MORE PYTHON

LECTURE 01–2:
EXECUTING A PYTHON SCRIPT
HOURS & MINUTES EXERCISE

JIM FIX, REED COLLEGE CSCI 121

SUMMARY OF PYTHON SO FAR

- 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).

SUMMARY OF PYTHON SO FAR

- So far, three kinds of statements:
  - `print` statement
  - assignment statement
  - `import` statement

SUMMARY OF PYTHON SO FAR

- So far, ~~three~~ four (um, *ish*) kinds of statements:
  - **`print`** statement
  - assignment statement
  - **`import`** statement

- the "input statement," an assignment statement like one of these below.
  - to get an integer input from the program's user:

    variable `=` `int(input(`***prompt-string***`))`
  - to get a floating point value…

    variable `=` `float(input(`***prompt-string***`))`
  - to get a string of text…

    variable `=` `input(`***prompt-string***`)`

BACK TO PYTHON SCRIPTING

- 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.")
```

BACK TO PYTHON SCRIPTING

- 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

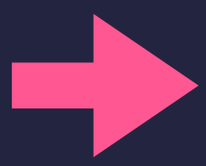
- Let's take a look at execution of this script:

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

- 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:

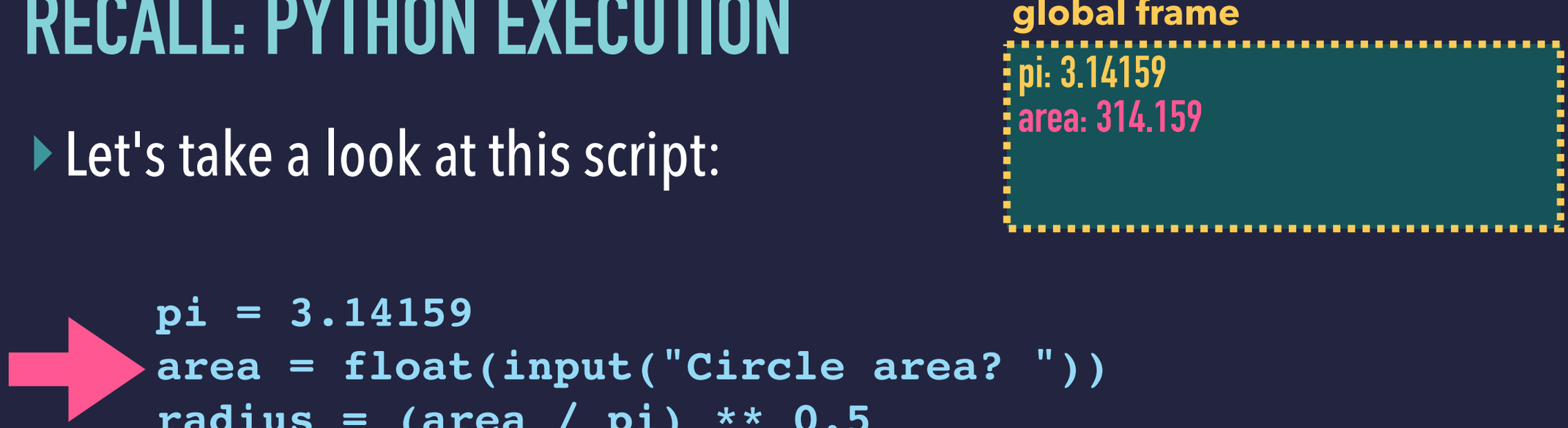


```
pi = 3.14159
area = float(input("Circle area? "))
radius = (area / pi) ** 0.5
print("That circle's radius is "+str(radius)+".")
```

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

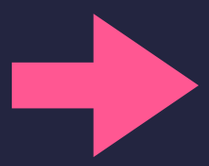



```
pi = 3.14159
area = float(input("Circle area? "))
radius = (area / pi) ** 0.5
print("That circle's radius is "+str(radius)+".")
```

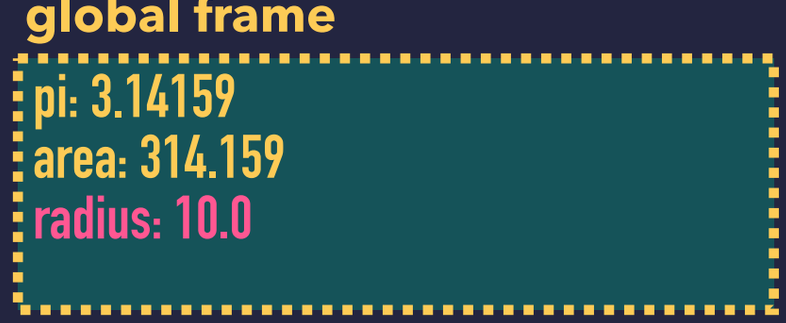
- What we know is that the Python interpreter runs the code, line by line, from the top line to the bottom line.
- It also creates named memory slots for each variable that gets introduced.
  - That named slot stores a calculated value.
  - A variable's value is changed with each assignment statement.

RECALL: PYTHON EXECUTION

- Let's take a look at this script:



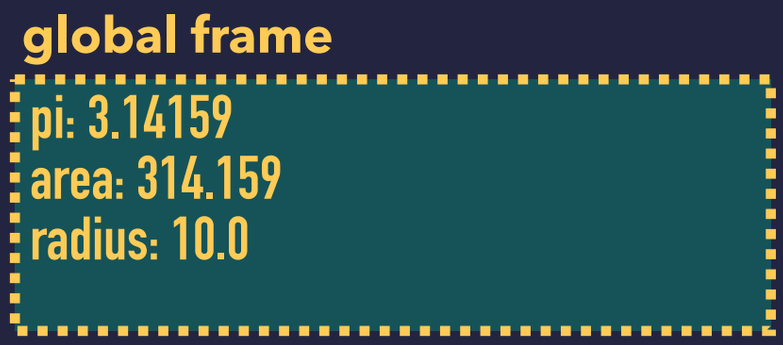
```
pi = 3.14159
area = float(input("Circle area? "))
radius = (area / pi) ** 0.5
print("That circle's radius is "+str(radius)+".")
```



- 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 value is changed with each assignment statement.

RECALL: PYTHON EXECUTION

- Let's take a look at this script:



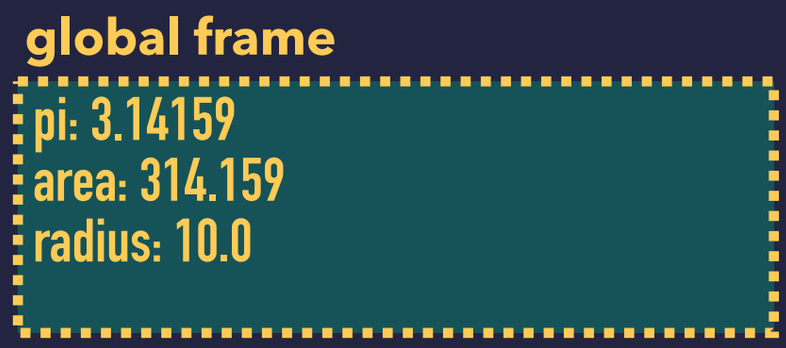

