

PYTHON SCRIPTING (CONT'D)

CONDITIONS

LECTURE 01-2

JIM FIX, REED COLLEGE CSC1 121

COURSE WEB PAGE

- ▶ There is a course webpage at <http://jimfix.github.io/csci121>
 - It has the syllabus and a schedule of topics covered.
 - There I'll post readings, assignments, lecture materials.

COURSE WEB PAGE

- ▶ There is a course webpage at [~~http://jimfix.github.io/csci121~~](http://jimfix.github.io/csci121)
 - It has the syllabus and a schedule of topics covered.
 - There I'll post readings, assignments, lecture materials.

For now and in the near future, it is at

<http://xifmij.github.io/csci121>

HOMWORK? LAB? HOW ARE THINGS?

- ▶ Don't forget to complete the **Homework 1** assignment:
 - due next Wednesday 9/8, before lecture
 - the description is at <https://xifmij.github.io/csci121/assign.html>
 - write several Python scripts much like today's examples

- ▶ Any questions from yesterday's lab? about Homework 1?

DROP-IN TUTORING; OFFICE HOURS

- **EVENING TUTORING:** Sunday through Thursdays, 7-9pm, ETC 208

→ *Starts tonight!*

- **MY OFFICE HOURS:**

10-11:20am Monday, 10-11:20am Wednesday

Also generally in my office 10-1:10 Monday and Wednesday

I'm also available on Zoom by appointment on Fridays.

LAB SWITCHING

- ▶ Some of you might still be seeking a different lab section
 - There may be some room for students in a different section.
 - Email Jim/Meaw about joining that section.
 - If you can, just bring us an **ADD/DROP** form from the Registrar.

CSCI 122

- ▶ If you have significant programming experience already, maybe email me about taking the 0.5 credit **CSCI 122**.

PYTHON SCRIPTING

- ▶ We start by looking at **Python scripting**:
 - A script is a text file containing lines of Python code.
 - Each line is a Python **statement**.
 - The Python **interpreter** (the `python3` command) executes each statement, line by line, from top to bottom.
 - A statement directs that an action be made by the interpreter, which has a *state-changing* effect.

PYTHON SCRIPTING (REVIEW)

Each Python statement directs that an action be taken, which has an effect on the *runtime system*.

▶ Some examples of effects:

- some text gets **output** (printed) to the *console*
- some typed console **input** is read
- some named **variable** gets assigned a newly computed value
- a window is displayed, a file is read, a URL's content is fetched, the program connects to a database or a network service, a noise is made, etc., etc.

INTERACTIVE SCRIPTS

- ▶ This program interacts with the program's user:

```
name = input("Could someone volunteer their name? ")
print("Hello there, " + name + "!")
print("Thanks for volunteering like that.")
print("This is our seventh Python program.")
```

- ▶ Here is one such interaction within Terminal:

```
C02MX1KLFH04:examples jimfix$ python3 shoutout.py
Could someone volunteer their name? Audrey Bilger
Hello there, Audrey Bilger!
Thanks for volunteering like that.
This is our seventh Python program.
C02MX1KLFH04:examples jimfix$
```

- ▶ The program has an assignment statement followed by 3 print statements.
- ▶ The assignment's right hand side uses a function named **input**
- ▶ That function first outputs a **prompt string** to the console...
 - And then it reads a **string of input** typed into the console.

INTERACTIVE SCRIPTS

- ▶ This program interacts with the program's user:

```
name = input("Could someone volunteer their name? ")
print("Hello there, " + name + "!")
print("Thanks for volunteering like that.")
print("This is our seventh Python program.")
```

- ▶ Here is one such interaction within Terminal:

```
C02MX1KLFH04:examples jimfix$ python3 shoutout.py
Could someone volunteer their name? Audrey Bilger
Hello there, Audrey Bilger!
Thanks for volunteering like that.
This is our seventh Python program.
C02MX1KLFH04:examples jimfix$
```

- ▶ The program has an assignment statement followed by 3 print statements.
- ▶ The assignment's right hand side uses a function named **input**
- ▶ That function first outputs a **prompt string** to the console...
 - And then it reads a **string of input** typed into the console.

