CONDITIONAL EXECUTION

LECTURE 02-1

THE IF STATEMENT
THE BOOLEAN TYPE
THE WHILE LOOP

JIM FIX, REED COLLEGE CSCI 121

THINGS

- ***Homework 1*** due tomorrow at 9am
  - ➡ Any questions?

THINGS

- ***Homework 1*** due tomorrow at 9am
  - ➡ Any questions?
- Drop-in tutoring:
  - ➡ SuMTuWTh, Library 340, 7-9pm

SCHEDULE:

Date	Week	Monday (Lecture)	Tuesday (Lab)	Wednesday (Lecture)	Thursday	Friday
26-Jan-26	1	course overview interaction: integers; strings; floats scripts: input; print; assignment	• **Homework 1 out** "*input, calculation, output*"	example scripts with % and // booleans; conditions if: the conditional statement		
2-Feb-26	2	the frame mechanism while loops definite vs. indefinite loops	• **Homework 2 out** "*conditionals and loops*" **» Homework 1 due**	nested loops functions: def and the return statement expression evaluation		
9-Feb-26	3	**§ Quiz #1 on scripting and //%** procedures: def and return revisited the None type	• **Homework 3 out** "*functions* and procedures" **» Homework 2 due**	the call stack mechanism **« Project 1 "*greed*" out**		
16-Feb-26	4	list construction and scanning list item update list mutability	• **Homework 4 out** "*lists*" **» Homework 3 due**	list slicing; lists of lists append, extend, insert, delete		
23-Feb-26	5	**§ Quiz #2 on functions and loops** dictionaries the for loop	• **Homework 5 out** "*dictionaries*" **» Homework 4 due**	recursive functions and procedures count up/down; sort preview **« Project 2 "ciphers" out**		**» Project 1 due**
2-Mar-26	6	the call stack revisited fibonacci, instrumented	• **Homework 6 out** "recursion" **» Homework 5 due**	objects and classes		
9-Mar-26	7	**§ Quiz #3 on lists and dictionaries** inheritance	• **Homework 7 out** "*classes and inheritance*" **» Homework 6 due**	lambda higher order functions		
16-Mar-26	8	**§ Exam on Homework 1-5** *scripting, conditionals, loops, functions, lists, dictionaries*	• **Homework 8 out** *"higher order functions"* **» Homework 7 due**	environment diagrams intro to linked lists; prepend; append **« Project 3 "hawks and doves" out**		**» Project 2 due**
		SPRING BREAK	**SPRING BREAK**	**SPRING BREAK**		**SPRING BREAK**

SCHEDULE: QUIZ NEXT WEEK!

Date	Week	Monday (Lecture)	Tuesday (Lab)	Wednesday (Lecture)	Thursday	Friday
26-Jan-26	1	course overview interaction: integers; strings; floats scripts: input; print; assignment	• **Homework 1 out** "*input, calculation, output*"	example scripts with % and // booleans; conditions if: the conditional statement		
2-Feb-26	2	the frame mechanism while loops definite vs. indefinite loops	• **Homework 2 out** "*conditionals and loops*" **» Homework 1 due**	nested loops functions: def and the return statement expression evaluation		
9-Feb-26	3	**§ Quiz #1 on scripting and //%** procedures: def and return revisited the None type	• **Homework 3 out** "*functions* and procedures" **» Homework 2 due**	the call stack mechanism **« Project 1 "*greed*" out**		
16-Feb-26	4	list construction and scanning list item update list mutability	• **Homework 4 out** "*lists*" **» Homework 3 due**	list slicing; lists of lists append, extend, insert, delete		
23-Feb-26	5	**§ Quiz #2 on functions and loops** dictionaries the for loop	• **Homework 5 out** "*dictionaries*" **» Homework 4 due**	recursive functions and procedures count up/down; sort preview **« Project 2 "ciphers" out**		**» Project 1 due**
2-Mar-26	6	the call stack revisited fibonacci, instrumented	• **Homework 6 out** "recursion" **» Homework 5 due**	objects and classes		
9-Mar-26	7	**§ Quiz #3 on lists and dictionaries** inheritance	• **Homework 7 out** "*classes and inheritance*" **» Homework 6 due**	lambda higher order functions		
16-Mar-26	8	**§ Exam on Homework 1-5** *scripting, conditionals, loops, functions, lists, dictionaries*	• **Homework 8 out** *"higher order functions"* **» Homework 7 due**	environment diagrams intro to linked lists; prepend; append **« Project 3 "hawks and doves" out**		**» Project 2 due**
		SPRING BREAK	**SPRING BREAK**	**SPRING BREAK**		**SPRING BREAK**

"FLOW OF CONTROL"

NOTE: so far, Python programs perform "*straight-line*" execution.

