

# CONDITIONAL EXECUTION

---

## LECTURE 02-1

JIM FIX, REED COLLEGE CSCI 121

## READING

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

# 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:
        msg = "Great work passing all the tests!\n"
        msg += "You've earned the prize points."
    else:
        msg = "To earn prize points, make sure all the tests pass."

else:

    if all_correct:
        msg = "Great work making all the tests pass.\n"
        msg += "Sadly we can't offer you any prize points.\n"
        msg += "You submitted this after the deadline."
    else:
        msg = "Sorry! No prize points."

print(msg)
```

# 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
msg = "Great work passing all the tests!\n"
msg += "You submitted " + str(attempts) + " times.\n"

if attempts <= 2:
    msg += "You earned the full prize points.\n"
    msg += "Excellent!"
elif attempts <= 6:
    msg += "You earned 80% of the prize points.\n"
    msg += "Nicely done."
else:
    msg += "This is a few more times than we'd prefer.\n"
    msg += "We awarded half of the prize points."

print(msg)
```

# 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)
```

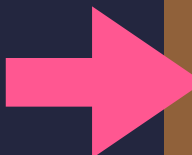
```
area = getArea()  
radius = radiusOfCircle(area)  
print("That circle's radius is "+str(radius)+".")
```

## 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
```

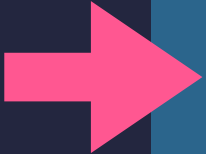
```
def radiusOfCircle(someArea):  
    from math import pi, sqrt  
    return sqrt(someArea / pi)
```



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

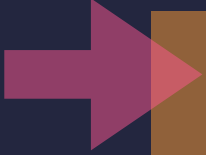
## CONTROL FLOW PREVIEW: CALL AND RETURN

- ▶ 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
```

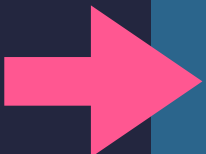
```
def radiusOfCircle(someArea):  
    from math import pi, sqrt  
    return sqrt(someArea / pi)
```



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

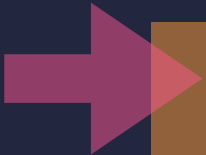
## CONTROL FLOW PREVIEW: CALL AND RETURN

- ▶ 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
```

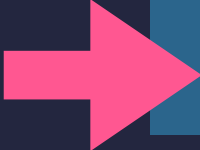
```
def radiusOfCircle(someArea):  
    from math import pi, sqrt  
    return sqrt(someArea / pi)
```



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

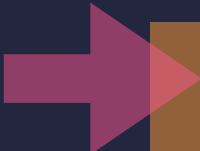
## CONTROL FLOW PREVIEW: CALL AND RETURN

- ▶ 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)
```



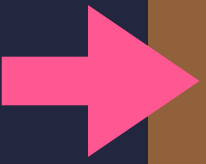
```
area = getArea()  
radius = radiusOfCircle(area)  
print("That circle's radius is "+str(radius)+".")
```

## 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)
```

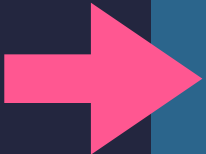


```
area = getArea()  
radius = radiusOfCircle(area)  
print("That circle's radius is "+str(radius)+".")
```

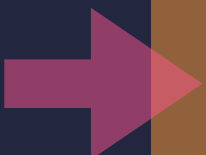
## CONTROL FLOW PREVIEW: CALL AND RETURN

- ▶ 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)
```

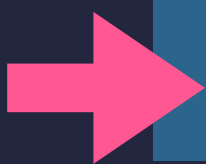


```
area = getArea()  
radius = radiusOfCircle(area)  
print("That circle's radius is "+str(radius)+".")
```

## 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)+".")
```



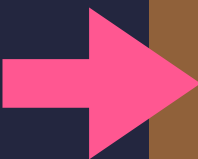
## CONTROL FLOW PREVIEW: CALL AND RETURN

- ▶ 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")