("smaller")
elif x > 10:
    print("bigger")
else:
    print("equal")
print("finished")
~
```



Multi-way decisions

```
x = 5
if x < 10:
    print("smaller")
elif x > 10:
    print("bigger")
print("finished")
```

- We can live without “else”

Multi-way decisions

```
x = 42
if x < 2:
    print("Small")
elif x < 8:
    print("bigger")
elif x < 32:
    print("Large")
elif x < 128:
    print("Huge")
else:
    print("Ginormous")
print("Finished")
```

```
x = 42
if x < 2:
    print("Small")
if x < 8:
    print("bigger")
if x < 32:
    print("Large")
if x < 128:
    print("Huge")
else:
    print("Ginormous")
print("Finished")
```

- What's the difference?

Any wasted code?

```
x = eval(input())
if x <= 2:
    print("Below 2")
elif x > 2:
    print("Above 2")
else:
    print("Something else")
print("Finished")
```

```
x = eval(input())
if x < 2:
    print("Below 2")
elif x < 8:
    print("Below 8")
elif x < 4:
    print("Below 4")
else:
    print("Something else")
print("Finished")
```

Break

Logical operators

- Logical operators can be used to combine several logical expressions into a single expression
- Python has three logical operators: not, and, or

Examples of logical operators

```
[>>> not True
False
[>>> False and True
False
[>>> not False and True
True
[>>> (not False) and True
True
[>>> True or False
True
[>>>
```

Examples of logical operators

```
[>>> not False or True  
True  
[>>> not (False or True)  
False  
[>>> False and False or True  
True  
[>>> False and (False or True)  
False
```

Let's handle errors!

- Codes need to face all kinds of possible scenarios/inputs

```
→ lecture2 python3 lecture2-bmi.py
This is a program to calculate the BMI
Please type in your weight in kilograms:
123abc
Traceback (most recent call last):
  File "/Users/yunming/Documents/CUHKSZ/Courses/CSC1001-25fall
/code/lecture2/lecture2-bmi.py", line 8, in <module>
    weight = float(weight)
ValueError: could not convert string to float: '123abc'
→ lecture2 █
```

A software tester walks into a bar...

- Runs into a bar.
 - Crawls into a bar.
 - Dances into a bar.
 - Flies into a bar.
 - Jumps into a bar.
- And orders:
 - a beer.
 - 2 beers.
 - 0 beers.
 - 99999999 beers.
 - a lizard in a beer glass.
 - -1 beer.
 - "qwertyuiop" beers.
- Testing complete.
 - A real customer walks into the bar and asks where the bathroom is.
 - The bar goes up in flames.

Try/except structure

- You surround a dangerous part of code with **try/except**
- If the code in try block **works**, the except block is **skipped**
- If the code in try block **fails**, the except block will be **executed**

Use try/except structure to capture errors

```
print("Please type in your weight in kilograms:")  
weight = input()  
try:  
    weight = float(weight)  
except:  
    print("Invalid input! Now I assume you have 68kg")  
    weight = 50.0
```

- If the conversion **succeeds**, it just **skips the except block**
- If the conversion **fails**, it just **jumps into the except block**, and the program continues

Practice

- Write a program to instruct the user to input the working hours and hourly rate, and then output the salary. If the working hours exceed 40 hours, then the extra hours received 1.5 times pay.
- If the working hour input is invalid, set it to be 0
- If the hourly rate is invalid, set it to 23.40 (minimum by law)

Practice

- Write a program to instruct a user to input a date (both month and day), and then output the new month and day when the inputted date is advanced by one day (leap years are ignored)
- If the month input is invalid, set it to be 9
- If the day input is invalid, set it to 16

Break

Repeated flow

Program

```
n = 5
while n > 0:
    print(n)
    n = n - 1
print("finished")
~
~
~
```

Outputs

```
code python3 lecture2-repeated-flow.py
5
4
3
2
1
finished
code
```

- Loops (repeated steps) have iterative variables that change each time through a loop
- Often these iterative variables go through a sequence of numbers

A problematic loop

```
n = 5
while n > 0:
    print("Wash")
    print("Rinse")
n = n - 1
print("Dry off!")
```

- What's wrong with this program?

Another problematic loop

```
n = 0
while n > 0:
    print("Wash")
    print("Rinse")
    n = n - 1
print("Dry off!")
```

- What's wrong with this program?

Breaking out of a loop

- The break statement ends the current loop, and jumps to the statement which directly follows the loop

