CONDITIONAL EXECUTION

LECTURE 02-1

JIM FIX, REED COLLEGE CSCI 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~~
  - 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

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.

~~**For now and in the near future, it is at**~~

~~http://xifmij.github.io/csci121~~

The far future is here!!! Go to http://jimfix.github.io/csci121

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.

For now and in the near future, it is at

http://xifmij.github.io/csci121

HOMEWORK? LAB? HOW ARE THINGS?

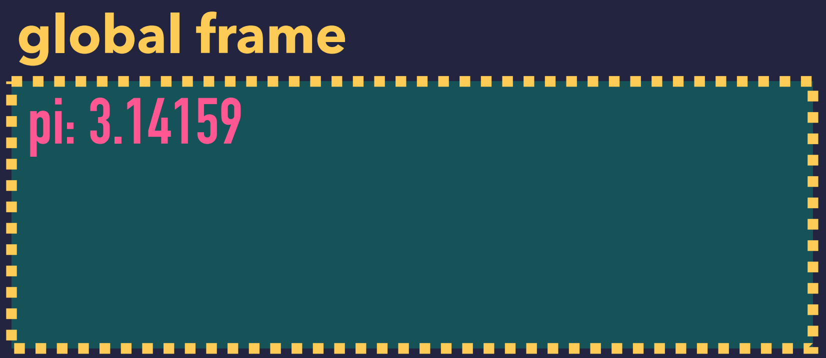
- Don't forget to complete the **Homework 2** assignment:
  - due next Tuesday 9/16, before 9am
- Any questions from lab? about Homework 2? about Homework 1?

READING

- This week's lecture material can be supplemented with:
  - **Reading**:
    - TP Chs 4.1-4.8 (conditionals)
    - CP 1.5 ("control")

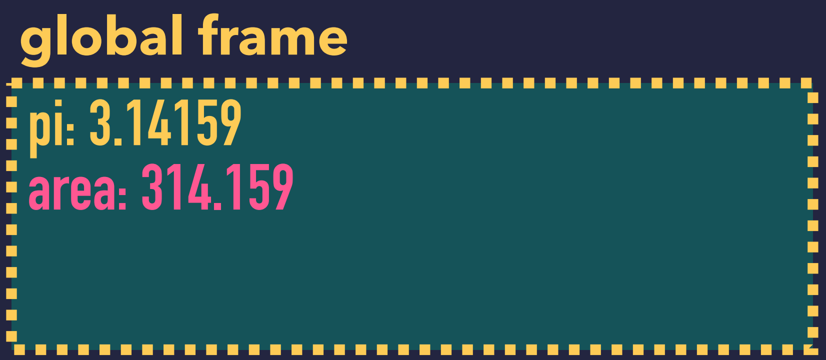
RECALL: STRAIGHT LINE PYTHON EXECUTION

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



global frame

pi: 3.14159
area: 314.159

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)+".")
```

global frame

pi: 3.14159
area: 314.159
radius: 10.0

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)+".")
```

global frame

pi: 3.14159
area: 314.159
radius: 10.0

"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` ~> `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 the `award_prize` code in the autograder:

```
if on_time:

    if all_correct:
        mesg =  "Great work passing all the tests!\n"
        mesg += "You've earned the prize points."
    else:
        mesg = "To earn prize 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 prize points.\n"
        mesg += "You submitted this after the deadline."
    else:
        mesg =  "Sorry! No prize 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

- The code below is like some code in the 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

- The code below is also like the `award_prize` code in the autograder:

```
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 prize points.\n"
    mesg += "Excellent!"
elif attempts <= 6:
    mesg += "You earned 80% of the prize points.\n"
    mesg += "Nicely done."
else:
    mesg += "This is a few more times than we'd prefer.\n"
    mesg += "We awarded half of the prize 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.

