#2-1: Booleans and Conditionals

CS SCHOLARS – PROGRAMMING

Course Logistics

Hw1 feedback has been released!

Let's go over how to view your feedback on Gradescope.

Notes from Hw1

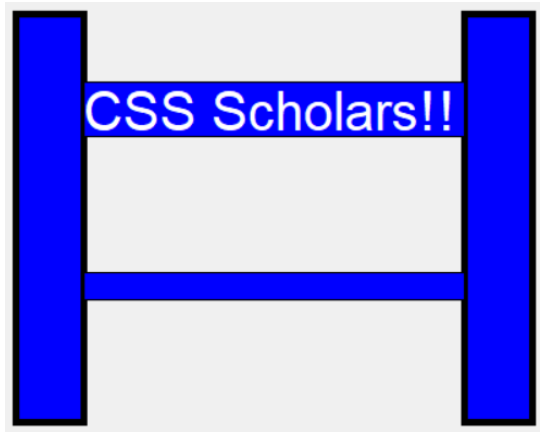
Be specific! [#1]: when communicating an algorithm, make sure you're **specific** about what the user should do. You want to make sure there's no room for misinterpretation!

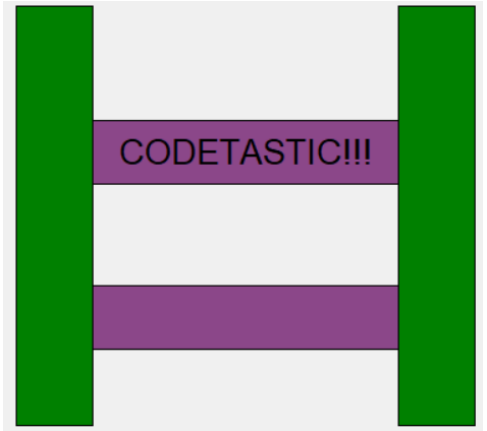
Variable Assignments & Updating [#2]: Recall that **variables can change**! When we use a variable assignment we can change a pre-existing variable to a new value. However, we must actually assign the variable (with =) for the value held in it to change.

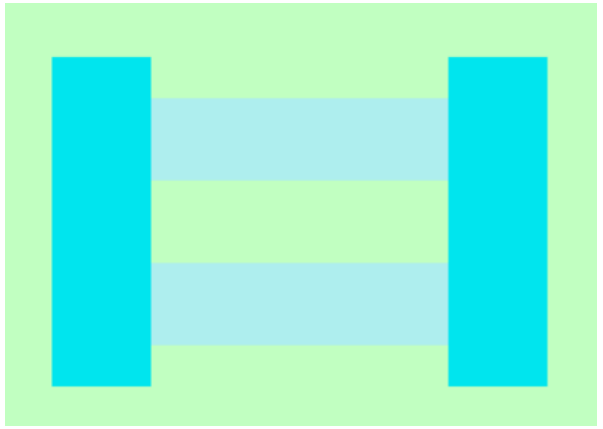
print Returned Value [#3]: `print`'s returned value is **always** `None`, not the value it displays! The action of displaying something is a **side effect**.

Function definition vs. call [#4]: note that there is a difference between **defining** a function vs. **calling** a function. Function definitions have parameters and return statements and are **abstract**; function calls have arguments and returned values and are **specific**.

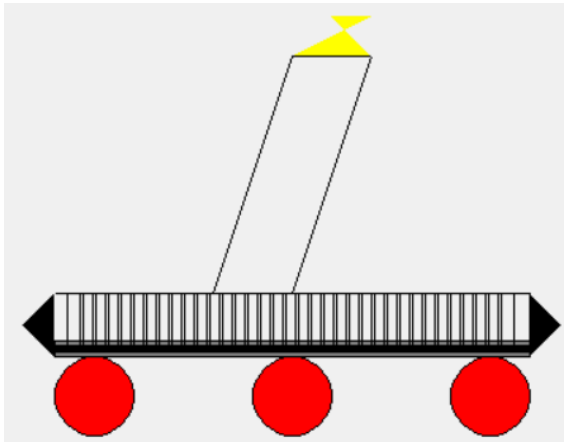
Awesome Graphics!