```
while True:
    line = input("Enter a word:")
    if line == "done":
        break
    print(line)
print("Finished")
```

Finishing an iteration with continue

- The **continue** statement ends the current iteration, and **start the next iteration immediately**

```
while True:
    line = input("Enter a word:")
    if line[0] == "#":
        continue
    if line == "done":
        break
    print(line)
print("Finished")
```

Indefinite loop

- **While** loops are called “indefinite loops”, since they keep going until a logical condition becomes **false**
- Till now, the loops we have seen are relatively easy to check whether they will terminate
- Sometimes it can be hard to determine whether a loop will terminate

Definite loop

- Quite often we have **a finite set of items**
- We can use a loop, each iteration of which will be executed for each item in the set, using the **for** statement
- These loops are called “definite loops” because they execute **an exact number of times**
- It is said that “definite loops iterate through the members of a set”

A simple definite *for* loop

Program

```
for i in [5, 4, 3, 2, 1]:  
    print(i)  
print("Finished")
```

Outputs

```
lecture3 python3 lecture3-for.py  
5  
4  
3  
2  
1  
Finished  
→ lecture3
```

A simple definite *for* loop

Program

```
friends = ["Tom", "Jerry", "Bat"]
for friend in friends:
    print("Happy new year", friend)
print("Finished")
```

Outputs

```
lecture3 python3 lecture3-for2.py
Happy new year Tom
Happy new year Jerry
Happy new year Bat
Finished
→ lecture3
```

For loop

Program

```
for i in [5, 4, 3, 2, 1]:  
    print(i)  
print("Finished")  
~  
~  
~  
~
```

Outputs

```
lecture3 python3 lecture3-for.py  
5  
4  
3  
2  
1  
Finished  
→ lecture3
```

- **For loops (definite loops)** have explicit iteration variables that change each time through a loop.
- These iteration variables move through a sequence or a set

For loop

- The iteration variable “**iterates**” through a **sequence** (ordered set)
- The block (body) of the code is executed once for each value **in** the sequence
- The **iteration variable** moves through **all** of the values in the sequence

Iteration variable

Sequence with
five elements



```
for i in [5, 4, 3, 2, 1]:  
    print(i)
```

Loop patterns

- Note: though these examples are simple, the patterns apply to all kinds of loops

Making “smart” loops

- The trick is “knowing” something about the whole loop when you are stuck writing code that only sees one entry at a time

Set some variables to initial values

for things in data:

Look for something or do something to each entry respectively

Update a variable

Check out the variables

Looping through a list

Program

```
print("Before")
for thing in [3, 5, 100, 34, 6, 87]:
    print(thing)
print("After")
```

Outputs

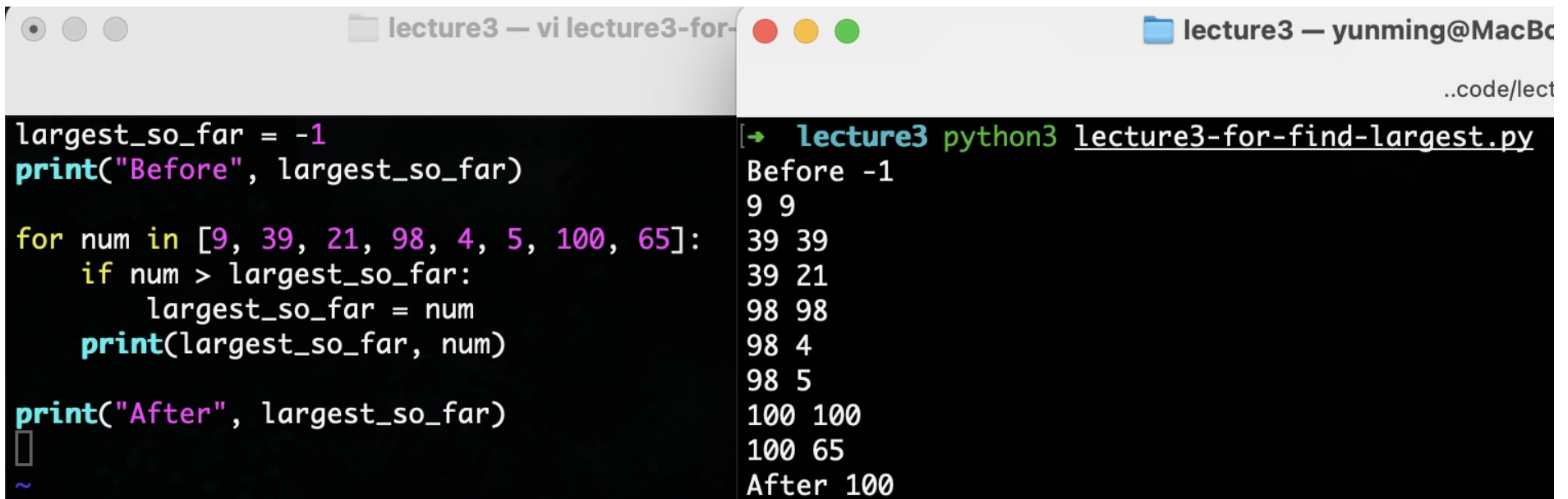
```
lecture3 python3 lecture3-for-list.py
Before :
3
5
100
34
6
87
After
```

- **For loops (definite loops)** have explicit iteration variables that change each time through a loop.
- These iteration variables move through a sequence or a set

Example: finding the largest number

Program

Outputs



