Booleans and Conditionals

15-110 – Monday 09/14

Learning Goals

- Use **logical operators** on Booleans to compute whether an expression is True or False
- Use **conditionals** when reading and writing algorithms that make choices based on data
- Use **nesting** of control structures to create complex control flow
- Debug logical errors by using the **scientific method**

Logical Operators

Booleans are values that can be True or False

In week 1, we learned about the **Boolean** type, which can be one of two values: `True` or `False`.

Until now, we've made Boolean values by comparing different values, such as:

```
x < 5
s == "Hello"
7 >= 2
```

Logical Operations Combine Booleans

We aren't limited to only evaluating a single Boolean comparison! We can **combine** Boolean values using **logical operations.** We'll learn about three – **and, or,** and **not**.

Combining Boolean values will let us check complex requirements while running code.

and Operation Checks Both

The `and` operation takes two Boolean values and evaluates to `True` if **both** values are `True`. In other words, it evaluates to `False` if **either** value is `False`.

We use `and` when we want to require that both conditions be met at the same time.

Example:

```
(x >= 0) and (x < 10)
```

a	b	a and b
True	True	**True**
True	False	False
False	True	False
False	False	False

`or` Operation Checks Either

The `or` operation takes two Boolean values and evaluates to `True` if **either** value is `True`. In other words, it only evaluates to `False` if **both** values are `False`.

We use `or` when there are multiple valid conditions to choose from.

a	b	a or b
True	True	True
True	False	True
False	True	True
False	False	**False**

Example:

```
(day == "Saturday") or (day == "Sunday")
```

`not` Operation Reverses Result

Finally, the `not` operation takes a single Boolean value and switches it to the opposite value (negates it). `not True` becomes `False`, and `not False` becomes `True`.

We use `not` to switch the result of a Boolean expression. For example, `not (x < 5)` is the same as `x >= 5`.

Example:

```
not (x == 0)
```

a	not a
True	False
False	True

Activity: Guess the Result

If `x = 10`, what will each of the following expressions evaluate to?

```
x < 25 and x > 15
```

```
x < 25 or x > 15
```

```
not (x > 5 and x < 10)
```

```
(x > 5) or ((x**2 > 50) and (x == 20))
```

```
((x > 5) or (x**2 > 50)) and (x == 20)
```

Sidebar: DeMorgan's Laws

not (A and B) ⇔ (not A) or (not B)

not (A or B) ⇔ (not A) and (not B)

Augustus De Morgan
(1806–1871)

Conditionals

Conditionals Make Decisions

With Booleans, we can make a new type of code called a **conditional**. Conditionals are a form of a **control structure** – they let us change the direction of the code based on the value that we provide.

To write a conditional (`if` **statement**), we use the following structure:

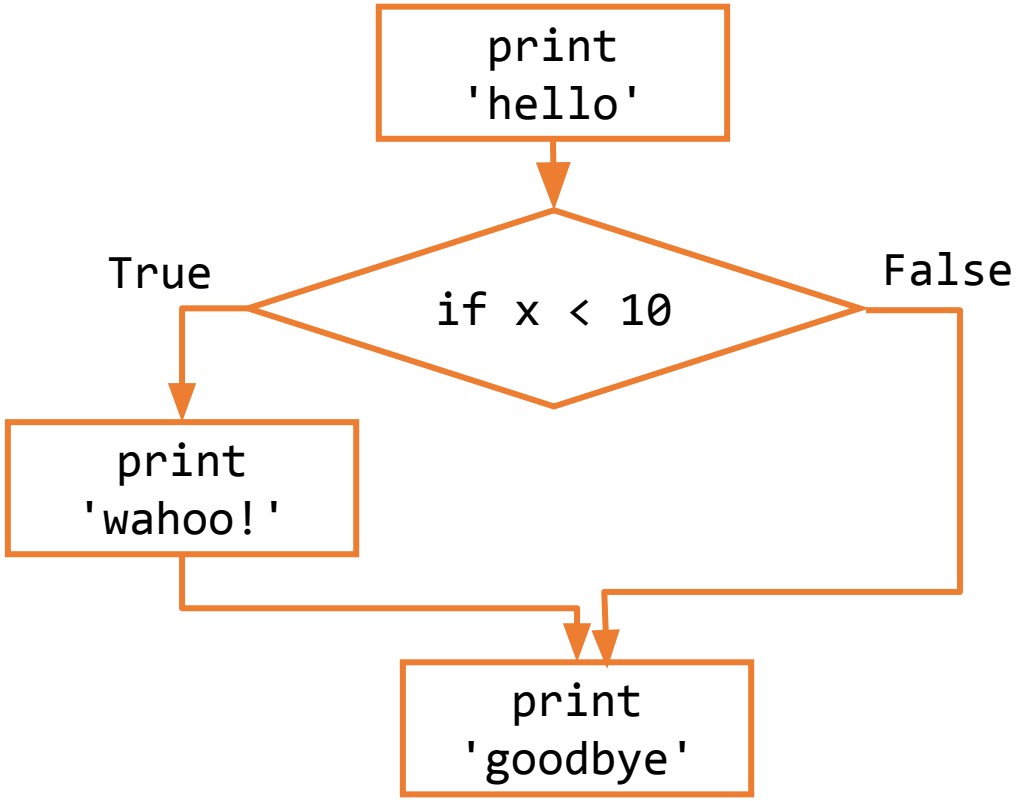
```
if <BooleanExpression>:
    <bodyIfTrue>
```