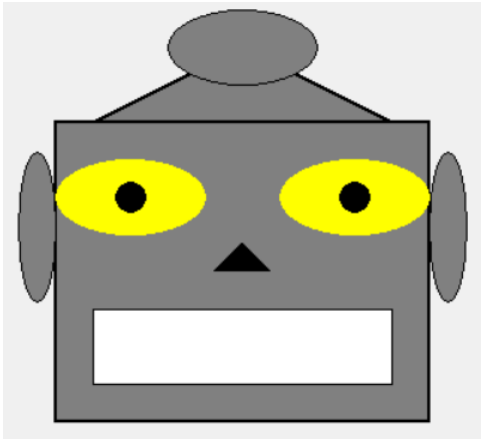


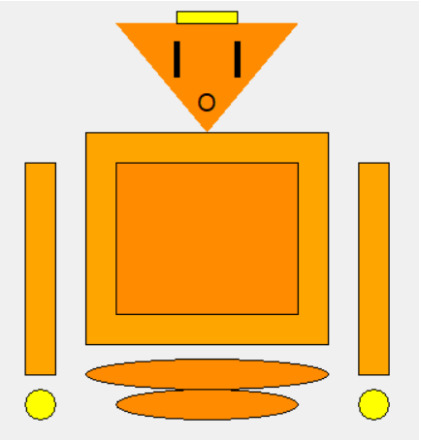


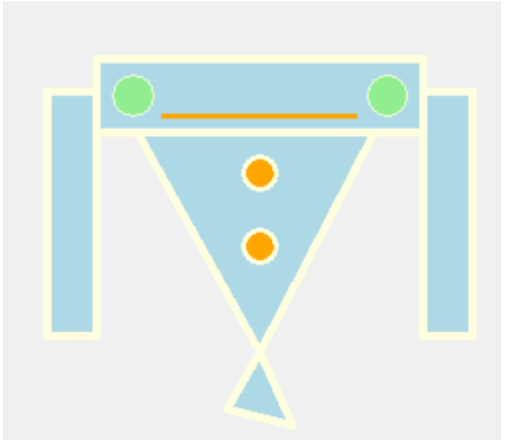


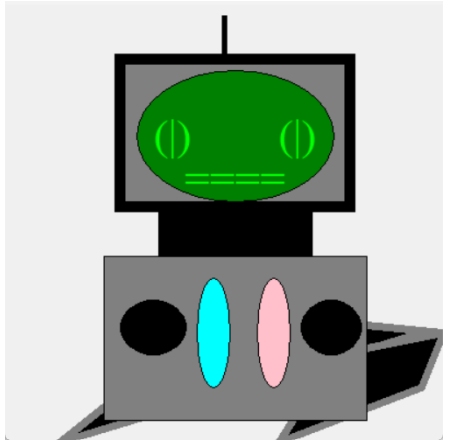
More Awesome Graphics!











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 conditionals and function definitions to create complex control flow