STRING ARITHMETIC

▶ Another sample program:

```
name = input("Could someone volunteer their name? ")
print("Hello there, "+name+"!")
print("Thanks for volunteering like that.")
print("This is our eighth Python program.")
repeated = (name + ", ") * 3 + name
print("Below is your name repeated four times:")
print(repeated)
```

▶ Its execution within Terminal

```
C02MX1KLFH04:examples jimfix$ python3 shoutout4x.py
Could someone volunteer their name? Audrey Bilger
Hello there, Audrey Bilger!
Thanks for volunteering like that.
This is our eighth Python program.
Below is your name repeated four times:
Audrey Bilger, Audrey Bilger, Audrey Bilger, Audrey Bilger
C02MX1KLFH04:examples jimfix$
```

ANOTHER EXAMPLE

- ▶ Consider this Python program:

```
pi = 3.14159
area = float(input("Circle area? "))
radius = (area / pi) ** 0.5
print("The radius of that circle is "+str(radius)+" units.")
```

- ▶ This has is 3 assignment statements and a print statement.
- ▶ The first defines/assigns the variable named **pi**.
- ▶ The second gets a floating point value (a "calculator number") as input, assigned to **area**. We compute that using an arithmetic formula.
- ▶ The functions **float** and **str** convert values of one type to values of another type.

RECALL: PYTHON EXECUTION

- ▶ Let's take a look at this script:

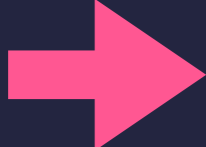
```
pi = 3.14159
area = float(input("Circle area? "))
radius = (area / pi) ** 0.5
print("That circle's radius is "+str(radius)+".")
```

- ▶ What we know is that the Python interpreter runs the code, line by line, from the top line to the bottom line.



RECALL: PYTHON EXECUTION

- ▶ Let's take a look at this script:



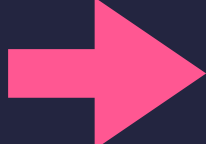
```
pi = 3.14159
area = float(input("Circle area? "))
radius = (area / pi) ** 0.5
print("That circle's radius is "+str(radius)+".")
```

- ▶ What we know is that the Python interpreter runs the code, **line by line**, from the top line to the bottom line.



RECALL: PYTHON EXECUTION

- ▶ Let's take a look at this script:



```
pi = 3.14159
area = float(input("Circle area? "))
radius = (area / pi) ** 0.5
print("That circle's radius is "+str(radius)+".")
```

- ▶ What we know is that the Python interpreter runs the code, **line by line**, from the top line to the bottom line.



RECALL: PYTHON EXECUTION

- ▶ Let's take a look at this script:

```
pi = 3.14159
area = float(input("Circle area? "))
radius = (area / pi) ** 0.5
print("That circle's radius is "+str(radius)+".")
```

- ▶ What we know is that the Python interpreter runs the code, **line by line**, from the top line to the bottom line.

RECALL: PYTHON EXECUTION

- ▶ Let's take a look at this script:

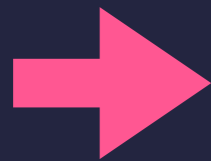
```
pi = 3.14159
area = float(input("Circle area? "))
radius = (area / pi) ** 0.5
print("That circle's radius is "+str(radius)+".")
```

- ▶ What we know is that the Python interpreter runs the code, **line by line**, from the top line to the bottom line.

RECALL: PYTHON EXECUTION

- ▶ Let's take a look at this script:

```
pi = 3.14159
area = float(input("Circle area? "))
radius = (area / pi) ** 0.5
print("That circle's radius is "+str(radius)+".")
```



- ▶ If you ever want to "watch" a Python program, try out **The Python Tutor**
<https://pythontutor.com/>
- ▶ *Using it, you'll see something like this...*

RECALL: PYTHON EXECUTION

- ▶ Let's take a look at this script:

global frame

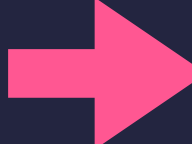
pi: 3.14159

```
➔ pi = 3.14159
   area = float(input("Circle area? "))
   radius = (area / pi) ** 0.5
   print("That circle's radius is "+str(radius)+".")
```

- ▶ What we know is that the Python interpreter runs the code, line by line, from the top line to the bottom line.
- ▶ It also creates **named memory slots** for each variable that gets introduced.
 - That named slot stores a calculated value.
 - A variable's associated value is changed with each assignment statement.