CONTROL FLOW PREVIEW: CALL AND RETURN

- Python lets us define our own functions.
- Below is an example with two: `getArea` and `radiusOfCircle`.

```
def getArea():
    a = float(input("Circle area? “))
    while a < 0.0:
        a = float(input(“Not an area. Try again:”))
    return a

def radiusOfCircle(someArea):
    from math import pi, sqrt
    return sqrt(someArea / pi)

area = getArea()
radius = radiusOfCircle(area)
print("That circle’s radius is “+str(radius)+”.”)
```

CONTROL FLOW PREVIEW: CALL AND RETURN

- Python lets us define our own functions.
- Below is an example with two: **`getArea`** and **`radiusOfCircle`**.

```
def getArea():
    a = float(input("Circle area? “))
    while a < 0.0:
        a = float(input(“Not an area. Try again:”))
    return a
```

```
def radiusOfCircle(someArea):
    from math import pi, sqrt
    return sqrt(someArea / pi)
```

```
area = getArea()
radius = radiusOfCircle(area)
print("That circle’s radius is “+str(radius)+”.”)
```

CONTROL FLOW PREVIEW: CALL AND RETURN

- The instruction pointer jumps from the main script code, up to the function's code, and then returns back.

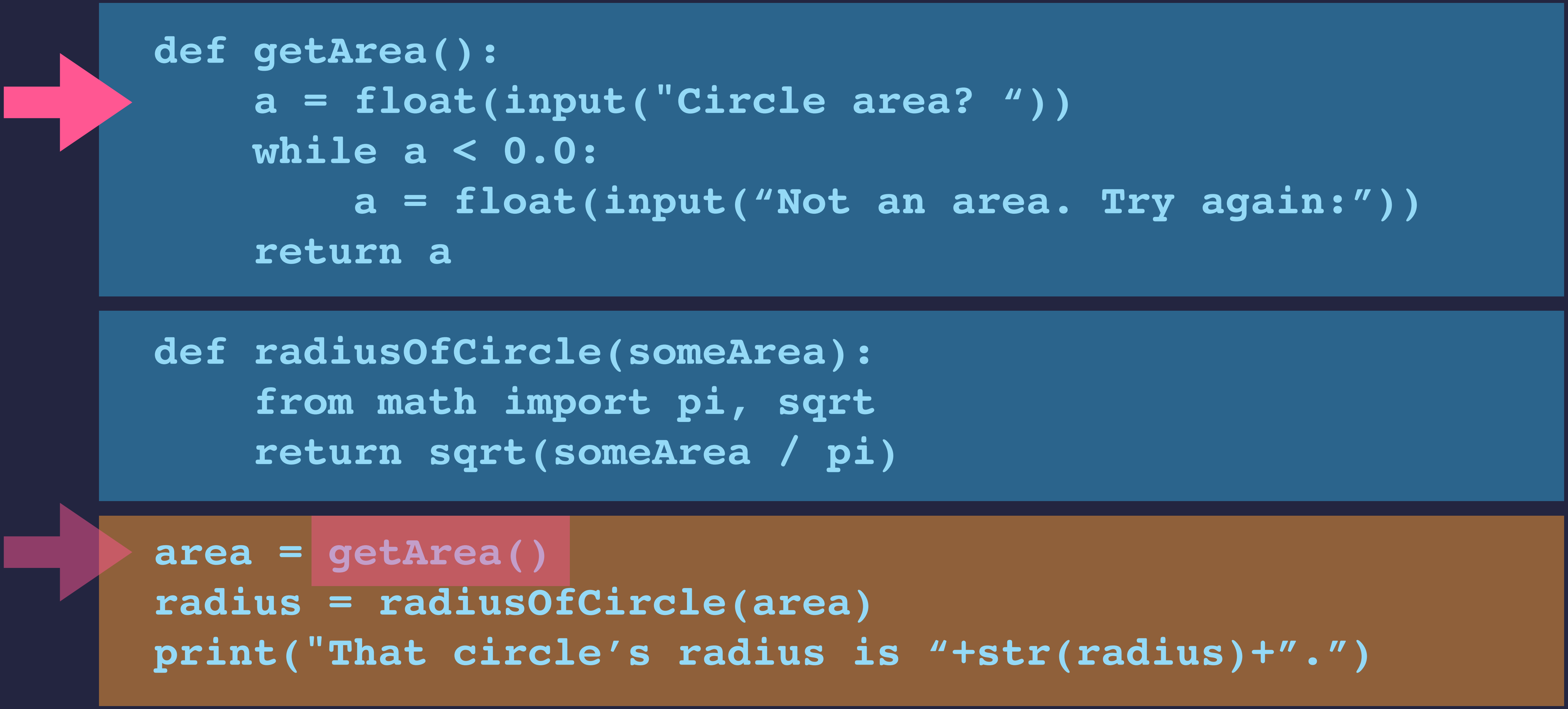
```
def getArea():
    a = float(input("Circle area? “))
    while a < 0.0:
        a = float(input(“Not an area. Try again:”))
    return a
```

```
def radiusOfCircle(someArea):
    from math import pi, sqrt
    return sqrt(someArea / pi)
```

```
area = getArea()
radius = radiusOfCircle(area)
print("That circle’s radius is “+str(radius)+”.”)
```

