CONDITIONAL EXECUTION

LECTURE 02-1

THE CONDITIONAL STATEMENT
THE BOOLEAN TYPE
LOOPS

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

THINGS

- ***Homework 1*** due tomorrow at 9am
  - ➡ Any questions?
- Drop-in tutoring:
  - ➡ Su~~X~~TuWTh, Library 340, 7-9pm
  - ➡ Tonight: Jim (me) in Library 314, 7-9pm

THINGS

- ***Homework 1*** due tomorrow at 9am
  - ➡ Any questions?
- Drop-in tutoring:
  - ➡ SuMTuWTh, Library 340, 7-9pm
  - ➡ Tonight: Jim (me) in Library 314, 7-9pm
- ***Homework 2*** assigned tomorrow in lab
- I've linked two things as a moodle page:
  - ➡ Adam Groce's text ***Principled Programming***
  - ➡ Video lectures from this past fall version of CSCI 121

THINGS

- ***Homework 1*** due tomorrow at 9am
  - ➡ Any questions?
- Drop-in tutoring:
  - ➡ SuMTuWTh, Library 340, 7-9pm
  - ➡ Tonight: Jim (me) in Library 314, 7-9pm
- ***Homework 2*** assigned tomorrow in lab
- I've linked two things as a moodle page:
  - ➡ Adam Groce's text ***Principled Programming***
  - ➡ Video lectures from this past fall version of CSCI 121
- I've posted a semester schedule.

LECTURE 02-1: CONDITIONAL EXECUTION

Date	Week	Monday (Lecture)	Tuesday (Lab)	Wednesday (Lecture)	Thursday	Friday
27-Jan-25	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		
3-Feb-25	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		
10-Feb-25	3	**§ Quiz #1 on scripting and //%** procedures: def and return revisited the None type	• **Homework 3 out** "*function*s and procedures" **» Homework 2 due**	the call stack mechanism **« Project 1 "*greed*" out**		
17-Feb-25	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		
24-Feb-25	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**	
3-Mar-25	6	the call stack revisited fibonacci, instrumented	• **Homework 6 out** "recursion" **» Homework 5 due**	objects and classes		
10-Mar-25	7	**§ Quiz #3 on lists and dictionaries** inheritance	• **Homework 7 out** "*classes and inheritance*" **» Homework 6 due**	lambda higher order functions		
17-Mar-25	8	**§ Exam on Homework 1-5** *scripting, conditionals, loops, functions, lists, dictionaries*	• **Homework 8 out** "*higher order functions*" **» Homework 7 due**	environment diagrams linked list traversal; insertion **« Project 3 "hawk/dove" out**	**» Project 2 due**	
24-Mar-25		**SPRING BREAK**	**SPRING BREAK**	**SPRING BREAK**	**SPRING BREAK**	
31-Mar-25	9	**§ Quiz #4 on object orientation** linked list deletion linked list traversal	• **Homework 9 out** "*linked lists*" **» Homework 8 due**	linked list reversal algorithm efficiency big Oh and big Theta notation		
7-Apr-25	10	binary search bubblesort; insertion sort **« Project 4: "adventure" or "arcade" out**	• **Homework 10 out** "*sorting and searching*" **» Homework 9 due**	**§ Quiz #5 on recursion** quicksort; mergesort	**» Project 3 due**	
14-Apr-25	11	bst insertion and search bst traversal	• **Homework 11 out** "bsts" **» Homework 10 due**	**§ Quiz #6 on linked lists** bst deletion		
21-Apr-25	12	file I/O	• **Homework 12 out** "*files and exceptions*" ***» Homework 11 due***	exceptions		
28-Apr-25	13	**§ Exam on Homework 6-9** *recursion; objects, inheritance; higher order functions; linked lists*	• **Continue Homework 12** **» Project 4 beta test**	review networked game demo **» Homework 12 due**	**» Project 4 due**	
5-May-25		**READING WEEK**	**READING WEEK**	**READING WEEK**		
12-May-25		**§ Comprehensive Final Exam** *time/place to be determined*				