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)
```

Conditionals

Conditionals Make Decisions

With Booleans, we can make a new type of code called a **conditional**. Conditionals are another 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:

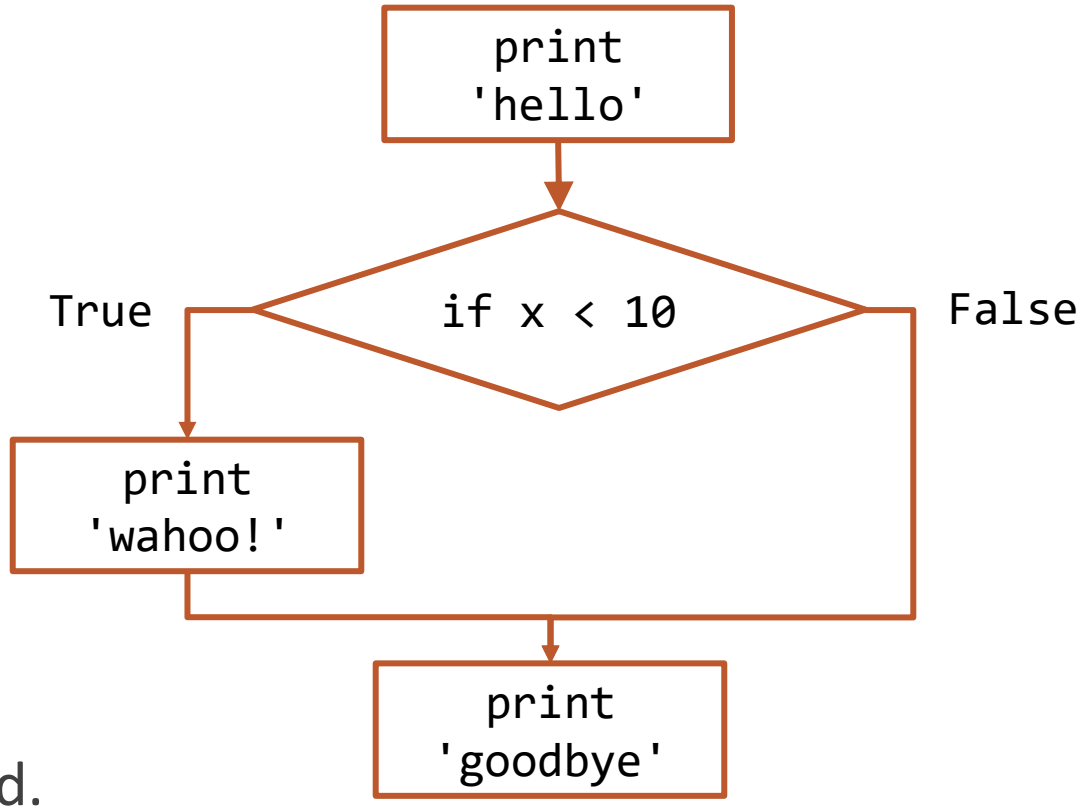
```
if <BooleanExpression>:
    <bodyIfTrue>
```

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. The body must have at least one line and can have as many more lines as it needs.

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.

Example: Print Number of Digits