CONTROL FLOW PREVIEW: CALL AND RETURN

- The instruction pointer jumps from the main script code, up to the function's code, and then returns back.



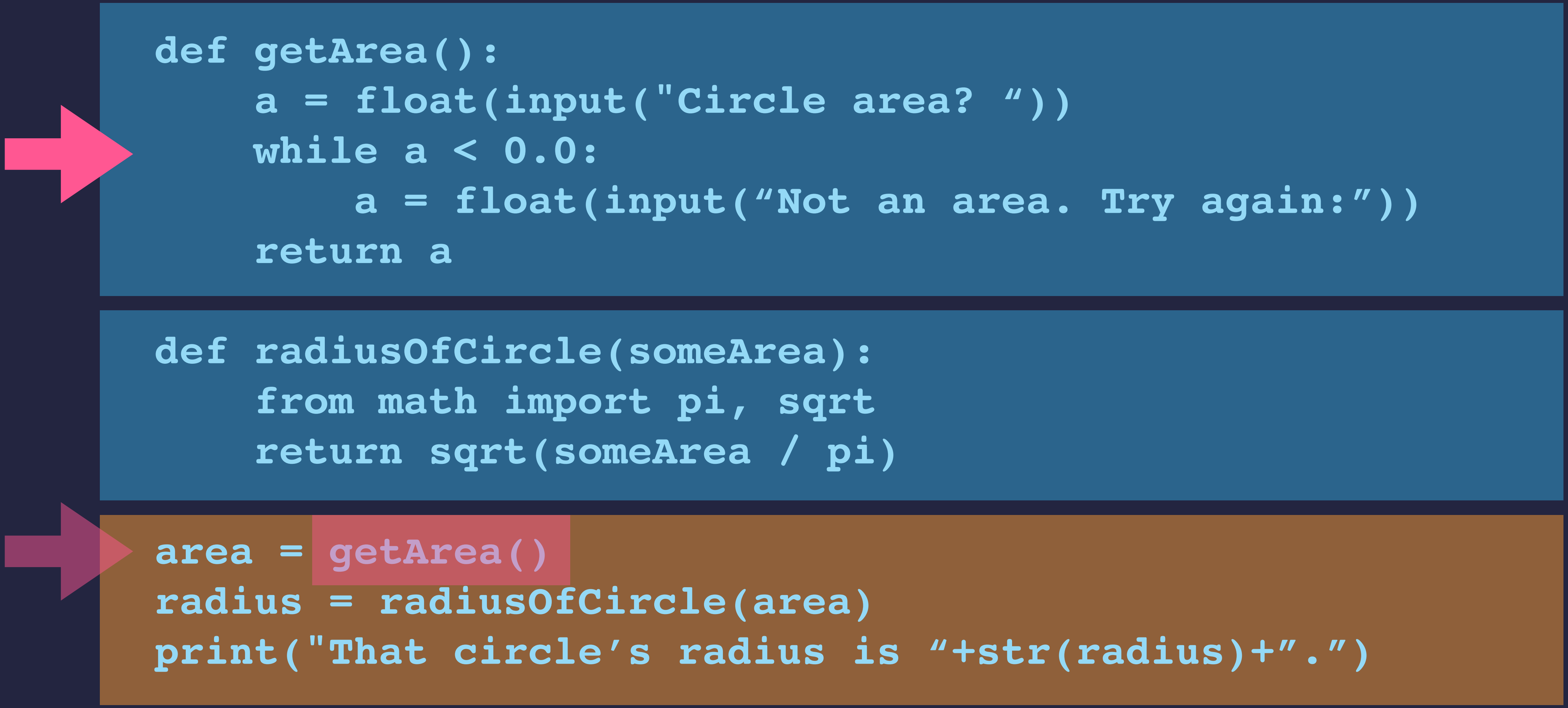
```
def getArea():
    a = float(input("Circle area? “))
    while a < 0.0:
        a = float(input(“Not an area. Try again:”))
    return a
```

```
def radiusOfCircle(someArea):
    from math import pi, sqrt
    return sqrt(someArea / pi)
```

```
area = getArea()
radius = radiusOfCircle(area)
print("That circle’s radius is “+str(radius)+”.”)
```