- The interpreter goes through the code line-by-line, tracking where it's at with an instruction pointer.
  - ➡ The movement of that pointer is called the program's *flow of control*.

- With code that has ***conditional statements*** and ***loops***, we'll see flow that's not just top to bottom.
  - ➡ Lines might get *repeatedly executed*, or lines might get *skipped*.

- With conditions and loops, there are ***branches*** in the possible flows.

CONDITIONAL EXECUTION

- We look at writing code that can run in several different ways.
- Which statements it executes depend on the conditions it checks.
- We introduce `if` and `if-else` statements, and their variants.
- **Reading** for this material:
  - PP Ch 1.7 (along with loops)
  - TP Ch 4.1 and 4.8
  - CP Ch 1.5 (along with loops)

"BRANCHING"

- Here is an example of a conditional (or "if") statement:

```
pi = 3.14159
area = float(input("Circle area? "))
if area < 0.0:
    print("That's not a valid area.")
else:
    radius = (area / pi) ** 0.5
    print("That circle's radius is "+str(radius)+".")
```

- Depending on the value of **`area`**, either the first **`print`** or the second **`print`** will execute.
  - ➡ The other one will get skipped.

THE "IF-ELSE" CONDITIONAL STATEMENT

- Python allows us to reason about values and act on them *conditionally*.
- For example, consider this script:

```
x = float(input("Enter a value: "))
if x < 0:
    abs_x = -x
else:
    abs_x = x
print("The absolute value of it is " + str(abs_x))
```

THE "IF-ELSE" CONDITIONAL STATEMENT

- Python allows us to reason about values and act on them *conditionally*.
- For example, consider this script:

```
x = float(input("Enter a value: "))
if x < 0:
    abs_x = -x
else:
    abs_x = x
print("The absolute value of it is " + str(abs_x))
```

```
% python3 absolute.py
Enter a value: -5.5
The absolute value of it is 5.5
% python3 absolute.py
Enter a value: 105.77
The absolute value of it is 105.77
% python3 absolute.py
Enter a value: 0.0
The absolute value of it is 0.0
```

THE "IF-ELSE" CONDITIONAL STATEMENT

- Python allows us to reason about values and act on them *conditionally*.
- For example, consider this script:

```
x = float(input("Enter a value: "))
if x < 0:
    abs_x = -x
else:
    abs_x = x
print("The absolute value of it is " + str(abs_x))
```

- When fed a negative value, it prints the value with its sign flipped.
  - ➡I.e. the positive value with the same magnitude. `-5.5` ~> `5.5`
- Otherwise, if positive or `0.0`, it just prints that value.

SYNTAX: IF–ELSE STATEMENT

Below is a template for conditional statements:

```
if condition-expression:
        lines of statements executed if the condition holds
        ...
else:
        lines of statements executed if the condition does not hold
        ...
lines of code executed after, in either case
```

CONDITION EXPRESSIONS COMPUTE A BOOL VALUE

```
>>> 345 < 10
False
>>> 345 == 300 + 50 - 5
True
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
>>> x = 57
>>> x == 57
True
>>> x != 57
False
>>> x > 0
True
>>> x <= 100
True
>>> x > 100
False
```

COMPARISON OPERATIONS

- The full range of comparisons you can make are:

`==`	equality
`!=`	inequality
`<`	less than
`>`	greater than
`>=`	greater than or equal
`<=`	less than or equal

CONDITIONAL STATEMENT EXECUTION

- Python allows us to reason about values and act on them *conditionally*.
- For example, consider this script:

```
x = float(input("Enter a value: "))
if x < 0:
    abs_x = -x
else:
    abs_x = x
print("The absolute value of it is " + str(abs_x))
```

When the script is run, the **`if`** code gets executed as follows:

- Python first checks the condition before the colon.
  - ➡ If the condition is **`True`**, it executes the first **`abs_x`** assignment statement.
  - ➡ If the condition is **`False`**, it executes the 2nd **`abs_x`** assignment statement. This is the one sitting under the **`else`** line.

