#2-2: Booleans, Conditionals, and Errors

CS SCHOLARS – PROGRAMMING

Course Logistics

Hw1 feedback has been released!

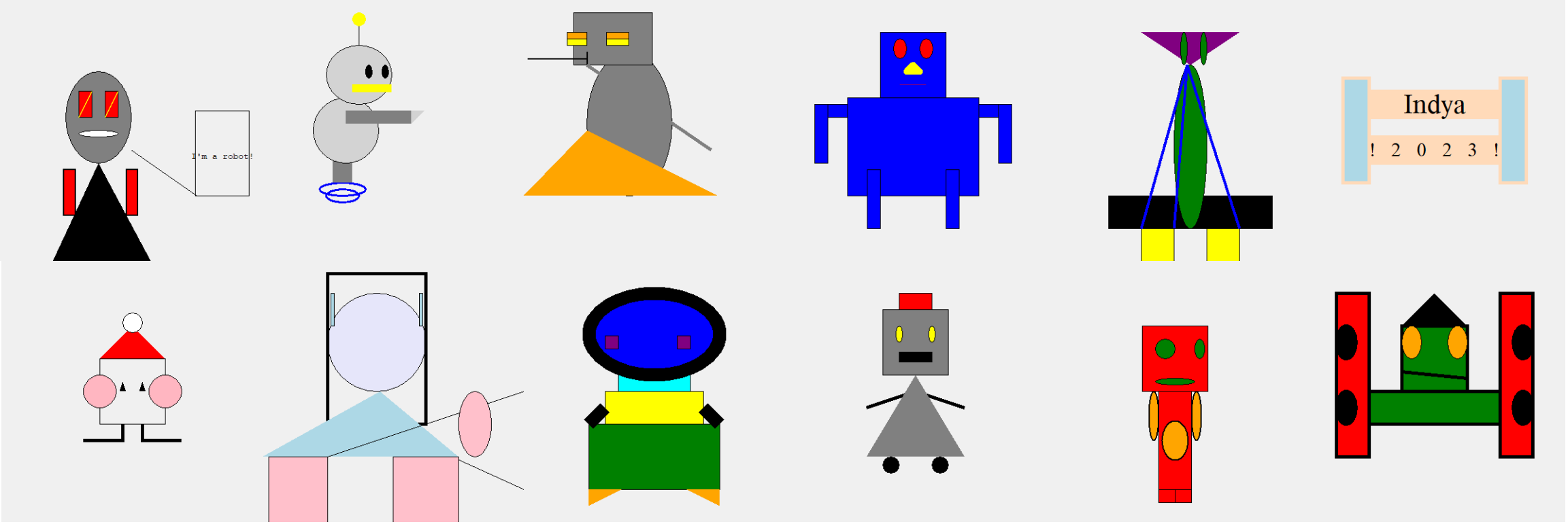
Let's go over how to view your feedback on Gradescope.

Notes from Hw1

Recap on print (returned value vs. side effect)

Make sure to read instructions carefully! (high abstraction)

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

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

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:

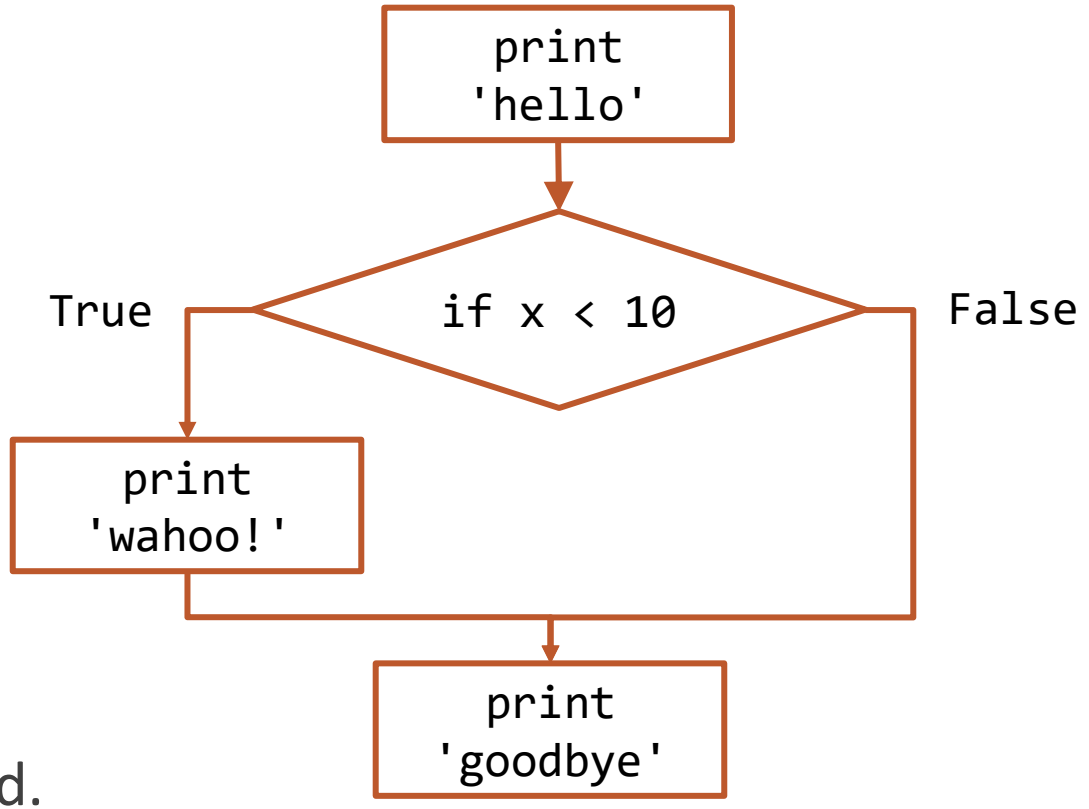
```
if <BooleanExpression>:
    <bodyIfTrue>
```