Note that, like a function definition, the top line of the `if` statement ends with a colon, and the **body** of the if statement is indented.

Flow Charts Show Code Choices

We'll use a **flow chart** to demonstrate how Python executes an `if` statement based on the values provided.

```
print("hello")
if x < 10:
    print("wahoo!")
print("goodbye")
```



`wahoo!` is only printed if `x` is less than `10`. But `hello` and `goodbye` are always printed.

The Body of an If Can Have Many Statements

The body of an `if` statement can have any number of statements in it. As with function definitions, each statement of the body is on a separate line and indented. The body ends when the next line of code is unindented.

```
print("hello")
if x < 10:
    print("wahoo!")
    print("wahoo!")
print("goodbye")
```

if x < 10, prints:

```
hello
wahoo!
wahoo!
goodbye
```

if x >= 10, prints:

```
hello
goodbye
```

Else Clauses Allow Alternatives

Sometimes we want a program to do one of two alternative actions based on the condition. In this case, instead of writing two `if` statements, we can write a single `if` statement and add an **`else`**.

The `else` is executed when the Boolean expression is `False`.

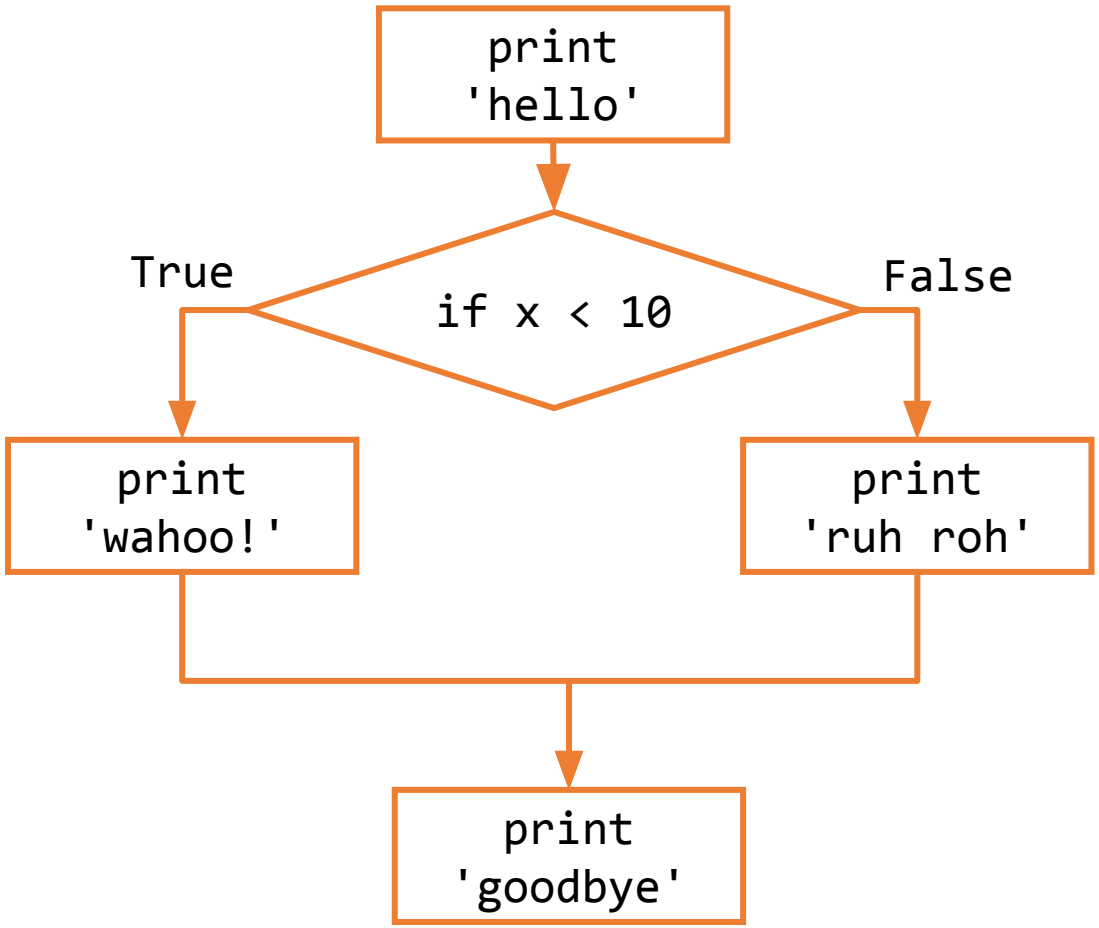
```
if <BooleanExpression>:
    <bodyIfTrue>
else:
    <bodyIfFalse>
```