For example, we could use the following code to print whether a number has one digit or more than one digit:

```
x = 24
if -10 < x and x < 10:
    print("Only one digit")
if x <= -10 or x >= 10:
    print("More than one digit")
```

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`.

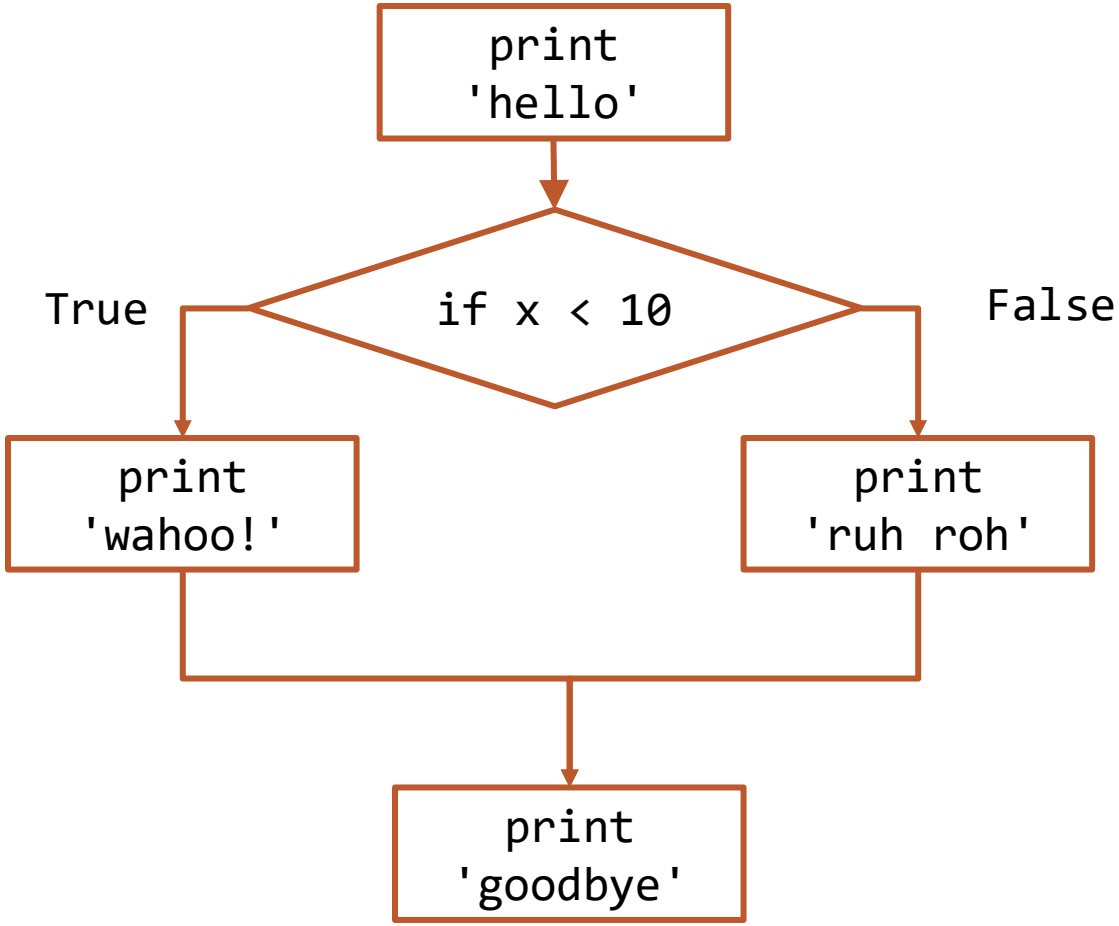
```
if <BooleanExpression>:
    <bodyIfTrue>
else:
    <bodyIfFalse>
```

} `if` clause

} `else` clause

Updated Flow Chart Example

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



Revised Example: Print Number of Digits

Using an else statement makes our earlier code much easier to write and understand!