SYNTAX: IF–ELSE STATEMENT

Below is a template for conditional statements:

```
if condition-expression:
    lines of statements executed if the condition holds
    ...
else:
    lines of statements executed if the condition does not hold
    ...
lines of code executed after, in either case
```

- Use indentation to indicate the "true" code block and the "false" code block.

ANOTHER EXAMPLE: CHECKING PARITY

- Here is a script that acts differently, depending on the *parity* of a number.

```
n = int("Enter an integer: ")
if n % 2 == 0:
    print(“That number is even.”)
else:
    print(“That number is odd.”)
```

- The equality test `==` is used to compare...
  - the left-hand expression's value `n % 2`
  - with the right-hand expression's value `0`.
- It is used to check whether they are equal.

CONDITIONAL STATEMENT WITH NO ELSE

- Here is a different version of the absolute value script:

```
x = float(input("Enter a value: “))
if x < 0:
    x = -x
print("The absolute value of it is " + str(x))
```

SYNTAX: IF STATEMENT

Below is a template for conditional statements with no "else" block:

```
if condition-expression:
    lines of statements executed only if the condition holds
    ...
lines of code executed after, in either case
```

- Use indentation to indicate the "true" code block.

ANOTHER EXAMPLE: AUTOGRADER FEEDBACK

- The code below is like some code in some autograder:

```
print("Your code passed " + str(passed))
print(" out of " + str(tested) + "tests.")
if passed == tested:
    print("Your code passed all our tests!")
if not on_time:
    print("But you submitted after the deadline.")
```

NESTING CONDITIONAL STATEMENTS

- The code below is like some code in some autograder:

```
if on_time:

    if all_correct:
        mesg =  "Great work passing all the tests!\n"
        mesg += "You've earned the points for this problem."
    else:
        mesg = "To earn points, make sure all the tests pass."

else:

    if all_correct:
        mesg =  "Great work making all the tests pass.\n"
        mesg += "Sadly we can't offer you any points.\n"
        mesg += "You submitted this after the deadline."
    else:
        mesg =  "Sorry! There's still a problem. No points."

print(mesg)
```

EXPRESSING COMPLEX CONDITIONS

- The code below determines whether an integer **`rating`** is from 1 to 100:

```
rating = int(input("Enter a rating: "))
if (rating > 0) and (rating <= 100):
    print("Thanks for that rating!")
else:
    print("That is not a rating.")
```

EXPRESSING COMPLEX CONDITIONS: AND

- The code below determines whether an integer **`rating`** is from 1 to 100:

```
rating = int(input("Enter a rating: "))
if (rating > 0) and (rating <= 100):
    print("Thanks for that rating!")
else:
    print("That is not a rating.")
```

- This is using the logical connective **`and`** to check whether both conditions hold. This is their *logical conjunction*.

EXPRESSING COMPLEX CONDITIONS: OR

- The code below determines whether an integer **`rating`** is from 1 to 100:

```
rating = int(input("Enter a rating: ")
if (rating <= 0) or (rating > 100):
    print("That is not a rating.")
else:
    print("Thanks for that rating!")
```

- This is using the logical connective **`and`** to check whether both conditions hold. This is their *logical conjunction*.
- There is also the connective **`or`** for checking whether at least one condition holds. It described *logical disjunction*.

EXPRESSING COMPLEX CONDITIONS: NOT

- The code below determines whether an integer **`rating`** is from 1 to 100:

```
rating = int(input("Enter a rating: "))
if not ((rating <= 0) or (rating > 100)):
    print("Thanks for that rating!")
else:
    print("That is not a rating.")
```

- This is using the logical connective **`and`** to check whether both conditions hold. This is their *logical conjunction*.
- There is also the connective **`or`** for checking whether at least one condition holds. It described *logical disjunction*.
- There is also logical negation using **`not`**.