`if` clause

`else` clause

Updated Flow Chart Example

```
print("hello")
if x < 10:
    print("wahoo!")
else:
    print("ruh roh")
print("goodbye")
```



Activity: Conditional Prediction

Prediction Exercise: What will the following code print?

```
x = 5
if x > 10:
    print("Up high!")
else:
    print("Down low!")
```

Question: What could we change to print the other string instead?

Question: Can we get the if/else statement to print out both statements?

Else Must Be Paired With If

It's impossible to have an `else` clause by itself, as it would have no condition to be the alternative to.

Therefore, **every `else` must be paired with an `if`**. On the other hand, every `if` can have **at most one `else`**.

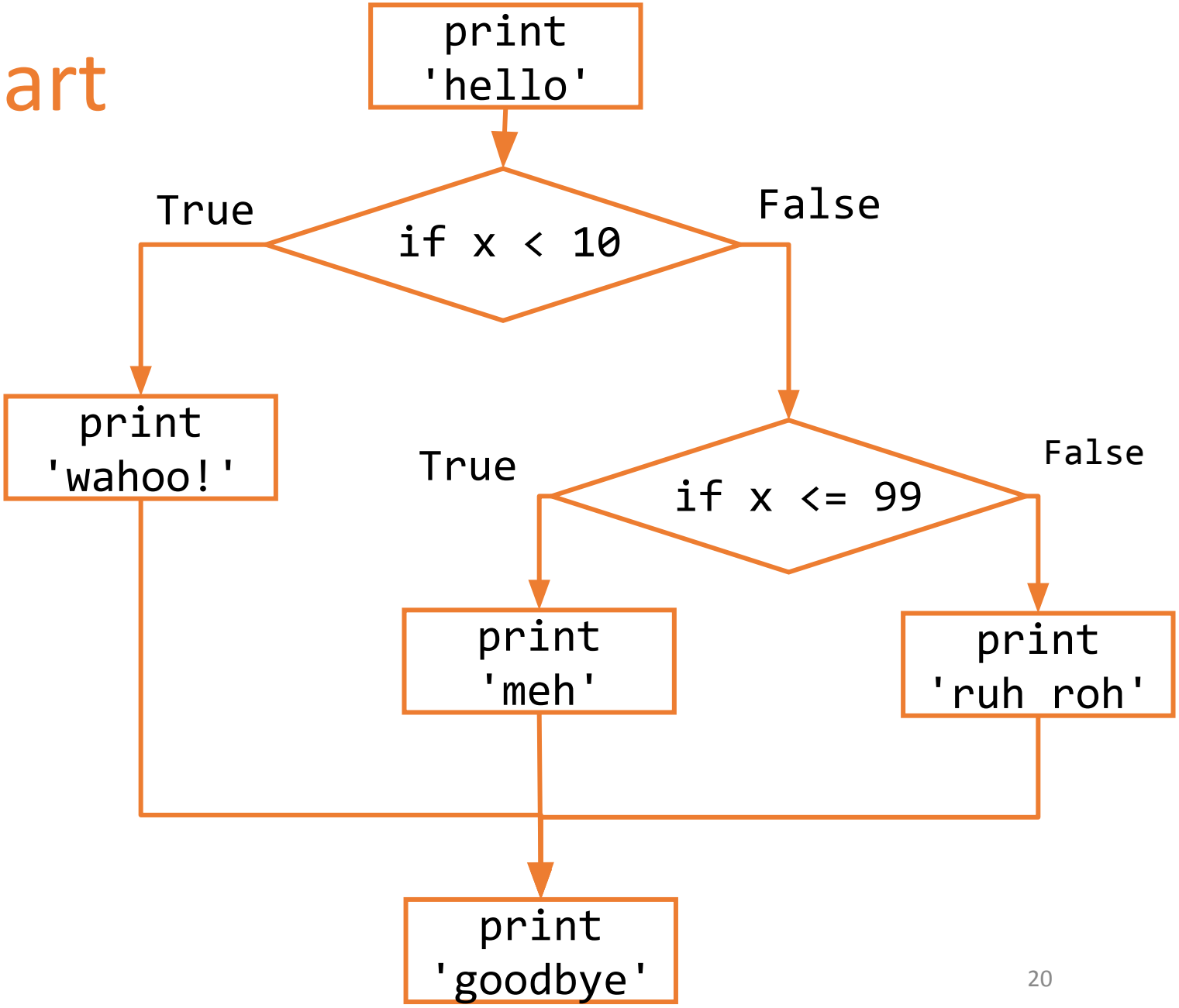
`Elif` Implements Multiple Alternatives

Finally, we can use **`elif`** statements to add alternatives with their own conditions to `if` statements. An `elif` is like an `if`, except that it is checked **only if all previous conditions evaluate to `False`**.

```
if <BooleanExpressionA>:
    <bodyIfATrue>
elif <BooleanExpressionB>:
    <bodyIfAFalseAndBTrue>
else:
    <bodyIfBothFalse>
```

Updated Flow Chart

```
print("hello")
if x < 10:
    print("wahoo!")
elif x <= 99:
    print("meh")
else:
    print("ruh roh")
print("goodbye")
```



Conditional Statements Join Clauses Together

We can have more than one `elif` clause associated with an `if` statement. In fact, we can have as many as we need! But, as with `else`, an `elif` must be associated with an `if` (or a previous `elif`).

In general, a **conditional statement** is an `if` clause, with zero or more `elif` clauses, then an optional `else` clause that are all joined together. These joined clauses can be considered a single **control structure**. Only one clause will have its body executed.

Example: gradeCalculator

Let's write a few lines of code that takes a grade as a number, then prints the letter grade that corresponds to that number grade.

90+ is an A, 80-90 is a B, 70-80 is a C, 60-70 is a D, and below 60 is an R.

Short-Circuit Evaluation

When Python evaluates a logical expression, it acts lazily. It only evaluates the second part **if it needs to**. This is called **short-circuit evaluation**.

When checking `x and y`, if `x` is `False`, **the expression can never be `True`**. Therefore, Python doesn't even evaluate `y`.

When checking `x or y`, if `x` is `True`, **the expression can never be `False`**. Python doesn't evaluate `y`.

This is a handy method for keeping errors from happening. For example:

```
if x != 0 and y % x == 0:
    print("Factor:", x)
```

Nesting Control Structures

Nesting Creates More Complex Control Flow

Now that we have a control structure, **we can put `if` statements inside of `if` statements**.

In general, we'll be able to **nest** control structures inside of other control structures. We can also nest control structures inside of function definitions.