```
x = 24
if -10 < x and x < 10:
    print("Only one digit")
else:
    print("More than one digit")
```

Activity: Conditional Prediction

Prediction Exercise: What will the following code print?

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

Question: Can we change the **program state** to print the other string instead?

Question: Can we change the state to make the if/else statement print out both statements?

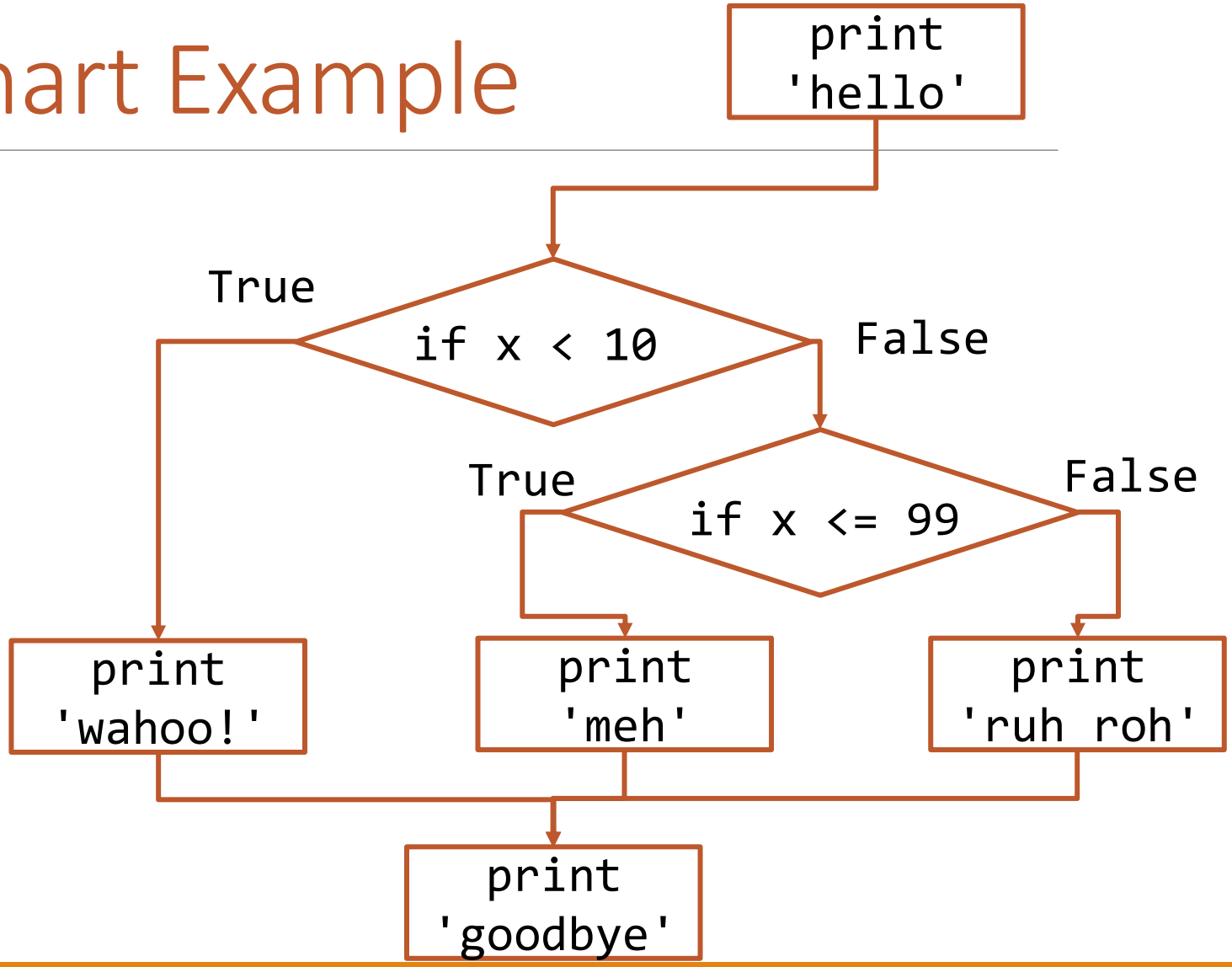
`Elif` Implements Multiple Alternatives

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 Example

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



Conditional Statements Join Clauses Together

A **conditional statement** is a joined group of `if`, `elif`, and `else`. All conditional statements have:

- Exactly one `if` clause
- Followed by zero or more `elif` clauses
- Followed by zero or one `else` clause(s)

These joined clauses can be considered a single **control structure**. Only one clause will have its body executed.

Note that it's impossible to have an `else` or `elif` clause by itself, as it would have no condition to be the alternative to. That means we always need an `if` at the beginning.

Example: grade calculator

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.

Activity: calculate late fee

You do: write a few lines of code that determine whether a library book is late. If it isn't, print out a message saying that everything is fine; if it is late, print out the late fee.

Start with a few variables. `maxDays` is the number of days a book is allowed to be checked out; set it to `30`. `dailyFee` is the fine per day once a book is late; set it to `10` (10 cents). `daysPassed` can then be the number of days that you've had the book checked out. Try changing that variable to different values as you run the code to see what happens!

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 type(x) == type(y) and x < y:
    print("Smaller:", x)
```