LOGICAL CONNECTIVES ARE BOOL OPERATORS

```
>>> x = 57
>>> (x > 0) and (x <= 100)
True
>>> (x <= 0) or (x > 100)
False
>>> not (345 < 10)
True
>>> not ((x <= 0) or (x > 100))
True
```

LOGIC CONNECTIVES ARE BOOLEAN OPERATORS

- The logical connectives **`and`**, **`or`**, and **`not`** can be thought of as operations that act on boolean values and return a boolean value:

```
>>> (7 > 3) and (2 < 4)
True
>>> (4 < 2) and False
False
>>> (2 > 3) or (not (7 < 10))
False
>>> True and False
False
>>> True or False
True
>>> not (True or False)
False
```

ITERATION; LOOPS

LECTURE 02–1

JIM FIX, REED COLLEGE CSCI 121

"LOOPING"

- Here is an example of a looping "while" statement:

```
pi = 3.14159
area = float(input("Circle area? “))
while area < 0.0:
    print(“That’s not a valid area.”)
    area = float(input(“Try again:”))
radius = (area / pi) ** 0.5
print("That circle’s radius is “+str(radius)+”.”)
```

- Because of that **`while`** statement, the re-prompting and re-input of an **`area`** with that second **`input`** can be repeatedly executed.
  - ➡ Lines 3 and 4 are repeated until the user enters a good **`area`** value.

ITERATION WITH LOOPS

- We look at code that uses ***iteration*** or ***loop*** statements.
  - ➡ In Python, these are the **`while`** and **`for`** statements.
  - ➡ These statements allow us to repeat actions several times.
    - ✦ *Definite* loops: perform an action several times.
    - ✦ *Indefinite* loops: perform an action until a condition is met.
- **Reading** about loops and iteration:
  - ✦ PP Ch 1.7
  - ✦ TP Ch 5
  - ✦ CP Ch 1.5

AN INFINITE LOOP

- Python lets you execute the same statement repeatedly with a **`while`** loop statement. For example:

```
print("This line runs once, first.")
while True:
    print("This line keeps getting run.")
print("This line never runs.")
```

- Output of the script above:

```
This line runs once, first.
This line keeps getting run.
This line keeps getting run.
This line keeps getting run.
This line keeps getting run.
…
```

- **NOTE:** hit [CTRL-c] to terminate the Python script's execution.

MORE LOOPING FOREVER

- The prior example loops forever. And so does this one:

```
print("This line runs once, first.")
while True:
    print("This line keeps getting run.")
    print("And so does this one.")
print("This line never runs.")
```

- Output of the script above:

```
This line runs once, first.
This line keeps getting run.
And so does this one.
This line keeps getting run.
And so does this one.
This line keeps getting run.
And so does this one.
…
```

COUNTING FOREVER

- The prior example loops forever. And so does this one:

```
hellos_said = 0
while True:
    print(“Hello!!!”)
    hellos_said = hellos_said + 1
    print(“That was 'hello' #” + str(hellos_said) + ".")
print(“This line never runs.”)
```

- Output of the script above:

```
Hello!!!
That was 'hello' #1.
Hello!!!
That was 'hello' #2.
Hello!!!
That was 'hello' #3.
Hello!!!
That was 'hello' #4.
…
```

COUNTING ONLY SO FAR

- This outputs a count from 0 up to 5:

```
print("I'm going to count for you.")
count = 0
while count < 6:
    print(count)
    count = count + 1
print("I'm done counting now.")
```

- Output of the script above:

```
I'm going to count for you.
0
1
2
3
4
5
I'm done counting now.
```

COUNTING ONLY SO FAR

- This outputs a count from 0 up to ***2***:

```
print("I'm going to count for you.")
count = 0
while count < 3:
    print(count)
    count = count + 1
print("I'm done counting now.")
```

- Output of the script above:

```
I'm going to count for you.
0
1
2
I'm done counting now.
```

COUNTING ACCORDING TO AN INPUT

- This outputs a count from 0 up to some input value:

```
print(“I'm going to count for you.”)
max = int(input("Enter how far you'd like me to count: "))
count = 0
while count <= max:
    print(count)
    count = count + 1
print(“I'm done counting now.”)
```