RECALL: PYTHON EXECUTION

- ▶ Let's take a look at this script:



```
pi = 3.14159
area = float(input("Circle area? "))
radius = (area / pi) ** 0.5
print("That circle's radius is "+str(radius)+".")
```

global frame



```
pi: 3.14159
area: 314.159
```

- ▶ What we know is that the Python interpreter runs the code, line by line, from the top line to the bottom line.
- ▶ It also creates **named memory slots** for each variable that gets introduced.
 - That named slot stores a calculated value.
 - A variable's associated value is **changed with each assignment statement**.

RECALL: PYTHON EXECUTION

- ▶ Let's take a look at this script:

```
pi = 3.14159
area = float(input("Circle area? "))
radius = (area / pi) ** 0.5
print("That circle's radius is "+str(radius)+".")
```

global frame

```
pi: 3.14159
area: 314.159
radius: 10.0
```

- ▶ What we know is that the Python interpreter runs the code, line by line, from the top line to the bottom line.
- ▶ It also creates **named memory slots** for each variable that gets introduced.
 - That named slot stores a calculated value.
 - A variable's associated value is changed with each assignment statement.

RECALL: PYTHON EXECUTION

- ▶ Let's take a look at this script:

```
pi = 3.14159
area = float(input("Circle area? "))
radius = (area / pi) ** 0.5
print("That circle's radius is "+str(radius)+".")
```

global frame

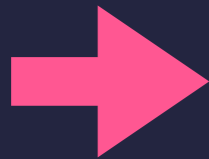
```
pi: 3.14159
area: 314.159
radius: 10.0
```

- ▶ What we know is that the Python interpreter runs the code, line by line, from the top line to the bottom line.
- ▶ It also creates **named memory slots** for each variable that gets introduced.
 - That named slot stores a calculated value.
 - A variable's associated value is changed with each assignment statement.

RECALL: PYTHON EXECUTION

- ▶ Let's take a look at this script:

```
pi = 3.14159
area = float(input("Circle area? "))
radius = (area / pi) ** 0.5
print("That circle's radius is "+str(radius)+".")
```



global frame

```
pi: 3.14159
area: 314.159
radius: 10.0
```

- ▶ What we know is that the Python interpreter runs the code, line by line, from the top line to the bottom line.
- ▶ It also creates named memory slots for each variable that gets introduced.
 - That named slot stores a calculated value.
 - A variable's associated value is changed with each assignment statement.
 - The collection of variable slots of a script is called the **global frame**

SAME, BUT DIFFERENT

- ▶ Consider this Python program:

```
from math import pi, sqrt
area = float(input("Circle area? "))
radius = sqrt(area / pi)
print("The radius of that circle is "+str(radius)+" units.")
```

- ▶ Here we **import** some definitions from a Python package named **math**.
- ▶ **pi** is the name of a floating point constant.
- ▶ **sqrt** is the name of a floating point function.
- ▶ There are packages for all sorts of useful Python libraries.

SOME ISSUES I'D LIKE TO ADDRESS

- ▶ values versus variables versus expressions
- ▶ functions, *calling* functions, *defining* functions (next week)
- ▶ different types: **int** versus **float** versus **str**
- ▶ operations on each type (and the “overloaded” meanings of **+** and *****)
- ▶ built-in functions for each type
- ▶ managing print output carefully
- ▶ special characters (tab, end of line, quote, ...)

Let's switch modes in how we use the Python interpreter...