Activity: Kahoot!

Let's do a quick Kahoot to practice evaluating Boolean expressions that may or may not use short-circuit evaluation.

Join the Kahoot here: kahoot.it

Nesting Control Structures

Nesting Creates More Complex Control Flow

Now that we've learned more, **we can put `if` statements inside of `if` statements**.

In general, we'll be able to **nest** control structures inside of other control structures. This can currently be done with `if` statements and function definitions.

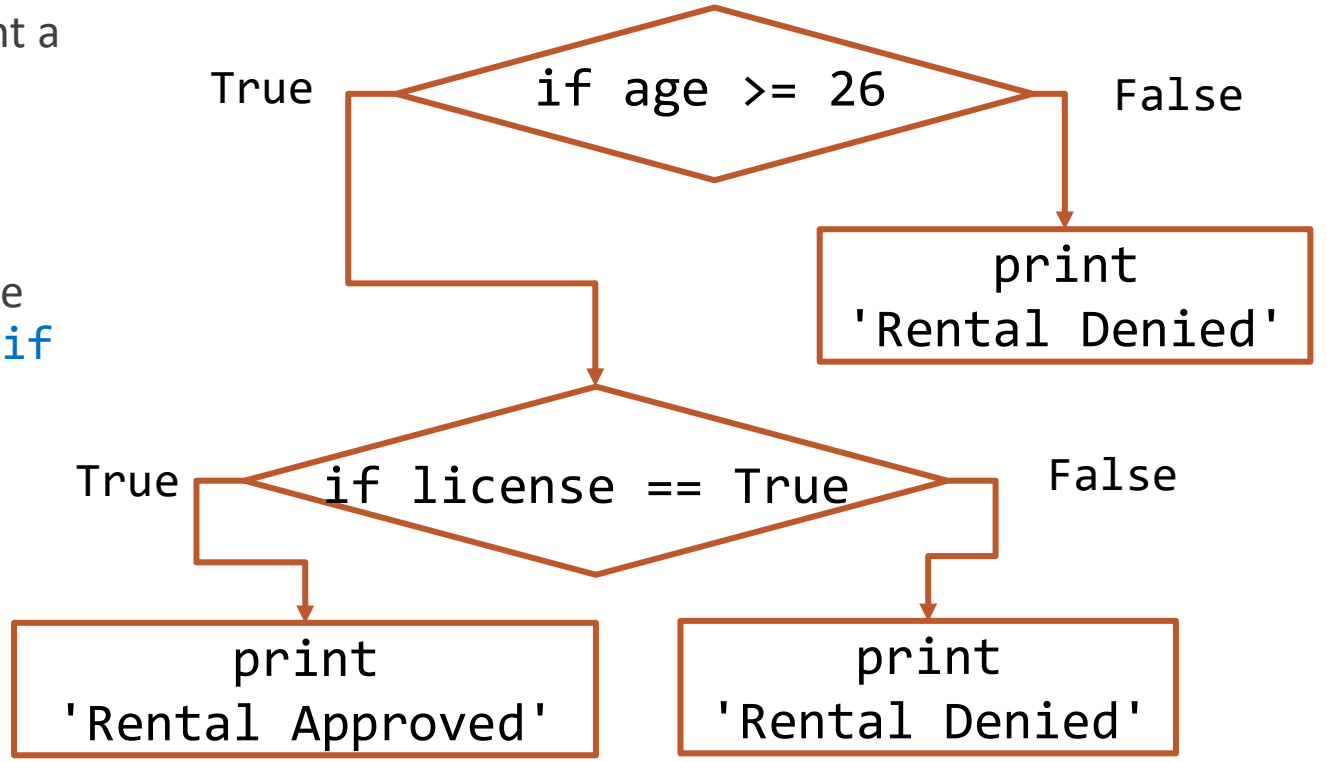
In program syntax, we demonstrate that a control structure is nested by **indenting the code** so that it's in the outer control structure's body.

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")
```



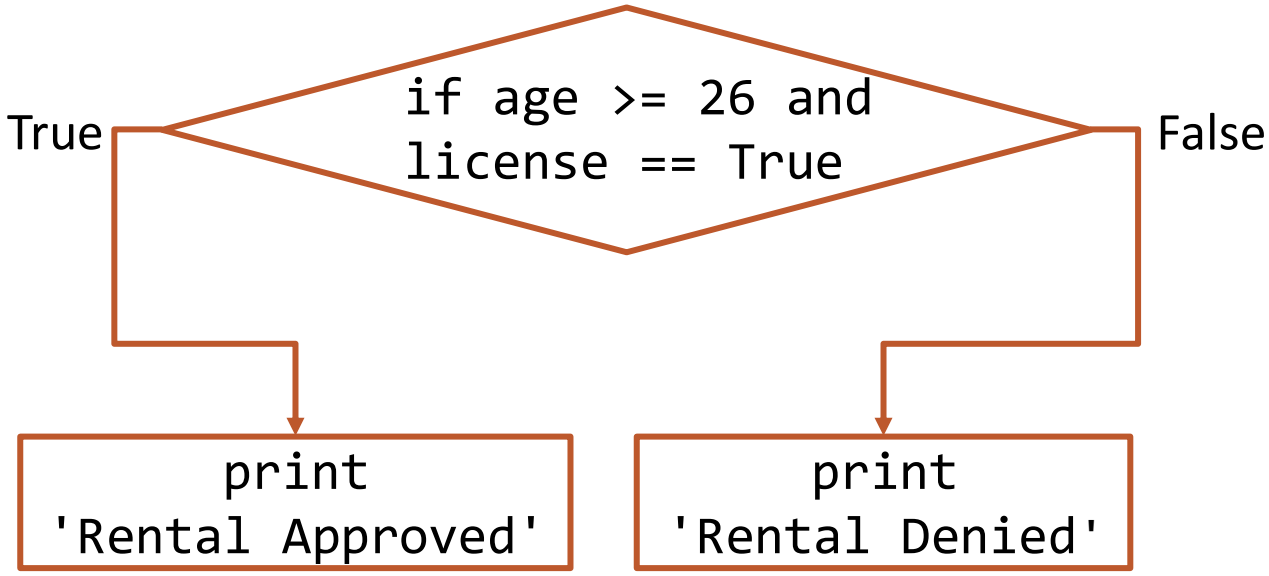
Note that each else is paired with the if at the **same indentation level**.

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")
```