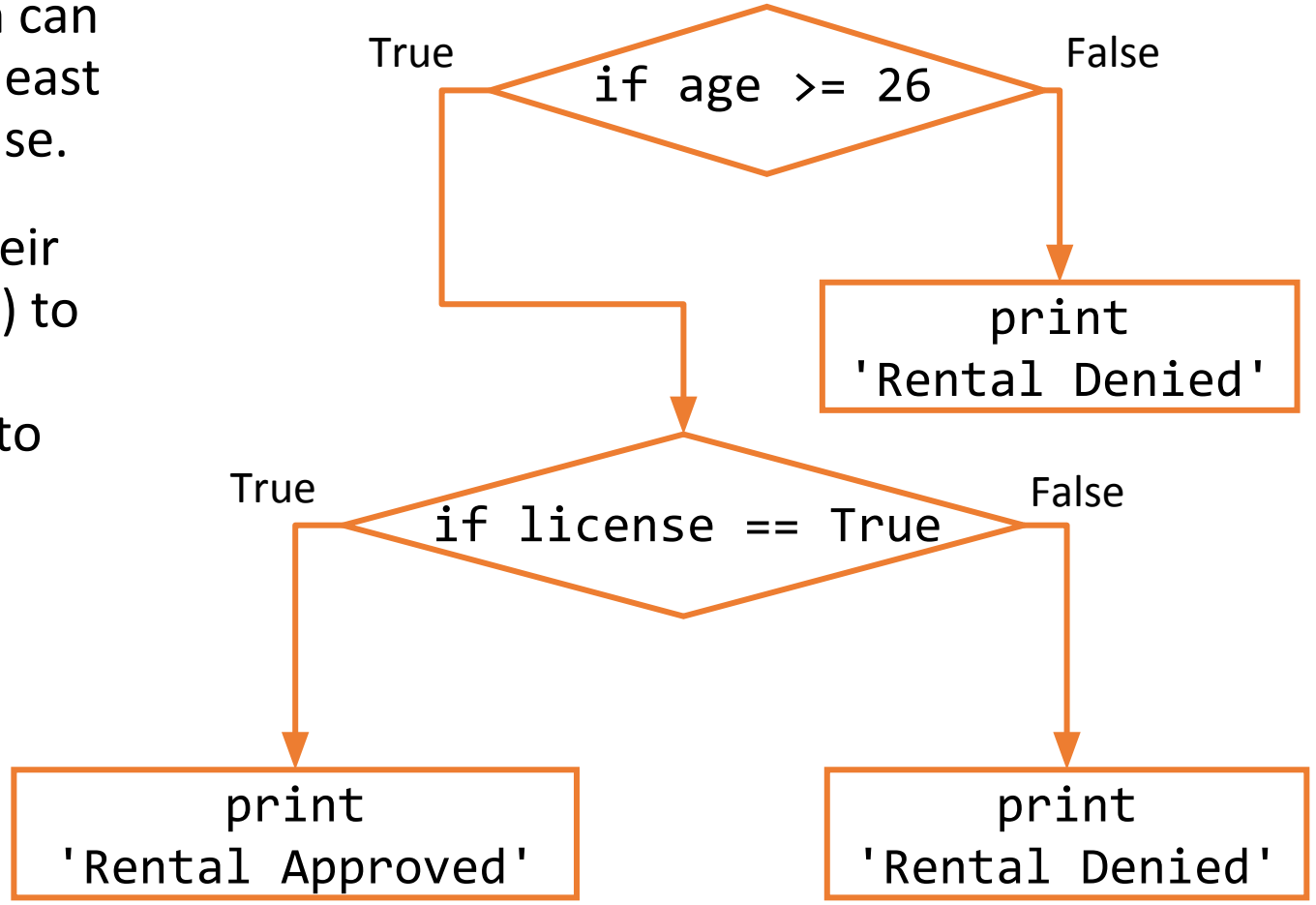
In program syntax, we demonstrate that a control structure is nested by **indenting the code** so that it's in the outer control structure's body.

Example: Car rental program

Consider code that determines if a person can rent a car based on their age (are they at least 26) and whether they have a driver's license.

We can use one `if` statement to check their age, then a second (nested inside the first) to check the license. We'll only print 'Rental Approved' if both `if` conditions evaluate to `True`.

```
if age >= 26:
    if license == True:
        print("Rental Approved")
    else:
        print("Rental Denied")
else:
    print("Rental Denied")
```