INTERACTING WITH THE PYTHON INTERPRETER

- ▶ Python can be used to "live script":

```
C02MX1KLFH04:examples jimfix$ python3
>>> print("hello")
hello
>>> print(6 * 7)
42
>>> result = 6 * 7
>>> print(result)
42
>>>
```

- ▶ We can try a Python coding by interacting directly with the interpreter.
- ▶ We type in Python statements one at a time.
- ▶ Each line gets executed immediately.

THE INTERPRETER AS CALCULATOR

- ▶ Python can be used to evaluate expressions:

```
C02MX1KLFH04:examples jimfix$ python3
>>> "hello"
hello
>>> 6 * 7
42
>>> result = 6 * 7
>>> result
42
>>>
```

- ▶ We enter Python expressions instead.
 - Python evaluates it and shows its value on the next line.
- ▶ A Python statement describes an action to be performed.
- ▶ A Python expression describes a value to be calculated.
 - This evaluation is different than printing.

THE INTERPRETER AS CALCULATOR

- ▶ Python can be used to evaluate expressions:

```
C02MX1KLFH04:examples jimfix$ python3
```

```
>>> "hello"
```

```
hello
```

```
>>> 6 * 7
```

```
42
```

```
>>> result = 6 * 7
```

```
>>> result
```

```
42
```

```
>>>
```

- ▶ Here, Python is acting differently. It calculates the value of the expression, then (quietly) converts that to a string of text, then reports that text representing the value.

THE INTERPRETER AS CALCULATOR

- ▶ Python can be used to evaluate expressions:

```
C02MX1KLFH04:examples jimfix$ python3
>>> "hello"
hello
>>> 6 * 7
42
>>> result = 6 * 7
>>> result
42
>>>
```

- ▶ It follows three steps:
 - **READS**: it looks at the expression entered after >>>
 - **EVALUATES**: it performs that calculation, obtaining a value, including looking up variables' values
 - **PRINTS**: it converts that value to a string; displays it.

THE INTERPRETER AS CALCULATOR

- ▶ Python can be used to evaluate expressions:

```
C02MX1KLFH04:examples jimfix$ python3
```

```
>>> "hello"
```

```
hello
```

```
>>> 6 * 7
```

```
42
```

```
>>> result = 6 * 7
```

```
>>> result
```

```
42
```

```
>>>
```

- ▶ This is the "**READ - EVALUATE - PRINT LOOP**" (or "**REPL**").
- ▶ Having access to a **REPL** for a programming language is wonderful!
- ▶ It's a big reason we teach programming in Python.

PYTHON PROVIDES SOME USEFUL FUNCTIONS...

```
>>> pow(2,3)
8
>>> abs(-3)
3
>>> abs(4 + 2)
6
>>> min(3,7)
3
>>> max(4, 10.5 + 8.3, 6)
18.8
>>> from math import sqrt, pow
>>> sqrt(2.0)
1.4142135623730951
>>> pow(2.0,4.5)
22.627416997969522
```


PYTHON PROVIDES ARITHMETIC

```
>>> 3 + 7
10
>>> 4 + 2 * 3
10
>>> (4 + 2) * 3
18
>>> 4 / 16
0.25
>>> 2 ** 4
16
>>> 0.1 + 0.2
0.30000000000000004
```

PYTHON PROVIDES ARITHMETIC

```
>>> 3 + 7
10
>>> 4 + 2 * 3
10
>>> (4 + 2) * 3
18
>>> 4 / 16
0.25
>>> 2 ** 4
16
>>> 0.1 + 0.2
0.30000000000000004
>>> type(4)
<class 'int'>
>>> type(0.25)
<class 'float'>
```

INTEGERS VERSUS FLOATING POINT NUMBERS

- ▶ Python has two types of number values: `int` and `float`
- ▶ With integers, computation is exact.
- ▶ With floating point numbers ("*floats*"), computation is approximate.

```
>>> 10 / 2
```

```
5.0
```

```
>>> 3 + 4.0
```

```
7.0
```

```
>>> int(8.7)
```

```
8
```

INTEGER VERSUS FLOATING POINT DIVISION

- ▶ With the normal division operation, the slash `/`, you get a float.

```
>>> 10.2 / 2.0
```

```
5.1
```

```
>>> 10 / 2
```

```
5.0
```

```
>>> 10 // 2
```

```
5
```

```
>>> 87 / 10
```

```
8.7
```

```
>>> 87 // 10
```

```
8
```

- ▶ There is also an integer division operation, the double slash operator `//`.
 - This gives the integer quotient.
 - The remainder due to the division is discarded.