CONTROL FLOW PREVIEW: CALL AND RETURN

- The instruction pointer jumps from the main script code, up to the function's code, and then returns back.



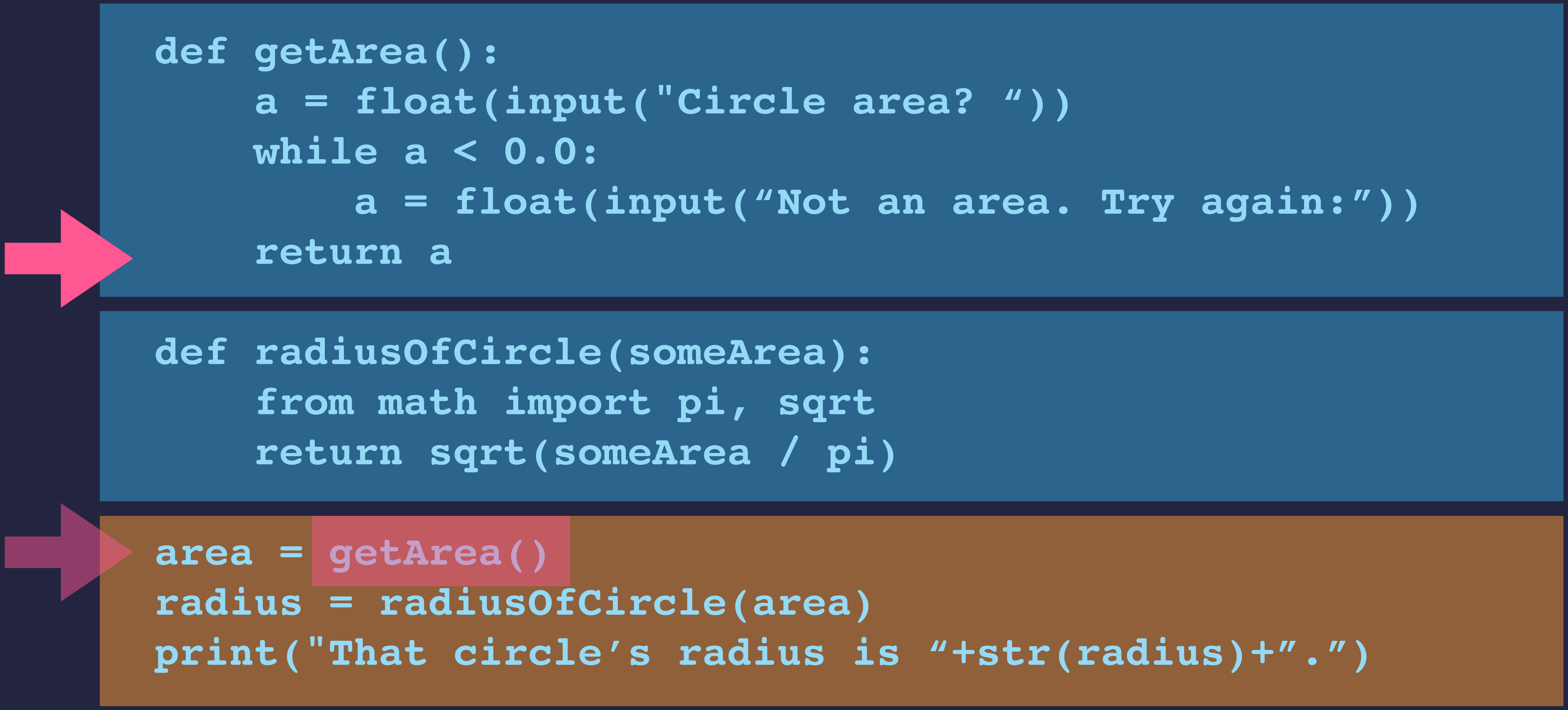
```
def getArea():
    a = float(input("Circle area? “))
    while a < 0.0:
        a = float(input(“Not an area. Try again:”))
    return a
```

```
def radiusOfCircle(someArea):
    from math import pi, sqrt
    return sqrt(someArea / pi)
```

```
area = getArea()
radius = radiusOfCircle(area)
print("That circle’s radius is “+str(radius)+”.”)
```