```
pi = 3.14159
area = float(input("Circle area? "))
radius = (area / pi) ** 0.5
print("That circle's radius is "+str(radius)+".")
```

- 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 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)+".")
```

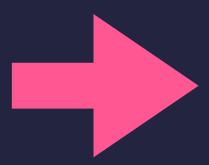


- What we know is that the Python interpreter runs the code, line by line, from the top line to the bottom line.
- It also creates named memory slots for each variable that gets introduced.
  - That named slot stores a calculated value.
  - A variable's value is changed with each assignment statement.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

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/

IMPORTS

- 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)+".")
```

IMPORTS

- Let's take a look at this script:

```
from math import pi, sqrt
area = float(input("Circle area? "))
radius = sqrt(area / pi)
print("That circle's radius is "+str(radius)+".")
```

SUMMARY OF PYTHON SO FAR

- 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)

- Binary operations (so far)
  - for integers: `+` `-` `*` `//` `%` `**`
  - for floats: `+` `-` `*` `/` `**`
  - for strings: `+` `*`

SUMMARY (CONT'D)

- Binary operations (so far)
  - for integers: `+` `-` `*` `//` `%` `**`
  - for floats: `+` `-` `*` `/` `**`
  - for strings: `+` `*`

EXERCISE: 24 HOUR CLOCK

- Let's write a script that does the following:

> *Compute the time for a 24-hour clock after some number of minutes have passed from the current time.*

REPORT AS 24-HOUR CLOCK

- Here is that full script:

```
hours = int(input("Hours on the clock? "))
minutes = int(input("Minutes on the clock? "))
time = hours * 60 + minutes
time_passes = int(input("Minutes that pass? "))
time = time + time_passes

hours = time // 60 % 24
minutes = time % 60

print("The clock reads: ", end="")
print(hours, "hours and", minutes, "minutes", end="")
print(".")
```

REPORT AS 24-HOUR CLOCK

- Here is that full script:

```
hours = int(input("Hours on the clock? "))
minutes = int(input("Minutes on the clock? "))
time = hours * 60 + minutes
time_passes = int(input("Minutes that pass? "))
time = time + time_passes

hh = str(time // 60 % 24)
mm = str(time % 60)

hh = "0" * (2-len(hh)) + hh
mm = "0" * (2-len(mm)) + mm

print(hh + ":" + mm)
```

REPORT AS 24-HOUR CLOCK

- Python execution:

```
hours = int(input("Hours on the clock? "))
minutes = int(input("Minutes on the clock? "))
time = hours * 60 + minutes
time_passes = int(input("Minutes that pass? "))
time = time + time_passes
hh = str(time // 60 % 24)
mm = str(time % 60)
hh = "0" * (2-len(hh)) + hh
mm = "0" * (2-len(mm)) + mm
print(hh + ":" + mm)
```

- Console interaction:

```
terminal% python3 clock24.py
```

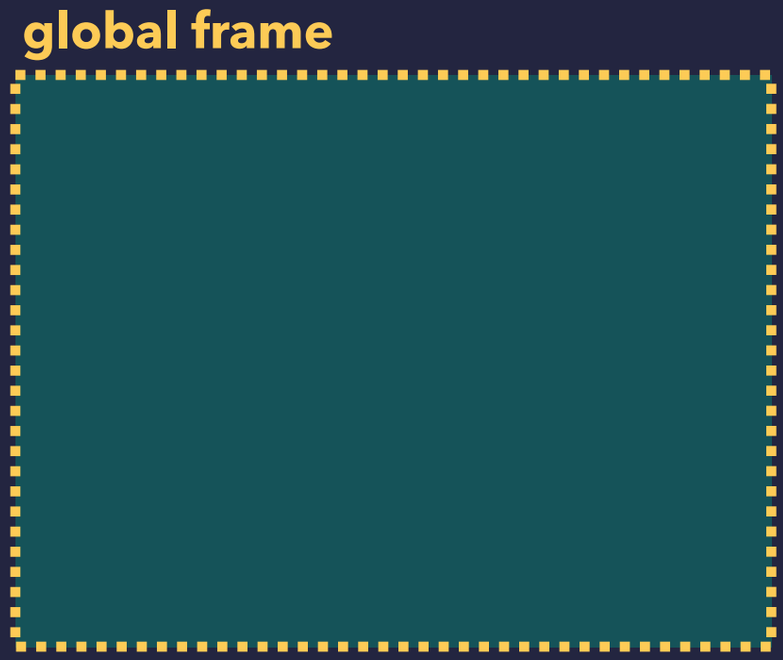
REPORT AS 24-HOUR CLOCK

- Python execution:

```
hours = int(input("Hours on the clock? "))
minutes = int(input("Minutes on the clock? "))
time = hours * 60 + minutes
time_passes = int(input("Minutes that pass? "))
time = time + time_passes
hh = str(time // 60 % 24)
mm = str(time % 60)
hh = "0" * (2-len(hh)) + hh
mm = "0" * (2-len(mm)) + mm
print(hh + ":" + mm)
```

- Console interaction:

```
terminal% python3 clock24.py
```



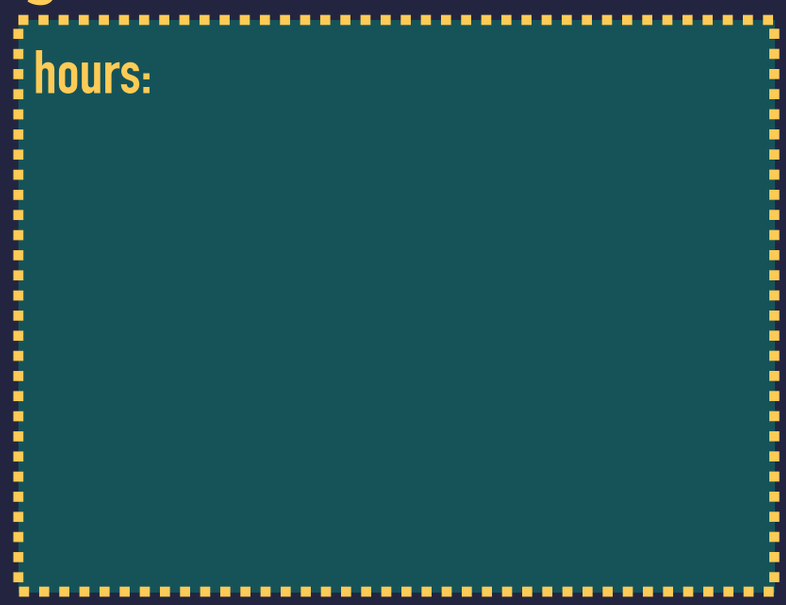
REPORT AS 24-HOUR CLOCK

- Python execution:

```
hours = int(input("Hours on the clock? "))
minutes = int(input("Minutes on the clock? "))
time = hours * 60 + minutes
time_passes = int(input("Minutes that pass? "))
time = time + time_passes
hh = str(time // 60 % 24)
mm = str(time % 60)
hh = "0" * (2-len(hh)) + hh
mm = "0" * (2-len(mm)) + mm
print(hh + ":" + mm)
```

- Console interaction:

```
terminal% python3 clock24.py
Hours on the clock?
```



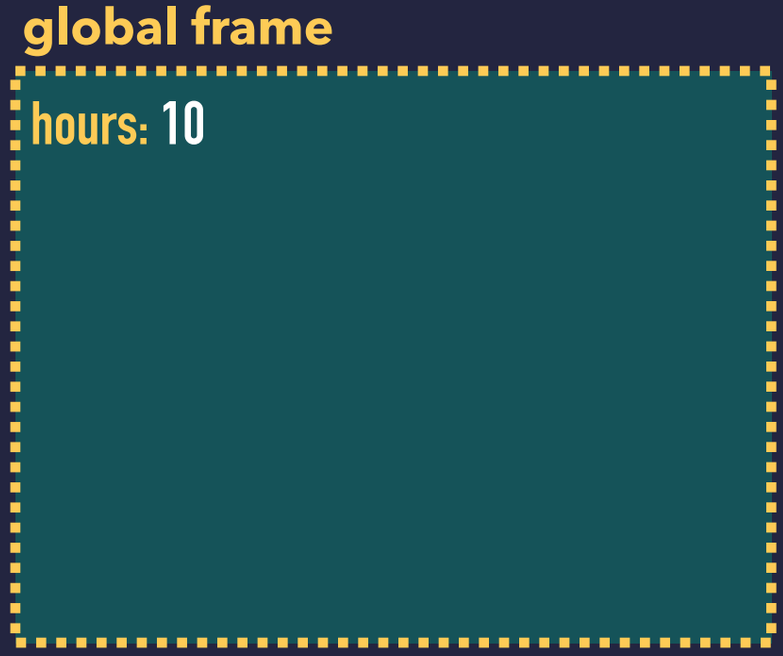
REPORT AS 24-HOUR CLOCK

- Python execution:

```
hours = int(input("Hours on the clock? "))
minutes = int(input("Minutes on the clock? "))
time = hours * 60 + minutes
time_passes = int(input("Minutes that pass? "))
time = time + time_passes
hh = str(time // 60 % 24)
mm = str(time % 60)
hh = "0" * (2-len(hh)) + hh
mm = "0" * (2-len(mm)) + mm
print(hh + ":" + mm)
```

- Console interaction:

```
terminal% python3 clock24.py
Hours on the clock? 10
```



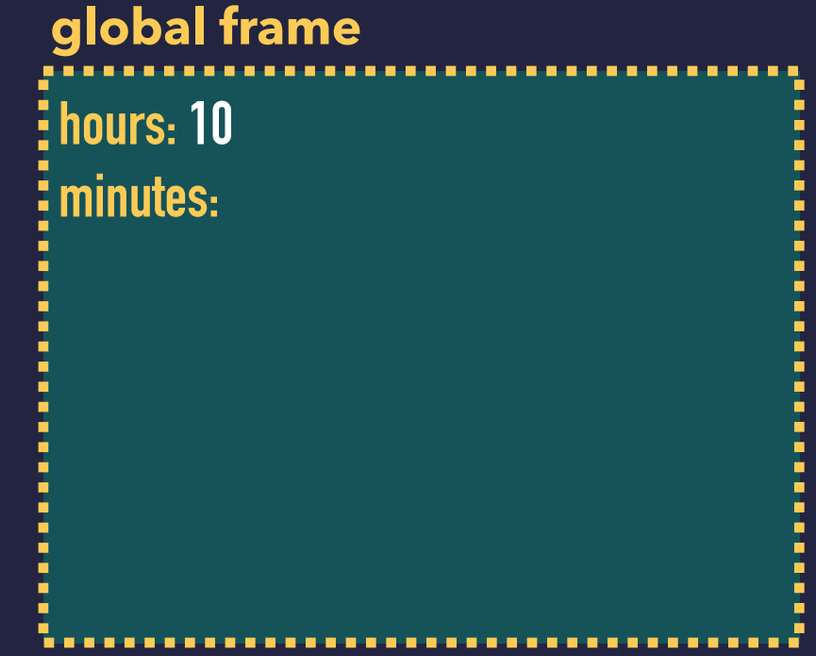
REPORT AS 24-HOUR CLOCK

- Python execution:

```
hours = int(input("Hours on the clock? "))
minutes = int(input("Minutes on the clock? "))
time = hours * 60 + minutes
time_passes = int(input("Minutes that pass? "))
time = time + time_passes
hh = str(time // 60 % 24)
mm = str(time % 60)
hh = "0" * (2-len(hh)) + hh
mm = "0" * (2-len(mm)) + mm
print(hh + ":" + mm)
```

- Console interaction:

```
terminal% python3 clock24.py
Hours on the clock? 10
Minutes on the clock?
```



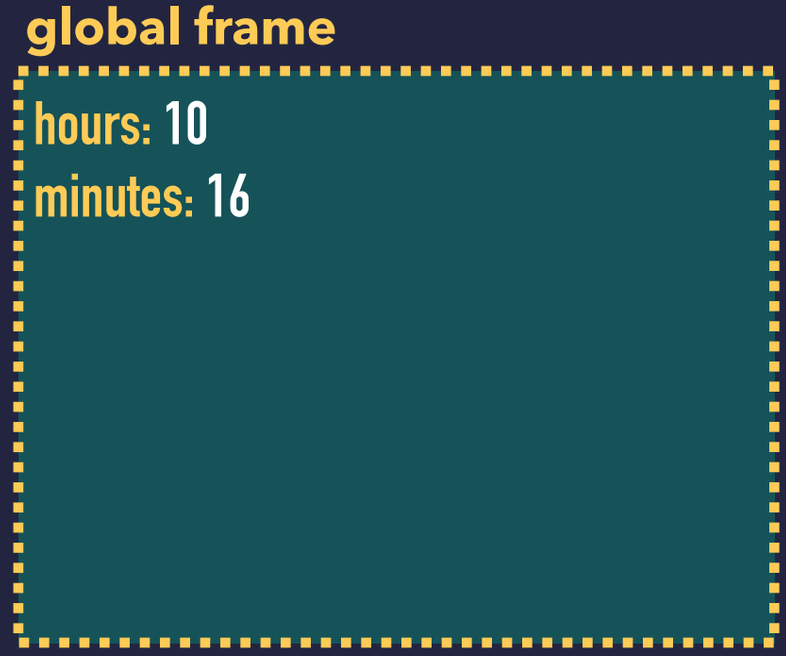
REPORT AS 24-HOUR CLOCK

- Python execution:

```
hours = int(input("Hours on the clock? "))
minutes = int(input("Minutes on the clock? "))
time = hours * 60 + minutes
time_passes = int(input("Minutes that pass? "))
time = time + time_passes
hh = str(time // 60 % 24)
mm = str(time % 60)
hh = "0" * (2-len(hh)) + hh
mm = "0" * (2-len(mm)) + mm
print(hh + ":" + mm)
```

- Console interaction:

```
terminal% python3 clock24.py
Hours on the clock? 10
Minutes on the clock? 16
```



REPORT AS 24-HOUR CLOCK

- Python execution:

```
hours = int(input("Hours on the clock? "))
minutes = int(input("Minutes on the clock? "))
time = hours * 60 + minutes
time_passes = int(input("Minutes that pass? "))
time = time + time_passes
hh = str(time // 60 % 24)
mm = str(time % 60)
hh = "0" * (2-len(hh)) + hh
mm = "0" * (2-len(mm)) + mm
print(hh + ":" + mm)
```

- Console interaction:

```
terminal% python3 clock24.py
Hours on the clock? 10
Minutes on the clock? 16
```

global frame

hours: 10
minutes: 16
time:

REPORT AS 24-HOUR CLOCK

- Python execution:

```
hours = int(input("Hours on the clock? "))
minutes = int(input("Minutes on the clock? "))
time = hours * 60 + minutes
time_passes = int(input("Minutes that pass? "))
time = time + time_passes
hh = str(time // 60 % 24)
mm = str(time % 60)
hh = "0" * (2-len(hh)) + hh
mm = "0" * (2-len(mm)) + mm
print(hh + ":" + mm)
```

- Console interaction:

```
terminal% python3 clock24.py
Hours on the clock? 10
Minutes on the clock? 16
```

global frame

hours: 10
minutes: 16
time: 616

REPORT AS 24-HOUR CLOCK

- Python execution:

```
hours = int(input("Hours on the clock? "))
minutes = int(input("Minutes on the clock? "))
time = hours * 60 + minutes
time_passes = int(input("Minutes that pass? "))
time = time + time_passes
hh = str(time // 60 % 24)
mm = str(time % 60)
hh = "0" * (2-len(hh)) + hh
mm = "0" * (2-len(mm)) + mm
print(hh + ":" + mm)
```

- Console interaction:

```
terminal% python3 clock24.py
Hours on the clock? 10
Minutes on the clock? 16
Minutes that pass?
```

global frame

hours: 10
minutes: 16
time: 616
time_passes:

REPORT AS 24-HOUR CLOCK

- Python execution:

```
hours = int(input("Hours on the clock? "))
minutes = int(input("Minutes on the clock? "))
time = hours * 60 + minutes
time_passes = int(input("Minutes that pass? "))
time = time + time_passes
hh = str(time // 60 % 24)
mm = str(time % 60)
hh = "0" * (2-len(hh)) + hh
mm = "0" * (2-len(mm)) + mm
print(hh + ":" + mm)
```

- Console interaction:

```
terminal% python3 clock24.py
Hours on the clock? 10
Minutes on the clock? 16
Minutes that pass? 51
```

global frame

hours: 10
minutes: 16
time: 616
time_passes: 51

REPORT AS 24-HOUR CLOCK

- Python execution:

```
hours = int(input("Hours on the clock? "))
minutes = int(input("Minutes on the clock? "))
time = hours * 60 + minutes
time_passes = int(input("Minutes that pass? "))
time = time + time_passes
hh = str(time // 60 % 24)
mm = str(time % 60)
hh = "0" * (2-len(hh)) + hh
mm = "0" * (2-len(mm)) + mm
print(hh + ":" + mm)
```

- Console interaction:

```
terminal% python3 clock24.py
Hours on the clock? 10
Minutes on the clock? 16
Minutes that pass? 51
```

global frame

hours: 10
minutes: 16
time: 616
time_passes: 51

REPORT AS 24-HOUR CLOCK

- Python execution:

```
hours = int(input("Hours on the clock? "))
minutes = int(input("Minutes on the clock? "))
time = hours * 60 + minutes
time_passes = int(input("Minutes that pass? "))
time = time + time_passes
hh = str(time // 60 % 24)
mm = str(time % 60)
hh = "0" * (2-len(hh)) + hh
mm = "0" * (2-len(mm)) + mm
print(hh + ":" + mm)
```

- Console interaction:

```
terminal% python3 clock24.py
Hours on the clock? 10
Minutes on the clock? 16
Minutes that pass? 51
```

global frame

hours: 10
minutes: 16
time: 667
time_passes: 51

REPORT AS 24-HOUR CLOCK

- Python execution:

```
hours = int(input("Hours on the clock? "))
minutes = int(input("Minutes on the clock? "))
time = hours * 60 + minutes
time_passes = int(input("Minutes that pass? "))
time = time + time_passes
hh = str(time // 60 % 24)
mm = str(time % 60)
hh = "0" * (2-len(hh)) + hh
mm = "0" * (2-len(mm)) + mm
print(hh + ":" + mm)
```

- Console interaction:

```
terminal% python3 clock24.py
Hours on the clock? 10
Minutes on the clock? 16
Minutes that pass? 51
```

global frame

hours: 10
minutes: 16
time: 667
time_passes: 51
hh:

REPORT AS 24-HOUR CLOCK

- Python execution:

```
hours = int(input("Hours on the clock? "))
minutes = int(input("Minutes on the clock? "))
time = hours * 60 + minutes
time_passes = int(input("Minutes that pass? "))
time = time + time_passes
hh = str(time // 60 % 24)
mm = str(time % 60)
hh = "0" * (2-len(hh)) + hh
mm = "0" * (2-len(mm)) + mm
print(hh + ":" + mm)
```

- Console interaction:

```
terminal% python3 clock24.py
Hours on the clock? 10
Minutes on the clock? 16
Minutes that pass? 51
```

global frame

hours: 10
minutes: 16
time: 667
time_passes: 51
hh: " 11 "

REPORT AS 24-HOUR CLOCK

- Python execution:

```
hours = int(input("Hours on the clock? "))
minutes = int(input("Minutes on the clock? "))
time = hours * 60 + minutes
time_passes = int(input("Minutes that pass? "))
time = time + time_passes
hh = str(time // 60 % 24)
mm = str(time % 60)
hh = "0" * (2-len(hh)) + hh
mm = "0" * (2-len(mm)) + mm
print(hh + ":" + mm)
```

- Console interaction:

```
terminal% python3 clock24.py
Hours on the clock? 10
Minutes on the clock? 16
Minutes that pass? 51
```

global frame

hours: 10
minutes: 16
time: 667
time_passes: 51
hh: " 11 "
mm:

REPORT AS 24-HOUR CLOCK

- Python execution:

```
hours = int(input("Hours on the clock? "))
minutes = int(input("Minutes on the clock? "))
time = hours * 60 + minutes
time_passes = int(input("Minutes that pass? "))
time = time + time_passes
hh = str(time // 60 % 24)
mm = str(time % 60)
hh = "0" * (2-len(hh)) + hh
mm = "0" * (2-len(mm)) + mm
print(hh + ":" + mm)
```

- Console interaction:

```
terminal% python3 clock24.py
Hours on the clock? 10
Minutes on the clock? 16
Minutes that pass? 51
```

global frame

hours: 10
minutes: 16
time: 667
time_passes: 51
hh: " 11 "
mm: " 7 "

REPORT AS 24-HOUR CLOCK

- Python execution:

```
hours = int(input("Hours on the clock? "))
minutes = int(input("Minutes on the clock? "))
time = hours * 60 + minutes
time_passes = int(input("Minutes that pass? "))
time = time + time_passes
hh = str(time // 60 % 24)
mm = str(time % 60)
hh = "0" * (2-len(hh)) + hh
mm = "0" * (2-len(mm)) + mm
print(hh + ":" + mm)
```

- Console interaction:

```
terminal% python3 clock24.py
Hours on the clock? 10
Minutes on the clock? 16
Minutes that pass? 51
```

global frame

hours: 10
minutes: 16
time: 667
time_passes: 51
hh: " 11 "
mm: " 7 "

REPORT AS 24-HOUR CLOCK

- Python execution:

```
hours = int(input("Hours on the clock? "))
minutes = int(input("Minutes on the clock? "))
time = hours * 60 + minutes
time_passes = int(input("Minutes that pass? "))
time = time + time_passes
hh = str(time // 60 % 24)
mm = str(time % 60)
hh = "0" * (2-len(hh)) + hh
mm = "0" * (2-len(mm)) + mm
print(hh + ":" + mm)
```

- Console interaction:

```
terminal% python3 clock24.py
Hours on the clock? 10
Minutes on the clock? 16
Minutes that pass? 51
```

global frame

hours: 10
minutes: 16
time: 667
time_passes: 51
hh: " 11 "
mm: " 7 "

REPORT AS 24-HOUR CLOCK

- Python execution:

```
hours = int(input("Hours on the clock? "))
minutes = int(input("Minutes on the clock? "))
time = hours * 60 + minutes
time_passes = int(input("Minutes that pass? "))
time = time + time_passes
hh = str(time // 60 % 24)
mm = str(time % 60)
hh = "0" * (2-len(hh)) + hh
mm = "0" * (2-len(mm)) + mm
print(hh + ":" + mm)
```

- Console interaction:

```
terminal% python3 clock24.py
Hours on the clock? 10
Minutes on the clock? 16
Minutes that pass? 51
```

global frame

hours: 10
minutes: 16
time: 667
time_passes: 51
hh: " 11 "
mm: " 7 "

REPORT AS 24-HOUR CLOCK

- Python execution:

```
hours = int(input("Hours on the clock? "))
minutes = int(input("Minutes on the clock? "))
time = hours * 60 + minutes
time_passes = int(input("Minutes that pass? "))
time = time + time_passes
hh = str(time // 60 % 24)
mm = str(time % 60)
hh = "0" * (2-len(hh)) + hh
mm = "0" * (2-len(mm)) + mm
print(hh + ":" + mm)
```

- Console interaction:

```
terminal% python3 clock24.py
Hours on the clock? 10
Minutes on the clock? 16
Minutes that pass? 51
```

global frame

hours: 10
minutes: 16
time: 667
time_passes: 51
hh: " 11 "
mm: " 07 "

REPORT AS 24-HOUR CLOCK

- Python execution:

```
hours = int(input("Hours on the clock? "))
minutes = int(input("Minutes on the clock? "))
time = hours * 60 + minutes
time_passes = int(input("Minutes that pass? "))
time = time + time_passes
hh = str(time // 60 % 24)
mm = str(time % 60)
hh = "0" * (2-len(hh)) + hh
mm = "0" * (2-len(mm)) + mm
print(hh + ":" + mm)
```

- Console interaction:

```
terminal% python3 clock24.py
Hours on the clock? 10
Minutes on the clock? 16
Minutes that pass? 51
```

global frame

hours: 10
minutes: 16
time: 667
time_passes: 51
hh: " 11 "
mm: " 07 "

REPORT AS 24-HOUR CLOCK

- Python execution:

```
hours = int(input("Hours on the clock? "))
minutes = int(input("Minutes on the clock? "))
time = hours * 60 + minutes
time_passes = int(input("Minutes that pass? "))
time = time + time_passes
hh = str(time // 60 % 24)
mm = str(time % 60)
hh = "0" * (2-len(hh)) + hh
mm = "0" * (2-len(mm)) + mm
print(hh + ":" + mm)
```

- Console interaction:

```
terminal% python3 clock24.py
Hours on the clock? 10
Minutes on the clock? 16
Minutes that pass? 51
11:07
```

global frame

hours: 10
minutes: 16
time: 667
time_passes: 51
hh: " 11 "
mm: " 07 "

REPORT AS 24-HOUR CLOCK

- Python execution:

```
hours = int(input("Hours on the clock? "))
minutes = int(input("Minutes on the clock? "))
time = hours * 60 + minutes
time_passes = int(input("Minutes that pass? "))
time = time + time_passes
hh = str(time // 60 % 24)
mm = str(time % 60)
hh = "0" * (2-len(hh)) + hh
mm = "0" * (2-len(mm)) + mm
print(hh + ":" + mm)
```

- Console interaction:

```
terminal% python3 clock24.py
Hours on the clock? 10
Minutes on the clock? 16
Minutes that pass? 51
11:07
terminal%
```

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

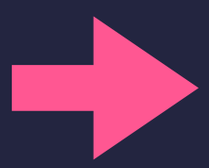
- Let's take a look at this script:

```
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:

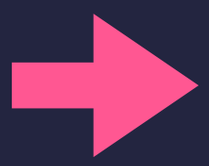


```
pi = 3.14159
area = float(input("Circle area? "))
radius = (area / pi) ** 0.5
print("That circle's radius is "+str(radius)+".")
```

- 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:

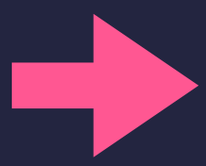
```
pi = 3.14159
area = float(input("Circle area? "))
radius = (area / pi) ** 0.5
print("That circle's radius is "+str(radius)+".")
```

- 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:

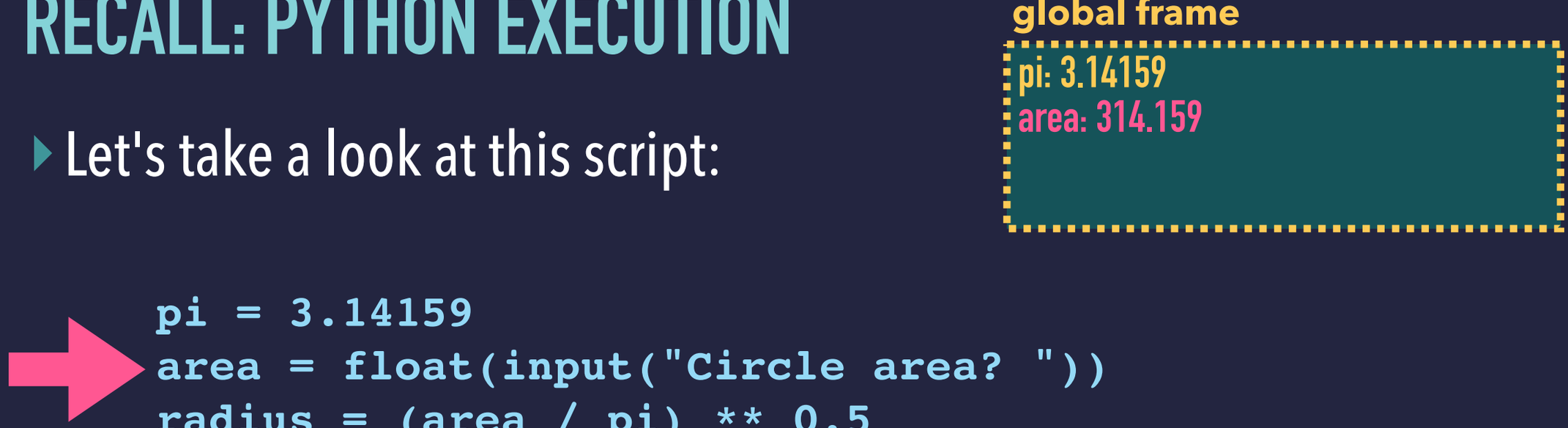


```
pi = 3.14159
area = float(input("Circle area? "))
radius = (area / pi) ** 0.5
print("That circle's radius is "+str(radius)+".")
```

- 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:

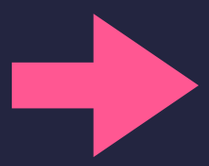



```
pi = 3.14159
area = float(input("Circle area? "))
radius = (area / pi) ** 0.5
print("That circle's radius is "+str(radius)+".")
```

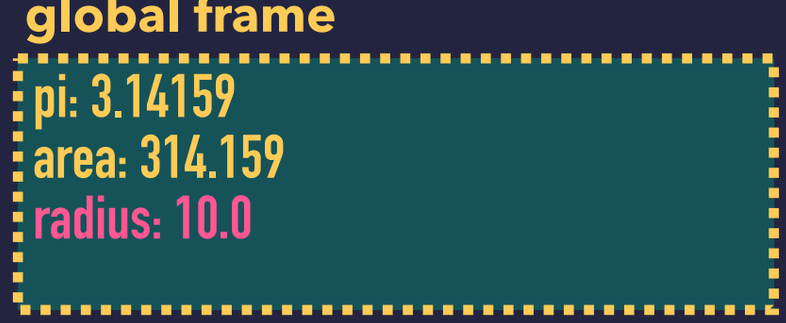
- What we know is that the Python interpreter runs the code, line by line, from the top line to the bottom line.
- It also creates named memory slots for each variable that gets introduced.
  - That named slot stores a calculated value.
  - A variable's value is changed with each assignment statement.

RECALL: PYTHON EXECUTION

- Let's take a look at this script:



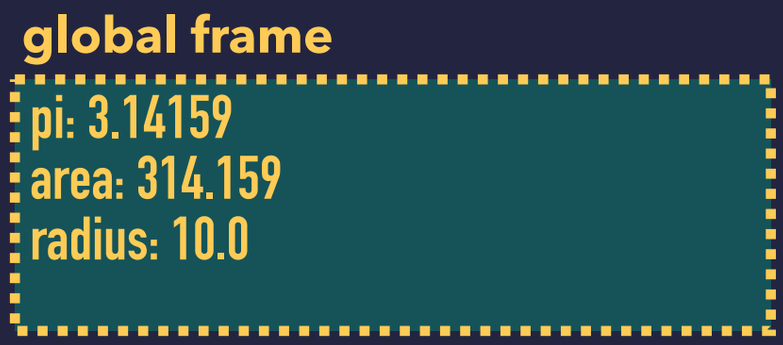
```
pi = 3.14159
area = float(input("Circle area? "))
radius = (area / pi) ** 0.5
print("That circle's radius is "+str(radius)+".")
```



- 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 value is changed with each assignment statement.

RECALL: PYTHON EXECUTION

- Let's take a look at this script:



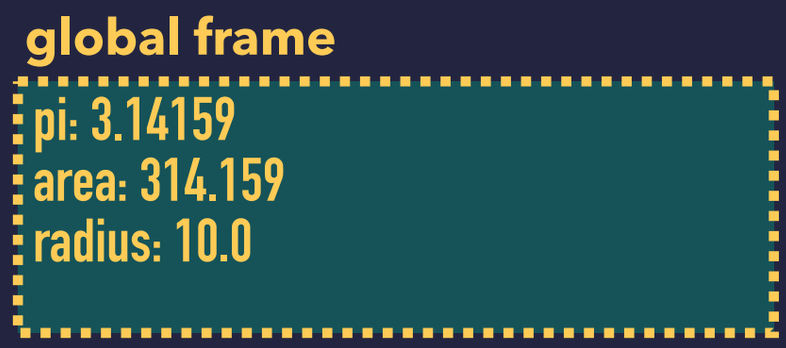

```
pi = 3.14159
area = float(input("Circle area? "))
radius = (area / pi) ** 0.5
print("That circle's radius is "+str(radius)+".")
```

- 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 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 value is changed with each assignment statement.
    - ➡ The collection of variable slots of a script is called the ***global frame***

CONDITIONAL EXECUTION

LECTURE 02-1

THE CONDITIONAL STATEMENT
THE BOOLEAN TYPE

JIM FIX, REED COLLEGE CSCI 121

"FLOW OF CONTROL"

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

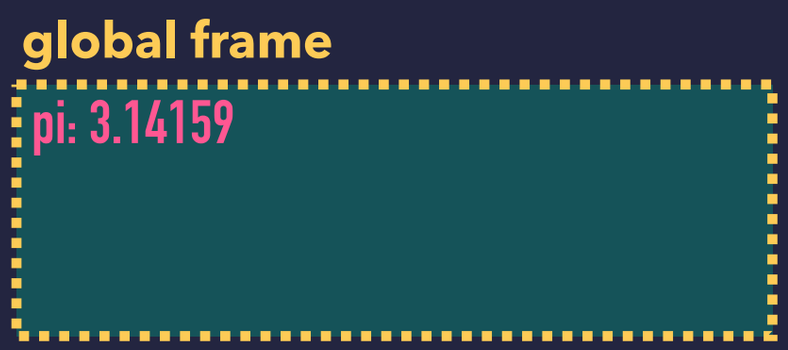
The interpreter goes through the code line-by-line, tracking where it's at with an instruction pointer.