RECALL: LONG DIVISION

$$12 \overline{) 345}$$

RECALL: LONG DIVISION

$$\begin{array}{r} 2 \\ 12 \overline{) 345} \end{array}$$

RECALL: LONG DIVISION

$$\begin{array}{r} 2 \\ 12 \overline{) 345} \\ \underline{-24} \\ 105 \end{array}$$

RECALL: LONG DIVISION

$$\begin{array}{r} 28 \\ 12 \overline{) 345} \\ \underline{-24} \\ 105 \end{array}$$

RECALL: LONG DIVISION

$$\begin{array}{r} 28 \\ 12 \overline{) 345} \\ \underline{-24} \\ 105 \\ \underline{-96} \\ 9 \end{array}$$

RECALL: LONG DIVISION

$$\begin{array}{r} 28.9 \\ 12 \overline{) 345} \\ \underline{-24} \\ 105 \\ \underline{-96} \\ 9 \end{array}$$

← the quotient

RECALL: LONG DIVISION

A handwritten long division problem on a light blue background. The divisor is 12, and the dividend is 345. The quotient is 28, and the remainder is 9. The numbers are written in black ink. The quotient '28' is positioned above the dividend, and the remainder '9' is positioned below the final subtraction. The entire calculation is enclosed in a light blue rectangular box.

$$\begin{array}{r} 28 \\ 12 \overline{) 345} \\ \underline{-24} \\ 105 \\ \underline{-96} \\ 9 \end{array}$$

← the quotient

← the remainder

PYTHON HAS // AND % OPERATORS

- ▶ The `//` operation ("div") gives the integer quotient due to the division of two integers:

```
>>> 345 // 12
28
```

- ▶ The `%` operation ("mod") gives the integer remainder due to the division of two integers:

```
>>> 345 % 12
9
```

- ▶ This property always holds: $n == q * d + r$

```
>>> 28 * 12 + 9
345
```

EXAMPLE USES

```
>>> 345 % 10
?????????
>>> 345 // 10
?????????
>>> 6789 % 2
?????????
>>> 6790 % 2
?????????
>>> -26 % 2
?????????
>>> -76 % 10
?????????
>>> -26 // 2
?????????
>>> -76 // 10
?????????
```

EXAMPLE USES

```
>>> 345 % 10
```

```
5
```

```
>>> 345 // 10
```

```
34
```

```
>>> 6789 % 2
```

```
1
```

```
>>> 6790 % 2
```

```
0
```

```
>>> -26 % 2
```

```
0
```

```
>>> -76 % 10
```

```
4
```

```
>>> -26 // 2
```

```
-13
```

```
>>> -76 // 10
```

```
-8
```

SPECIAL CHARACTERS

- ▶ A backslash character `\` followed by a second character expresses special characters
 - a tab is `\t`, a new line is `\n`, a quote is `\'`, a backslash is `\\`

```
>>> z = input('What\'s your name? ')
What's your name?John
>>> x = " " + z
'Hello John'
>>> print("I\'ve " + str(19) + " characters.\nSee?")
I've 19 characters.
See?
>>> len("I\'ve " + str(19) + " characters.\nSee?")
19
>>> print("\thello\nthere")
hello
there
>>> print("/\\/\\/\\/\\/\\/\\/\\/")
/\\/\\/\\/\\
```

AN INFORMAL QUIZ

```
>>> z = 7
>>> x = 5 + z
>>> z = z + 1
>>> print(str(z) + str(z))
??????????
>>> 0.2 + 0.1
??????????
>>> 0.2 - 0.1
??????????
>>> len('Jim\'s example:\t done.\n')
??????????
>>> print("abc\n"*4)
??????????
??????????
...?
>>> "hello" - "llo"
??????????
```