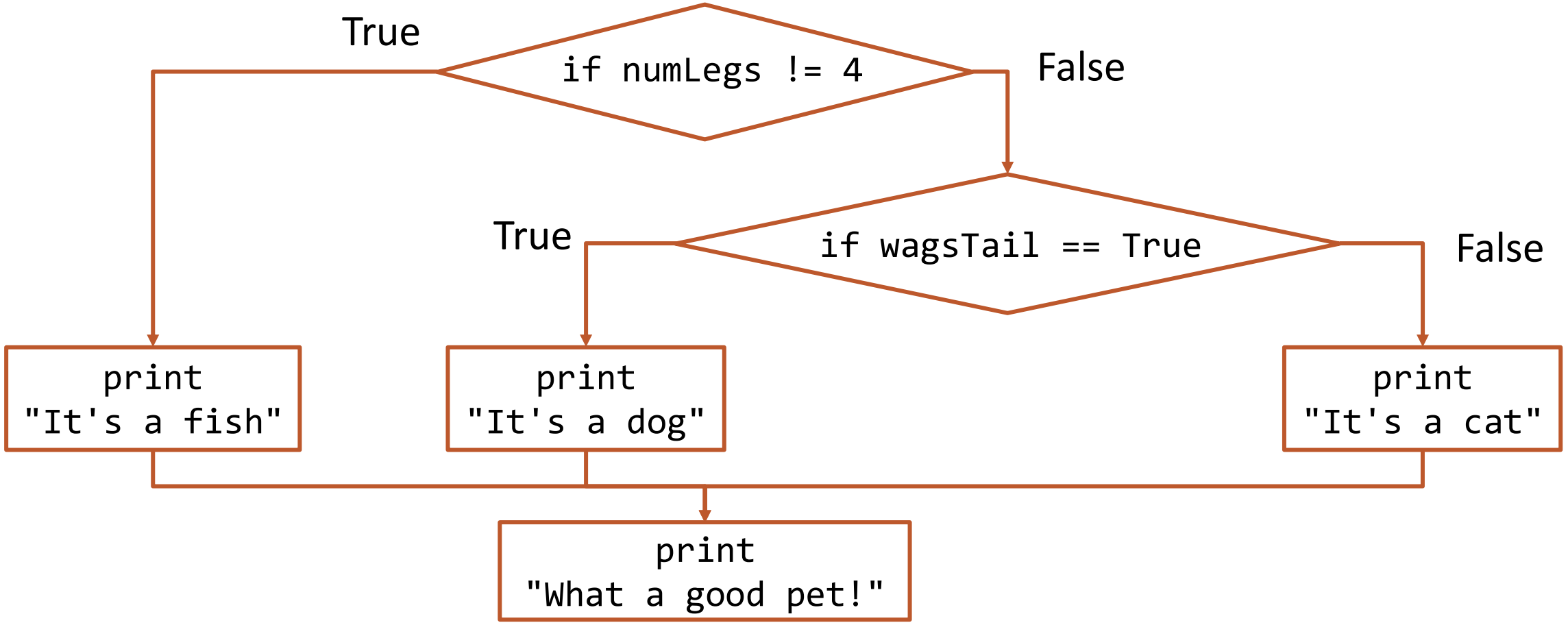
Nesting Conditionals in Functions

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 -1 # error code
    return total / n
```

Exercise: Convert Flow Chart to Code



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 conditionals and function definitions to create complex control flow

Recognize the different types of **errors** that can be raised when you run Python code