- Output of the script above:

```
I'm going to count for you.
Enter how far you'd like me to count: 4
0
1
2
3
4
I'm done counting now.
```

ANATOMY OF A WHILE LOOP

- The template below gives the syntax of a while loop statement:

lines of statements to execute first

`while` ***condition-expression*** **`:`**

    lines of statements to execute if the condition holds

    ...

    lines of statements to executed when the condition no longer holds

ANATOMY OF A COUNTING LOOP

- Here is the standard structure of a "counting loop":

*initialize the **count** to the **start-value***

`while` ***count*** `<` ***one-too-far*** `:`

    *actions to perform with that particular **count** value*

    *increment the **count** by 1*

*at this point can now use the fact that **count** == **one-too-far***

- This is an extremely common coding pattern...

➡ **PLEASE TAKE THIS TEMPLATE TO HEART!!!!**

DEFINITE VS. INDEFINITE LOOPS

- Some terminology:
  - "*Count up to 6.*" and "*Count up to the input value.*" are examples of *definite* loops.
  - "*Get an input until they've entered something valid.*" is an example of an *indefinite* loop. The number of repetitions isn't known.

- An example of the second kind of coding:

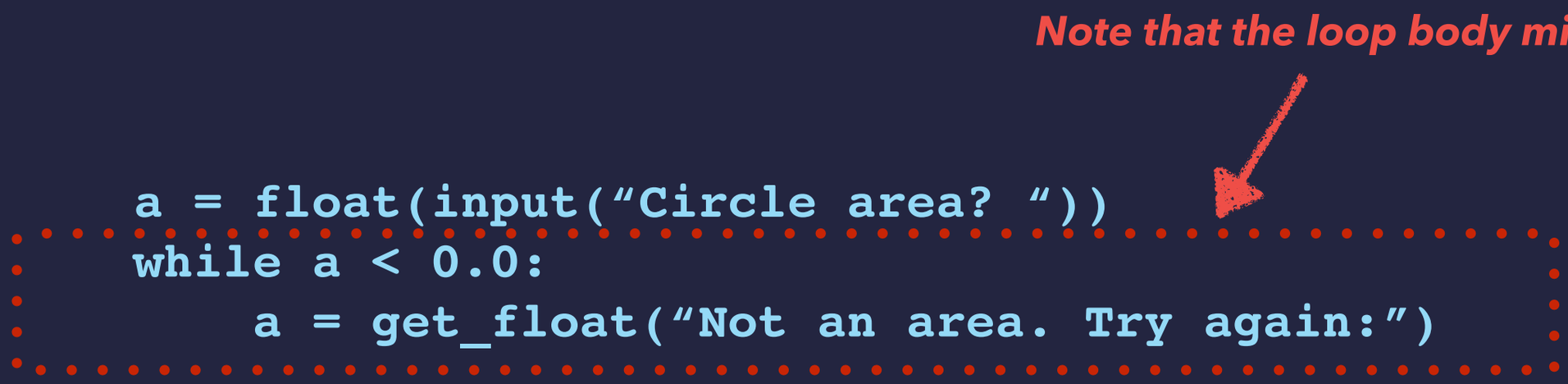
```
a = float(input("Circle area? "))
while a < 0.0:
    a = get_float("Not an area. Try again:")
```

DEFINITE VS. INDEFINITE LOOPS

- Some terminology:
  - "*Count up to 6.*" and "*Count up to the input value.*" are examples of *definite* loops.
  - "*Get an input until they've entered something valid.*" is an example of an *indefinite* loop. The number of repetitions isn't known.

- An example of the second kind of coding:

Note that the loop body might not run at all!



```
a = float(input("Circle area? "))
while a < 0.0:
    a = get_float("Not an area. Try again:")
```

SUMMARY

- The while loop statement expresses **iterative** code.
  - ➡ Allows you to perform a series of actions *until* a condition holds.
  - ➡ The negation of this *terminating condition* is the loop's condition.
- It's possible for the code to loop forever. This is an ***infinite*** loop.
- Counting loops are common examples of ***definite*** loops.
- Loops that iterate an undetermined number of times are ***indefinite***.