SUMMARY

- ▶ So far, three kinds of statements:
 - **print** statement
 - assignment statement
 - **import** statement
- ▶ Several built-in functions
 - **input**
 - conversions: **str, int, float**
 - **abs, min, max, pow**, and many more from the **math** library
 - **len**
 - **type**

SUMMARY (CONT'D)

▶ Binary operations (so far)

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

SUMMARY (CONT'D)

- ▶ The Python interpreter can be run *interactively* or *not*.
 - When **interactive**, you type in a statement or an expression.
 - When a statement is entered, it gets executed.
 - ◆ If there is any output, it appears on subsequent lines.
 - When an expression is entered, it gets evaluated.
 - ◆ The value that results is displayed on the next line.
 - When **not interactive**, Python just loads and runs a script.
 - Its code is executed, line by line (statement followed by statement).

READINGS; NEXT WEEK

- ▶ This week's lecture material can be supplemented with:
 - **Reading:** TP Ch. 1 and 2; CP Ch 1.1-1.2
- ▶ Next week we'll
 - try the conditional statement (i.e. **if**) in Tuesday's lab
 - define functions (i.e. **def** ...) in Wednesday's lecture
 - **Reading:**
 - ◆ TP Ch. 3, 6 (functions); TP Chs 4.1-4.8 (conditionals)
 - ◆ CP 1.3-1.4

READINGS; NEXT WEEK

▶ This week's lecture material can be supplemented with:

- **Reading:** TP Ch. 1 and 2, CP Ch 1.1-1.2

▶ Next week we'll

- try the conditional statement (i.e. **if**) in Tuesday's lab
- define functions (i.e. **def** ...) in Wednesday's lecture

• **Reading:**

- ◆ TP Ch. 3, 6 (functions); TP Chs 4.1-4.8 (conditionals)
- ◆ CP 1.3-1.4

READINGS; NEXT WEEK

▶ This week's lecture material can be supplemented with:

- **Reading:** TP Ch. 1 and 2, CP Ch 1.1-1.2

▶ Next week we'll

"Composing Programs" text

→ try the conditional statement (i.e. **if**) in Tuesday's lab

→ define functions (i.e. **def** ...) in Wednesday's lecture

• **Reading:**

- ◆ TP Ch. 3, 6 (functions); TP Chs 4.1-4.8 (conditionals)
- ◆ CP 1.3-1.4

▶ **No lecture MONDAY.** *Happy labor day!*

"FLOW OF CONTROL"

Recall: our animation of the "*circle area to radius*" calculation...

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*.
- ▶ When write code with *conditional statements* and *loops*, we'll see program flow that's not just top to bottom.
 - Lines might get repeatedly executed, or lines might get skipped.

"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 an 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.

"LOOPING"

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

```
pi = 3.14159
area = float(input("Circle area? "))
while area < 0.0:
    area = float(input("Not an area. 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.

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

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))
```

- ▶ Below is it in use:

```
% 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. $-5.5 \sim > 5.5$
- ▶ Otherwise, if positive or **0.0**, it just prints that value.

SYNTAX: IF-ELSE STATEMENT

Below gives 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

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 **return** statement.
 - If the condition is **False**, it executes the second **return** 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 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 gives 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
```

- ▶ Like function **def**, we use indentation to indicate the "true" block of code and the "false" block of code.

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))
```

- ▶ 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!

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")
```

- ▶ 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: -10
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

EXPRESSING COMPLEX CONDITIONS

- ▶ The function 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 function 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 function 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 function 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 is described *logical disjunction*.
- ▶ There is also logical negation using **not**.

READINGS; NEXT WEEK

▶ This week's lecture material can be supplemented with:

- **Reading:** TP Ch. 1 and 2, CP Ch 1.1-1.2

▶ Next week we'll

"Composing Programs" text

→ try the conditional statement (i.e. **if**) in Tuesday's lab

→ define functions (i.e. **def** ...) in Wednesday's lecture

• **Reading:**

- ◆ TP Ch. 3, 6 (functions); TP Chs 4.1-4.8 (conditionals)
- ◆ CP 1.3-1.4

▶ **No lecture MONDAY.** *Happy labor day!*