```
lecture3 — vi lecture3-for-
largest_so_far = -1
print("Before", largest_so_far)

for num in [9, 39, 21, 98, 4, 5, 100, 65]:
    if num > largest_so_far:
        largest_so_far = num
        print(largest_so_far, num)

print("After", largest_so_far)
~

lecture3 — yunming@MacBo
..code/lect
-> lecture3 python3 lecture3-for-find-largest.py
Before -1
9 9
39 39
39 21
98 98
98 4
98 5
100 100
100 65
After 100
```

- Use a **variable** to store the largest number we have seen so far
- If the current number is **larger**, we assign it to the store variable

Counting in a loop

Program

```
count = 0
print("Before", count)
for thing in [3, 4, 98, 38, 9, 10, 199, 78]:
    count += 1
    print(count, thing)
print("After", count)
```

Outputs

```
lecture3 python3 lecture3-for-counting.py
Before 0
1 3
2 4
3 98
4 38
5 9
6 10
7 199
8 78
After 8
```

- To count **how many times** we have executed a loop, we can introduce a counting variable, which **increases itself** in each iteration

Practice

- Given a set of numbers, write a program to calculate their sum using for loop

Practice

- Given a set of numbers, write a program to calculate their average using for loop

Filtering in a loop

Program

```
print("Before")
for value in [23, 3, 39, 80, 111, 99, 3]:
    if value > 50:
        print("Large value", value)
print("After")
```

Outputs

```
→ lecture3 python3 lecture3-for-filtering.py
Before
Large value 80
Large value 111
Large value 99
After
→ lecture3
```

- We can use an **if** statement in a loop to **catch/filter** the values we are interested at

Search using a Boolean variable

Program

```
found = False

print("Before", found)

for value in [9, 41, 12, 3, 74, 15]:
    if value == 74:
        found = True
        print(found, value)

print("After", found)
```

Outputs

```
→ lecture3 python3 lecture3-search-boolean.py
Before False
False 9
False 41
False 12
False 3
True 74
True 15
After True
→ lecture3
```

- If we want to search in a set and double check whether **a specific number is in that set**
- We can use a Boolean variable, set it to **False at the beginning**, and assign **True** to it as long as the **target number is found**

Alternative: search using break

Program