LECTURE 02-1: CONDITIONAL EXECUTION

Date	Week	Monday (Lecture)	Tuesday (Lab)	Wednesday (Lecture)	Thursday	Friday
27-Jan-25	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		
3-Feb-25	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		
10-Feb-25	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**		
17-Feb-25	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		
24-Feb-25	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**	
3-Mar-25	6	the call stack revisited fibonacci, instrumented	• **Homework 6 out** "recursion" **» Homework 5 due**	objects and classes		
10-Mar-25	7	**§ Quiz #3 on lists and dictionaries** inheritance	• **Homework 7 out** "*classes and inheritance*" **» Homework 6 due**	lambda higher order functions		
17-Mar-25	8	**§ Exam on Homework 1-5** *scripting, conditionals, loops, functions, lists, dictionaries*	• **Homework 8 out** "*higher order functions*" **» Homework 7 due**	environment diagrams linked list traversal; insertion **« Project 3 "hawk/dove" out**	**» Project 2 due**	
24-Mar-25		**SPRING BREAK**	**SPRING BREAK**	**SPRING BREAK**	**SPRING BREAK**	
31-Mar-25	9	**§ Quiz #4 on object orientation** linked list deletion linked list traversal	• **Homework 9 out** "*linked lists*" **» Homework 8 due**	linked list reversal algorithm efficiency big Oh and big Theta notation		
7-Apr-25	10	binary search bubblesort; insertion sort **« Project 4: "adventure" or "arcade" out**	• **Homework 10 out** "*sorting and searching*" **» Homework 9 due**	**§ Quiz #5 on recursion** quicksort; mergesort	**» Project 3 due**	
14-Apr-25	11	bst insertion and search bst traversal	• **Homework 11 out** "bsts" **» Homework 10 due**	**§ Quiz #6 on linked lists** bst deletion		
21-Apr-25	12	file I/O	• **Homework 12 out** "*files and exceptions*" ***» Homework 11 due***	exceptions		
28-Apr-25	13	**§ Exam on Homework 6-9** *recursion; objects, inheritance; higher order functions; linked lists*	• **Continue Homework 12** **» Project 4 beta test**	review networked game demo **» Homework 12 due**	**» Project 4 due**	
5-May-25		**READING WEEK**	**READING WEEK**	**READING WEEK**		
12-May-25		**§ Comprehensive Final Exam** *time/place to be determined*				

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(“Error: 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 error will get printed, or
  - ➡ the calculation will be made and its result reported

THE "IF-ELSE" CONDITIONAL STATEMENT

- Python allows us to reason about values and act on them *conditionally*.
- For another 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 another 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 another 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 `x`, it sets the value `abs_x` with its sign flipped.
  - ➡ the positive value with the same magnitude: `-5.5` yields `5.5`
- Otherwise, if `x` positive or `0.0`, it just sets `abs_x` to that value.

CONDITIONAL STATEMENT EXECUTION

- Python allows us to reason about values and act on them *conditionally*.
- For another 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 assignment statement.
  - ➡ If instead the condition is **`False`**, it executes the second assignment statement. This is the one sitting under the **`else`** line.

CONDITIONAL STATEMENT EXECUTION

- Python allows us to reason about values and act on them *conditionally*.
- For another 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))
```

- You could maybe say that **`if-else`** gives Python code "intelligence."
  - ➡ It reasons about the value of **`x`** and behaves one way or the other.
- The code is smart!

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
```

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.

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(“even”)
else:
    print(“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.

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(“even”)
else:
    print(“odd”)
```

- Below is it in use:

```
% python3 parity.py
Enter an integer: -11
odd
% python3 parity.py
Enter an integer: 0
even
```

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

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
```

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
```

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.

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)
```

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.

CONDITIONAL STATEMENT WITH NO ELSE

- 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))
```

CONDITIONAL STATEMENT WITH NO ELSE

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

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

SYNTAX: CASCADING IF-ELIF-...-ELSE STATEMENT

Below is a template for conditional statements:

```
if condition-1:
    execute if condition 1 holds
    ...
elif condition-2:
    execute if condition 1 does not hold but condition 2 does
    ...
...
else:
    executed if no condition above holds
    ...
lines of code executed after, in all cases
```

CASCADING IF STATEMENT

- Here is some other autograder code using the cascading conditional:

```
attempts = number_previous_submissions + 1
mesg =  "Great work passing all the tests!\n"
mesg += "You submitted " + str(attempts) + " times.\n"
if attempts <= 2:
    mesg += "You earned the full points.\n"
    mesg += "Excellent!"
elif attempts <= 6:
    mesg += "You earned 80% of the points.\n"
    mesg += "Nicely done."
else:
    mesg += "This is a few more times than we'd prefer.\n"
    mesg += "We awarded half of the points."

print(mesg)
```

SYNTAX: CASCADING IF–ELIF–...–ELIF STATEMENT

Below is a template for conditional statements:

```
if condition-1:
    execute if condition1 holds
    ...
elif condition-2:
    execute if condition1 does not hold but condition2 does
    ...
...
elif condition-n:
    execute if conditions 1 through (n-1) do not hold but condition n does
    ...
lines of code executed after, in all cases
```

SUMMARY OF OPERATIONS

- Binary operations
  - for integers: `+` `-` `*` `//` `%` `**`
  - for floats: `+` `-` `*` `/` `**`
  - for strings: `+` `*`

SUMMARY OF OPERATIONS

- Binary operations
  - for integers: `+` `-` `*` `//` `%` `**` `<` `<=` `>` `>=` `==` `!=`
  - for floats: `+` `-` `*` `/` `**` `<` `<=` `>` `>=` `==` `!=`
  - for strings: `+` `*` `<` `<=` `>` `>=` `==` `!=`

SUMMARY OF OPERATIONS

- Binary operations
  - for integers: `+ - * // % **`
  - for floats: `+ - * / **`
  - for strings: `+ *`

`<`	`<=`	`>`	`>=`	`==`	`!=`
`<`	`<=`	`>`	`>=`	`==`	`!=`
`<`	`<=`	`>`	`>=`	`==`	`!=`

SUMMARY OF OPERATIONS

- Binary operations
  - for integers: `+` `-` `*` `//` `%` `**`
  - for floats: `+` `-` `*` `/` `**`
  - for strings: `+` `*`

`<` `<=` `>` `>=` `==` `!=`

`<` `<=` `>` `>=` `==` `!=`

`<` `<=` `>` `>=` `==` `!=`

These are comparison operations

SUMMARY OF OPERATIONS

- Binary operations
  - for integers: `+` `-` `*` `//` `%` `**`
  - for floats: `+` `-` `*` `/` `**`
  - for strings: `+` `*`

`<` `<=` `>` `>=` `==` `!=`

`<` `<=` `>` `>=` `==` `!=`

`<` `<=` `>` `>=` `==` `!=`

These are comparison operations.
They produce a boolean value.

SUMMARY OF OPERATIONS

- Binary operations
  - for integers: `+ - * // % ** < <= > >= == !=`
  - for floats: `+ - * / ** < <= > >= == !=`
  - for strings: `+ * < <= > >= == !=`
  - for booleans: `and or not`

SUMMARY OF OPERATIONS

- Binary operations
  - for integers: `+` `-` `*` `//` `%` `**` `<` `<=` `>` `>=` `==` `!=`
  - for floats: `+` `-` `*` `/` `**` `<` `<=` `>` `>=` `==` `!=`
  - for strings: `+` `*` `<` `<=` `>` `>=` `==` `!=`
  - for booleans: `and` `or` `not` `==` `!=`

SUMMARY OF OPERATIONS

- Binary operations
  - for integers: `+ - * // % ** < <= > >= == !=`
  - for floats: `+ - * / ** < <= > >= == !=`
  - for strings: `+ * < <= > >= == !=`
  - for booleans: `and or not == !=`

SUMMARY OF THE CONDITIONAL STATEMENT

- The Python interpreter can be made to conditionally execute code.
- You can do so with the conditional, or `if` statement.

```
if condition-expression:
    lines of statements executed only if the condition holds
```

- You can do so with the conditional, or `if` statement.

```
if condition-expression:
    lines of statements executed only if the condition holds
else:
    lines of statements executed if the condition doesn't hold
```

- This is sometimes called a "branch"

- Indentation means something in Python! It is sensitive to it,

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

EXECUTION OF A WHILE LOOP

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

lines of "set up" statements to execute first

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

    lines of "loop body" statements to execute if the condition holds

    ...

lines of "follow up" to execute when the condition no longer holds

- Here is how Python executes this code:

1. Executes the **set up** code.
2. It evaluates the **condition**. If `False` it *skips* to **Step 5**.
3. Otherwise, if `True`, it evaluates the **loop body**'s code.
4. It goes back to **Step 2**.
5. It executes the **follow up**, and subsequent, code.

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:

```
def get_float(prompt):
    return float(input(prompt))

def get_area():
    a = get_float("Circle area? ")
    while a < 0.0:
        a = get_float("Not an area. Try again:")
    return a
```

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

NESTING CONTROL STATEMENTS WITHIN A LOOP

- Of course you can put a conditional statement within a loop's body.