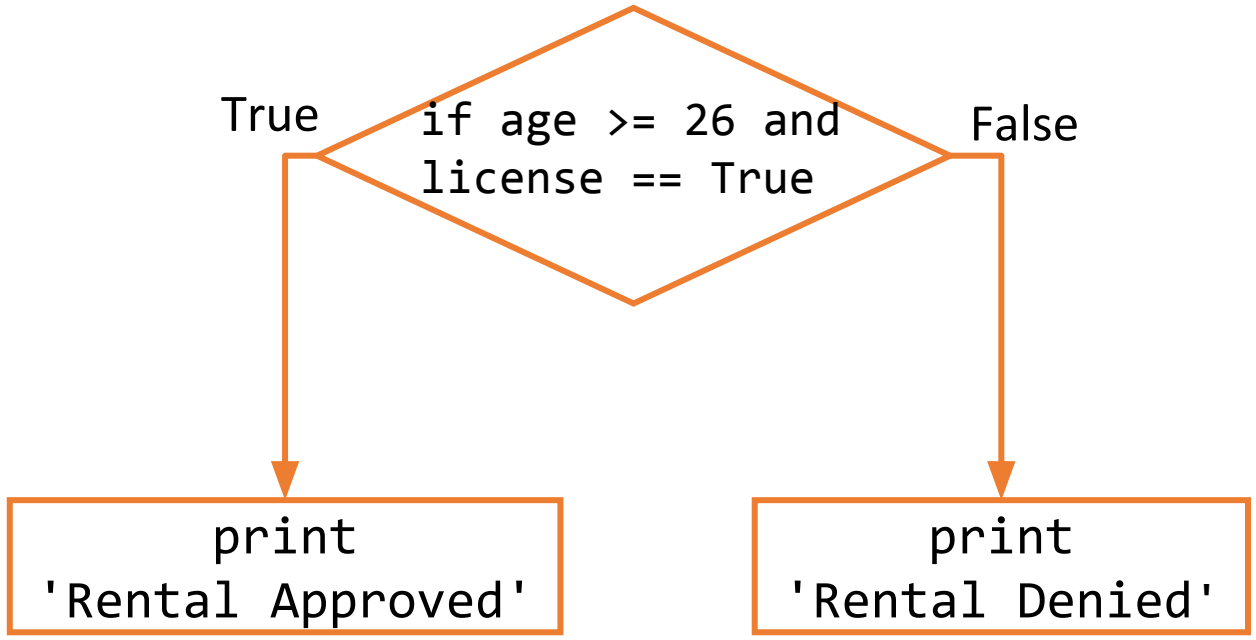


Alternative Car Rental Code

In the code below, we accomplish the same result with the and operation.

This won't always work, though – it depends on how many different results you want.

```
if age >= 26 and license == True:
    print("Rental Approved")
else:
    print("Rental Denied")
```



Nesting and If/Elif/Else Statements

When we have nested conditionals with `elif` or `else` clauses, Python pairs them with the `if` clause at the **same indentation level**. This is true even if an inner `if` statement comes in between the outer clauses! However, an outer `if`/`elif`/`else` statement **cannot** come between parts of an inner conditional.

```
if first == True:
    if second == True:
        print("both true!")
else:
    print("first not true")
```

Question: if we want to add an `else` statement to the inner `if`, where should it go?