CONTROL FLOW PREVIEW: CALL AND RETURN

- The instruction pointer jumps from the main script code, up to the function's code, and then returns back.



```
def getArea():
    a = float(input("Circle area? “))
    while a < 0.0:
        a = float(input(“Not an area. Try again:”))
    return a
```

```
def radiusOfCircle(someArea):
    from math import pi, sqrt
    return sqrt(someArea / pi)
```

```
area = getArea()
radius = radiusOfCircle(area)
print("That circle’s radius is “+str(radius)+”.”)
```

CONTROL FLOW PREVIEW: CALL AND RETURN

- The instruction pointer jumps from the main script code, up to the function's code, and then returns back.

```
def getArea():
    a = float(input("Circle area? “))
    while a < 0.0:
        a = float(input(“Not an area. Try again:”))
    return a
```

```
def radiusOfCircle(someArea):
    from math import pi, sqrt
    return sqrt(someArea / pi)
```

```
area = getArea()
radius = radiusOfCircle(area)
print("That circle’s radius is “+str(radius)+”.”)
```

CONTROL FLOW PREVIEW: CALL AND RETURN

- The instruction pointer jumps from the main script code, up to the function's code, and then returns back.

```
def getArea():
    a = float(input("Circle area? “))
    while a < 0.0:
        a = float(input(“Not an area. Try again:”))
    return a
```

```
def radiusOfCircle(someArea):
    from math import pi, sqrt
    return sqrt(someArea / pi)
```

```
area = getArea()
radius = radiusOfCircle(area)
print("That circle’s radius is “+str(radius)+”.”)
```

CONTROL FLOW PREVIEW: CALL AND RETURN

- The instruction pointer jumps from the main script code, up to the function's code, and then returns back.

```
def getArea():
    a = float(input("Circle area? “))
    while a < 0.0:
        a = float(input(“Not an area. Try again:”))
    return a
```

```
def radiusOfCircle(someArea):
    from math import pi, sqrt
    return sqrt(someArea / pi)
```

```
area = getArea()
radius = radiusOfCircle(area)
print("That circle’s radius is “+str(radius)+”.”)
```

CONTROL FLOW PREVIEW: CALL AND RETURN

- The instruction pointer jumps from the main script code, up to the function's code, and then returns back.

```
def getArea():
    a = float(input("Circle area? “))
    while a < 0.0:
        a = float(input(“Not an area. Try again:”))
    return a
```

```
def radiusOfCircle(someArea):
    from math import pi, sqrt
    return sqrt(someArea / pi)
```

```
area = getArea()
radius = radiusOfCircle(area)
print("That circle’s radius is “+str(radius)+”.”)
```

READING

- This and next week's lecture material can be supplemented with:
  - **Reading**:
    - TP Chs 4.1-4.8 (conditionals)
    - Ch. 3, 6 (functions)
    - CP 1.3-1.4 (user-defined functions); 1.5 ("control")