```
count = 0
while count < 6:
    if count % 2 == 0:
        print(str(count) + " is even.")
    else:
        print(str(count) + " is odd.")
    count = count + 1
print("Done.")
```

- Output of the script above:

```
0 is even.
1 is odd.
2 is even.
3 is odd.
4 is even.
5 is odd.
Done.
```

NESTING A LOOP WITHIN A LOOP

- Nested loops are a common programming pattern:

```
a = 0
while a < 6:
    b = 0
    while b < 8:
        print(str(a)+str(b),end=" ")
        b = b + 1
    print()
    a = a + 1
print("Done.")
```

- ***What does the code above do???***

NESTING A LOOP WITHIN A LOOP

- Nested loops are a common programming pattern:

```
a = 0
while a < 6:
    b = 0
    while b < 8:
        print(str(a)+str(b),end=" ")
        b = b + 1
    print()
    a = a + 1
print("Done.")
```

- *It outputs a sequence of digit pairs, separated by spaces:*

```
00 01 02 03 04 05 06 07
10 11 12 13 14 15 16 17
20 21 22 23 24 25 26 27
30 31 32 33 34 35 36 37
40 41 42 43 44 45 46 47
50 51 52 53 54 55 56 57
Done.
```

NESTING A LOOP WITHIN A LOOP

- Nested loops are a common programming pattern:

```
a = 0
while a < 6:
    b = 0
    while b < 8:
        print(str(a)+str(b),end=" ")
        b = b + 1
    print()
    a = a + 1
print("Done.")
```

Inner loop, along with set-up/follow-up

- ***It outputs a sequence of digit pairs, separated by spaces:***

```
00 01 02 03 04 05 06 07
10 11 12 13 14 15 16 17
20 21 22 23 24 25 26 27
30 31 32 33 34 35 36 37
40 41 42 43 44 45 46 47
50 51 52 53 54 55 56 57
Done.
```

NESTING A LOOP WITHIN A LOOP

- Nested loops are a common programming pattern:

```
a = 0
while a < 6:
    b = 0
    while b < 8:
        print(str(a)+str(b),end=" ")
        b = b + 1
    print()
    a = a + 1
print("Done.")
```

Inner loop, along with set-up/follow-up

Outer loop, along with set-up/follow-up

- ***It outputs a sequence of digit pairs, separated by spaces:***

```
00 01 02 03 04 05 06 07
10 11 12 13 14 15 16 17
20 21 22 23 24 25 26 27
30 31 32 33 34 35 36 37
40 41 42 43 44 45 46 47
50 51 52 53 54 55 56 57
Done.
```

NESTING A LOOP WITHIN A LOOP

- Nested loops are a common programming pattern:

```
a = 0
while a < 6:
    b = 0
    while b < 8:
        print(10*a+b, end=" ")
        b = b + 1
    print()
    a = a + 1
print("Done.")
```

Executed once for each value of a.

Inner loop, along with set-up/follow-up

Outer loop, along with set-up/follow-up

- ***It outputs a sequence of digit pairs, separated by spaces:***

```
00 01 02 03 04 05 06 07
10 11 12 13 14 15 16 17
20 21 22 23 24 25 26 27
30 31 32 33 34 35 36 37
40 41 42 43 44 45 46 47
50 51 52 53 54 55 56 57
Done.
```

BREAKING OUT OF A LOOP

- Here is another way of writing the counting loop.

```
print("Counting from 0 to 5:")
count = 0
while True:
    if count >= 6:
        break
    print(count)
    count = count + 1
print("Done.")
```

- The code uses a **`break`** statement to jump down to the follow-up code.
- If within several loops, it jumps to just after the innermost one.
- This is an artificial example
- Using **`break`** statements can sometimes make code more readable than code that expresses all the "break out" or stopping conditions.

USING CONDITION VARIABLES TO GOVERN LOOPING

- Using **`break`** to express other break-out conditions:

```
while count < 6:
    if somethingElseMakesMeStop(...)
        break
    ...
    count = count + 1
print(“Done.”)
```

- I worry that **`break`** can sometimes be missed by other coders.
- I usually prefer using explicit break-out conditions instead, like so:

```
done = False
while !done and count < 6:
    if somethingElseMakesMeStop(...)
        done = True
    if not done:
        ...
        count = count + 1
print(“Done.”)
```

USING CONDITION VARIABLES TO GOVERN LOOPING

- Using **`break`** to express other break-out conditions:

```
while count < 6:
    if somethingElseMakesMeStop(...)
        break
    ...
    count = count + 1
print(“Done.”)
```

PLEASE use `break` sparingly, and with taste.

- I worry that **`break`** can sometimes be missed by other coders.
- I usually prefer using explicit break-out conditions instead, like so:

```
done = False
while !done and count < 6:
    if somethingElseMakesMeStop(...)
        done = True
    if not done:
        ...
        count = count + 1
print(“Done.”)
```

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***.

SUMMARY (CONT'D)

- Loop bodies can contain other control statements:
  - For example, you can have `if` statements or other `while` statements.
  - If another loop statement is inside, then it is a ***nested loop***.
  - If a `break` statement, we can jump out of the loop mid-body.