```
found = False
print("Before", found)
for value in [9, 41, 12, 3, 74, 15]:
    if value == 74:
        break
    print(found, value)
print("After", found)
```

Outputs

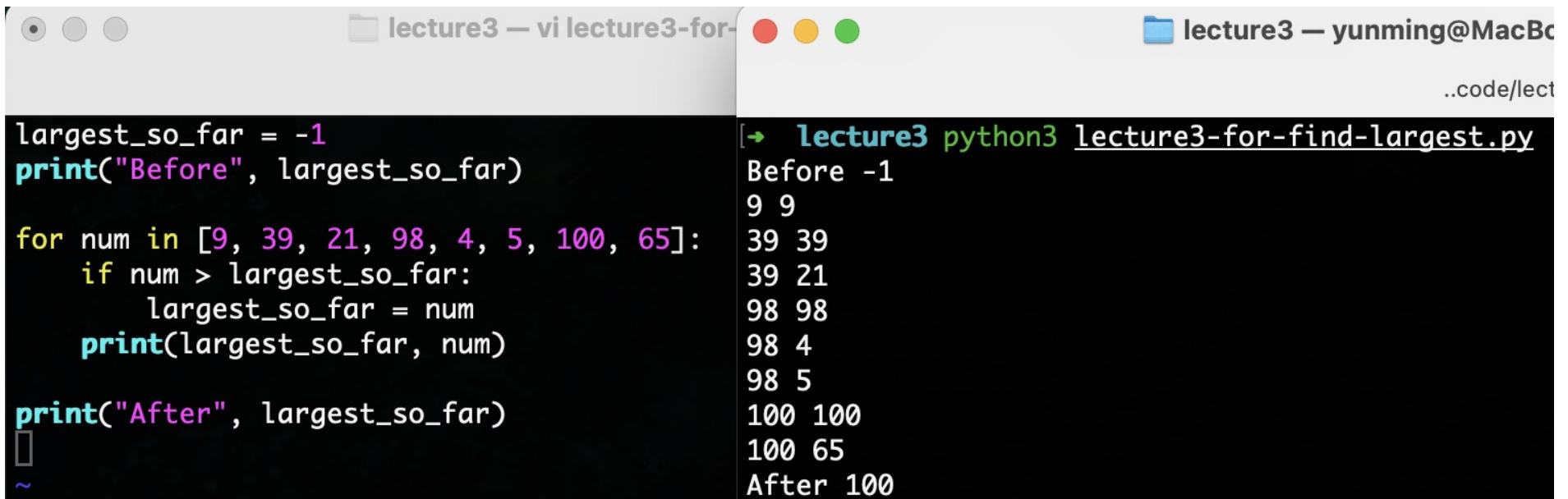
```
→ lecture3 python3 lecture3-search-break.py
Before False
False 9
False 41
False 12
False 3
After False
→ lecture3
```

- We can also use **break** to finish this task

Example: finding the largest number

Program

Outputs



```
lecture3 — vi lecture3-for-
largest_so_far = -1
print("Before", largest_so_far)

for num in [9, 39, 21, 98, 4, 5, 100, 65]:
    if num > largest_so_far:
        largest_so_far = num
        print(largest_so_far, num)

print("After", largest_so_far)
~
```

```
lecture3 — yunming@MacBo
..code/lect
-> lecture3 python3 lecture3-for-find-largest.py
Before -1
9 9
39 39
39 21
98 98
98 4
98 5
100 100
100 65
After 100
```

- Use a **variable** to store the largest number we have seen so far
- If the current number is **larger**, we assign it to the store variable

Finding the smallest number

```
smallest_so_far = -1
print("Before", smallest_so_far)

for num in [9, 39, 21, 98, 4, 5, 100, 65]:
    if num < smallest_so_far:
        smallest_so_far = num
        print(smallest_so_far, num)

print("After", smallest_so_far)
```

- Use a **variable** to store the smallest number we have seen so far
- If the current number is **smaller**, we assign it to the store variable
- What is the problem with this program?

Finding the smallest number

Program

Outputs

```
lecture3 — vi lecture3-for-
smallest_so_far = None
print("Before", smallest_so_far)

for num in [9, 39, 21, 98, 4, 5, 100, 65]:
    if smallest_so_far == None:
        smallest_so_far = num
    if num < smallest_so_far:
        smallest_so_far = num
    print(smallest_so_far, num)

print("After", smallest_so_far)
```

```
lecture3 — yunming@MacBook-
..code/lecture3
→ lecture3 python3 lecture3-for-find-smallest2.py
Before None
9 9
9 39
9 21
9 98
4 4
4 5
4 100
4 65
After 4
→ lecture3
```

- In the first iteration, the smallest value is **none**, so we need to use an **if** statement to check this

The *is* and *is not* operator

- Python has a “*is*” operator which can be used in logical expression
- Implies “*is the same as*”
- Similar to, but stronger than `==`
- “*is not*” is also an operator

```
smallest_so_far = None
print("Before", smallest_so_far)

for num in [9, 39, 21, 98, 4, 5, 100, 65]:
    if smallest_so_far == None:
        smallest_so_far = num
    if num < smallest_so_far:
        smallest_so_far = num
    print(smallest_so_far, num)

print("After", smallest_so_far)
```

The *is* and *is not* operator

```
[>>> print(10 is 10)
<python-input-10>:1: SyntaxWarning:
True
[>>> a = 10
[>>> b = 10
[>>> print(a is b)
True
[>>> a = '123'
[>>> b = '123'
[>>> print(a is b)
True
[>>> a = [1, 2, 3]
[>>> b = [1, 2, 3]
[>>> print(a is b)
False
```

Thanks