PYTHON SCRIPTING (CONT'D) CONTROL FLOW, CONDITIONS

LECTURE 01-2

JIM FIX, REED COLLEGE CSCI 121

HOMEWORK? HOW ARE THINGS?

- Don't forget to complete **Homework 0** assignment:
  - due this Sunday 1/29 by midnight
  - the description is at https://nchanath.github.io/121-S23 under tab “Homework”
  - write several Python scripts much like today's examples
- Any questions for Homework 0?

DROP-IN TUTORING; OFFICE HOURS

- **EVENING TUTORING**: Sunday through Thursdays, 7-9pm, ETC 208
  - ***Starts tonight!***
- **MY OFFICE HOURS**:
  - TBA
  - Also, generally in my office 10-3:00 MWF

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.

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:

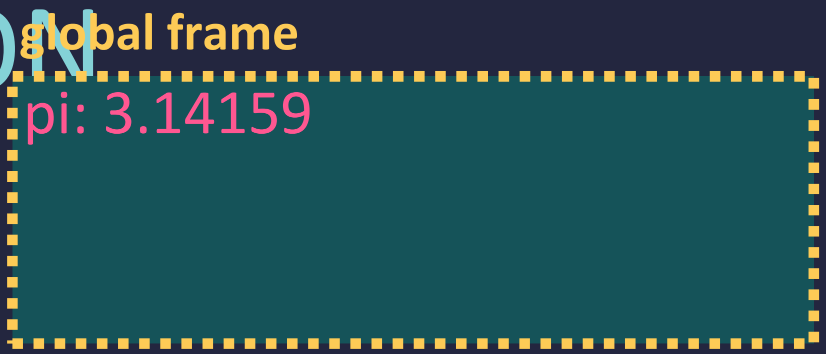
```
pi = 3.14159
area = float(input("Circle area? "))
radius = (area / pi) ** 0.5
print("That circle’s radius is "+str(radius)+".")
```

- 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

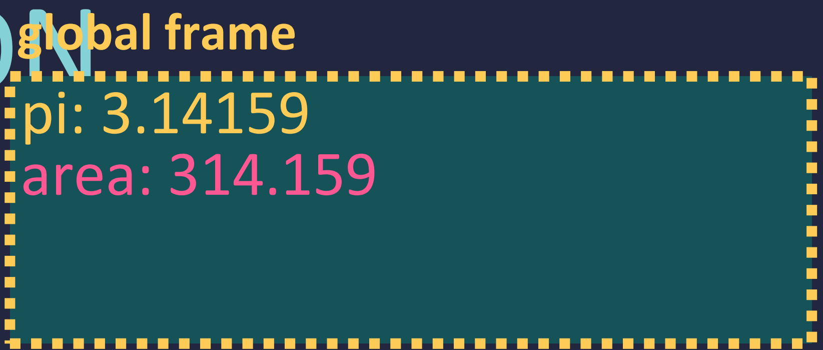


- Let's take a look at this script:

```
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)+".")
```

- 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

global frame

pi: 3.14159
area: 314.159
radius: 10.0

- 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.
- 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

global frame

pi: 3.14159
area: 314.159
radius: 10.0

- 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.
- 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

global frame

pi: 3.14159
area: 314.159
radius: 10.0

- 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.
- 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***

"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))
```

- 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` ~> `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
```

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))
```

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!

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 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 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 described *logical disjunction*.
- There is also logical negation using `not`.

"Think Python" text

READINGS; NEXT WEEK

- This week's lecture material can be supplemented with:
  - **Reading**: TP Ch. 1 and 2; CP Ch 1.1-1.2

"Composing Programs" text

- 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