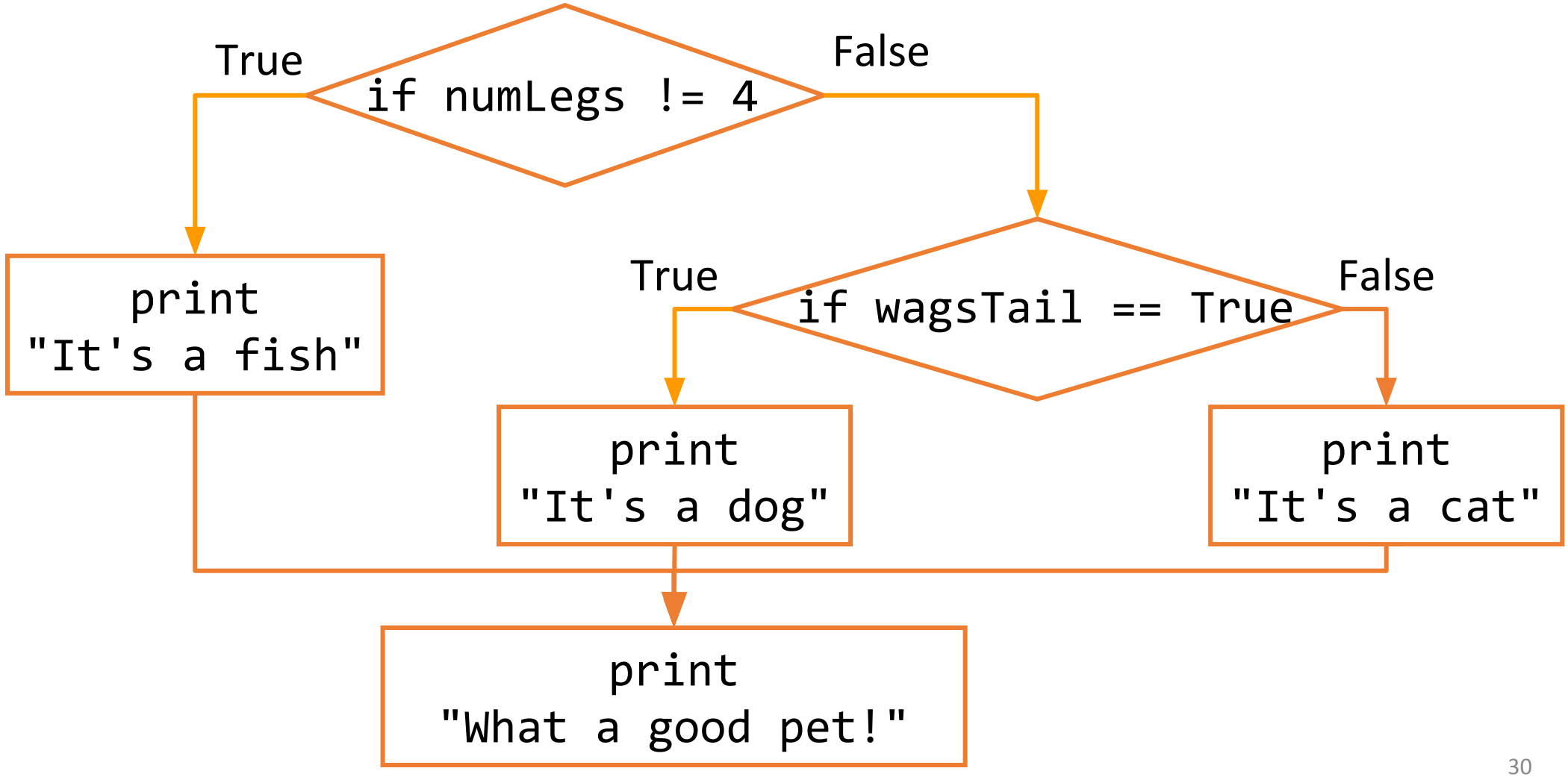
Nesting Conditionals in Functions

When we nest a conditional inside a function definition, we can **return values early,** instead of only returning on the last line. Returning early is fine as long as we ensure every possible path the function can take will eventually return a value.

A function will always end as soon as it reaches a `return` statement, even if more lines of code follow it. For example, the following function will not crash when `n` is zero.

```
def findAverage(total, n):
    if n <= 0:
        return "Cannot compute the average"
    return total / n
```

Exercise: Convert Flow Chart to Code



Testing and Debugging with Control Structures

Test Cases Use `assert` Statements

Starting with Check2, the **starter file** for programming assignments will contain test cases. A **test case** is a line of code that checks whether a function produces the expected output on a given input:

```
assert(findAverage(20, 4) == 5)
```

An `assert` statement takes a Boolean expression. If the expression evaluates to `True`, the statement does nothing. If it evaluates to `False`, the program crashes.

To check your solutions against the test cases, use **Run File as Script**. If you have not commented out the test cases and the file runs without crashing, your code is (probably) correct. On the other hand, if your code throws an `AssertionError`, that means you have a **logical error** in one or more of your solutions.

Debug Logical Errors By Checking Inputs and Outputs

When your code generates a logical error, the best thing to do is **compare the expected output to the actual output**.

1. Copy the function call from the `assert` that is failing into the interpreter. Compare the actual output to the expected output.
2. If the **expected** output seems incorrect, re-read the problem prompt.
3. If you're not sure why the actual output is produced, use the **debugging process** to investigate.

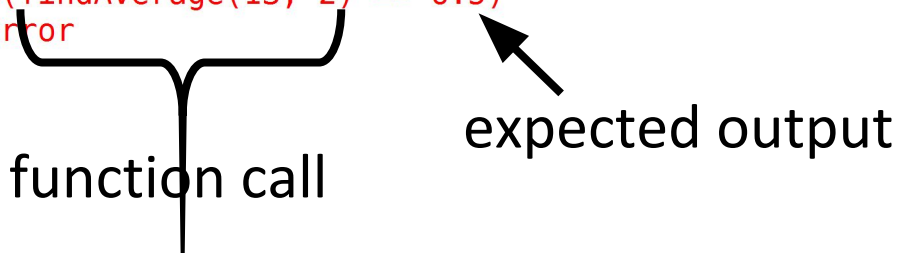
If you've written the test set yourself, you should also take a moment to make sure the test itself is not incorrect.