- ➡ 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.

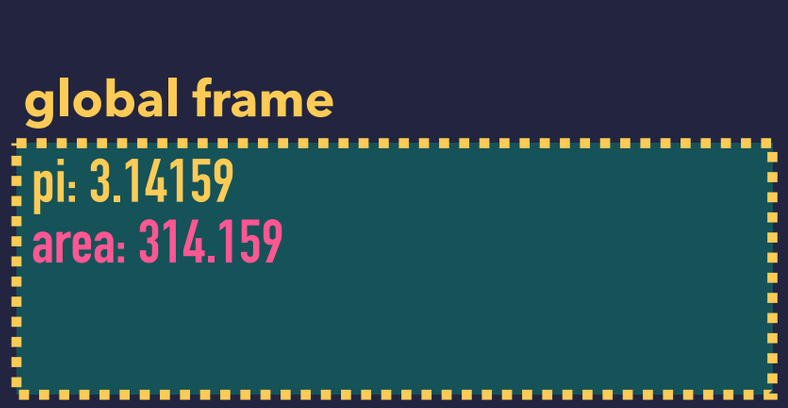
RECALL: STRAIGHT LINE PYTHON EXECUTION

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



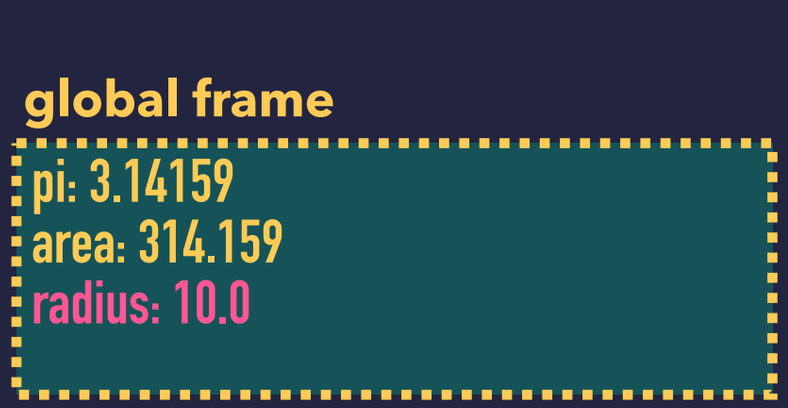
RECALL: STRAIGHT LINE PYTHON EXECUTION

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



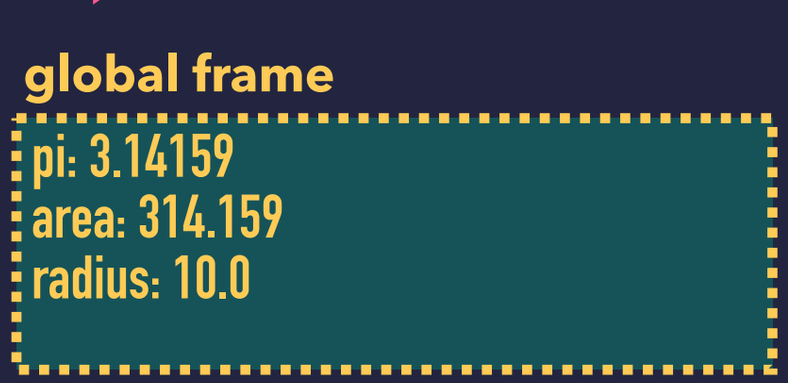
RECALL: STRAIGHT LINE PYTHON EXECUTION

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



RECALL: STRAIGHT LINE PYTHON EXECUTION

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



"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.

"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.

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))
```

```
% 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
```

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 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.

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 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`**.

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
```

SHORT-CIRCUITED LOGIC CONNECTIVES

- Evaluation of **`and`** and **`or`** is ***short-circuited***:

```
>>> x = 0
>>> 45 / x
ERROR!!!
>>> (x == 0) or ((45 / x) > 10)
True
>>> (x != 0) and ((45 / x) > 10)
False
```

- Python doesn't bother with the right of **`or`** if the left is **`True`**.
- Python doesn't bother with the right of **`and`** if the left is **`False`**.
- This means, for example, that **`and`** is executed like this:

```
if x != 0:
    return (45 / x) > 10
else:
    return 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 condition1:
    execute if condition1 holds
    ...
elif condition2:
    execute if condition1 does not hold but condition2 does
    ...
...
else:
    executed if no condition 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
```

CHECKING BOOLEAN VALUES

- Many beginning programmers are tempted to write this code:

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

CHECKING BOOLEAN VALUES IS REDUNDANT

- Many beginning programmers are tempted to write this code:

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

CHECKING BOOLEAN VALUES IS REDUNDANT

- Write this code instead:

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

- By using `if`, you are *already checking* whether the condition `== True`.

CHECKING BOOLEAN VALUES IS REDUNDANT

- Write this code instead:

```
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.")
```

- By using `if`, you are *already checking* whether the condition `== True`.

CONTROL FLOW PREVIEW: 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.

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,

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 == !=`