example.py

```
1  def findAverage(total, n):
2      if n <= 0:
3          return "Cannot compute the average"
4      return total // n
5
6  def testFindAverage():
7      print("Testing findAverage()...", end="")
8      assert(findAverage(20, 4) == 5)
9      assert(findAverage(13, 2) == 6.5)
10     assert(findAverage(10, 0) == "Cannot compute the average")
11     print("... done!")
12
13 testFindAverage()
```

```
Running script: "C:\Users\river\Downloads\example.py"
Testing findAverage()...Traceback (most recent call last):
  File "C:\Users\river\Downloads\example.py", line 13, in
<module>
    testFindAverage()
  File "C:\Users\river\Downloads\example.py", line 9, in t
estFindAverage
    assert(findAverage(13, 2) == 6.5)
AssertionError

>>>
```



Sidebar: Clean Up Top-Level Testing

Some students like to test their code by adding print statements and function calls at the top level of the code (not inside a function).

This is fine, but if you do this, **remove the top-level code** before you submit on Gradescope. Otherwise, the tool might mark your entire submission as incorrect, instead of only marking the single broken function.

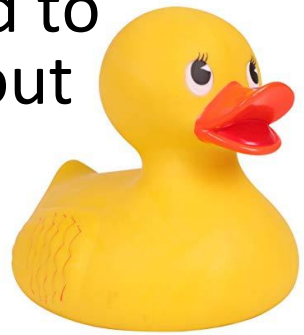
Alternative approach: do testing in the interpreter! After you 'Run File as Script', all of your functions are available there to be tested.

Understanding Your Code

When something goes wrong with your code, before rushing to change the code itself, you should make sure you understand **conceptually** what your code does.

First- make sure you're solving the right problem! Re-read the problem prompt to check that you're doing the right task.

If you find yourself getting stuck, try **rubber duck debugging**. Explain what your code is supposed to do and what is going wrong out loud to an inanimate object, like a rubber duck. Sometimes, saying things out loud will help you realize what's wrong.



Debug with the Scientific Method

When you're trying to debug a tricky error, you should use a process similar to the **scientific method**. We'll reduce it down to five core steps:

1. Collect data
2. Make a hypothesis
3. Run an experiment
4. Observe the result
5. Repeat the process (if necessary)

Step 1: Collect Data

First, you need to **collect data** about what your code is currently doing.

You can already see the steps of your algorithm, but you can't see how the variables change their values while the program runs. Add `print` **statements** at important junctures in the code to see what values the variables hold.

Each `print` statement should also include a brief string that gives context to what is being printed. For example:

```
print("Result pre-loop:", result)
```

Step 2 & 3: Make a Hypothesis; Experiment

At a certain point, you should see something in the values you are printing that is unexpected. At that point, **make a hypothesis** about why the variable is holding that value.

Once you have a hypothesis, **test it** by making an appropriate change in your code. For example, if you think the code never enters an `if` statement, add a print to the beginning of the conditional body to see if it gets printed.

Note: do not change things randomly, even if you get frustrated! Even if it makes you code work on one test, it might start failing another.

Step 4: Observe the Result

Once you've made the change, **observe the result** by checking the new output of your code.

`print` statements are still helpful here. You can also use **variable tables** to see how a variable's behavior changes before vs. after the experiment, by writing out the value in a variable at each juncture of the code by hand.

For particularly tricky code, there are **online visualization tools** that let you see how your code behaves step-by-step. Here's one we recommend: pythontutor.com/visualize.html

Step 5: Repeat As Necessary

Finally, know that you may have to **repeat** the debugging process several times before you get the code to work.

This is normal; sometimes bugs are particularly hard to unravel, and sometimes there are multiple different bugs between your code and a correct solution.

Debugging is Hard

Finally, remember that debugging is hard! If you've spent more than 15 minutes stuck on an error, more effort is not the solution. Get a friend or TA to help, or take a break and come back to the problem later. A fresh mindset will make finding your bug much easier.

Learning Goals

- Use **logical operators** on Booleans to compute whether an expression is True or False
- Use **conditionals** when reading and writing algorithms that make choices based on data
- Use **nesting** of control structures to create complex control flow
- Debug logical errors by using the **scientific method**
- **Feedback: https://bit.ly/110-feedback**