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.

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`.

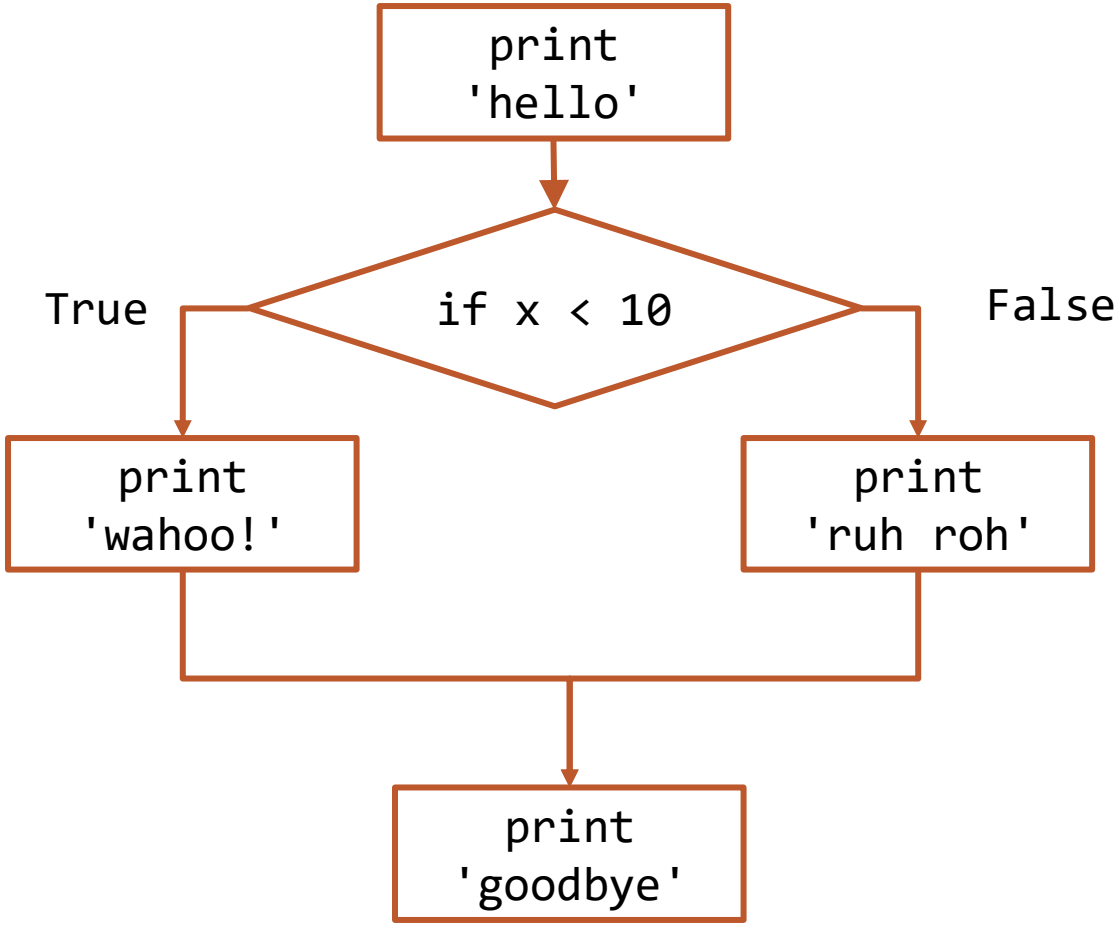
```
if <BooleanExpression>:
    <bodyIfTrue>
else:
    <bodyIfFalse>
```

} `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: How 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?

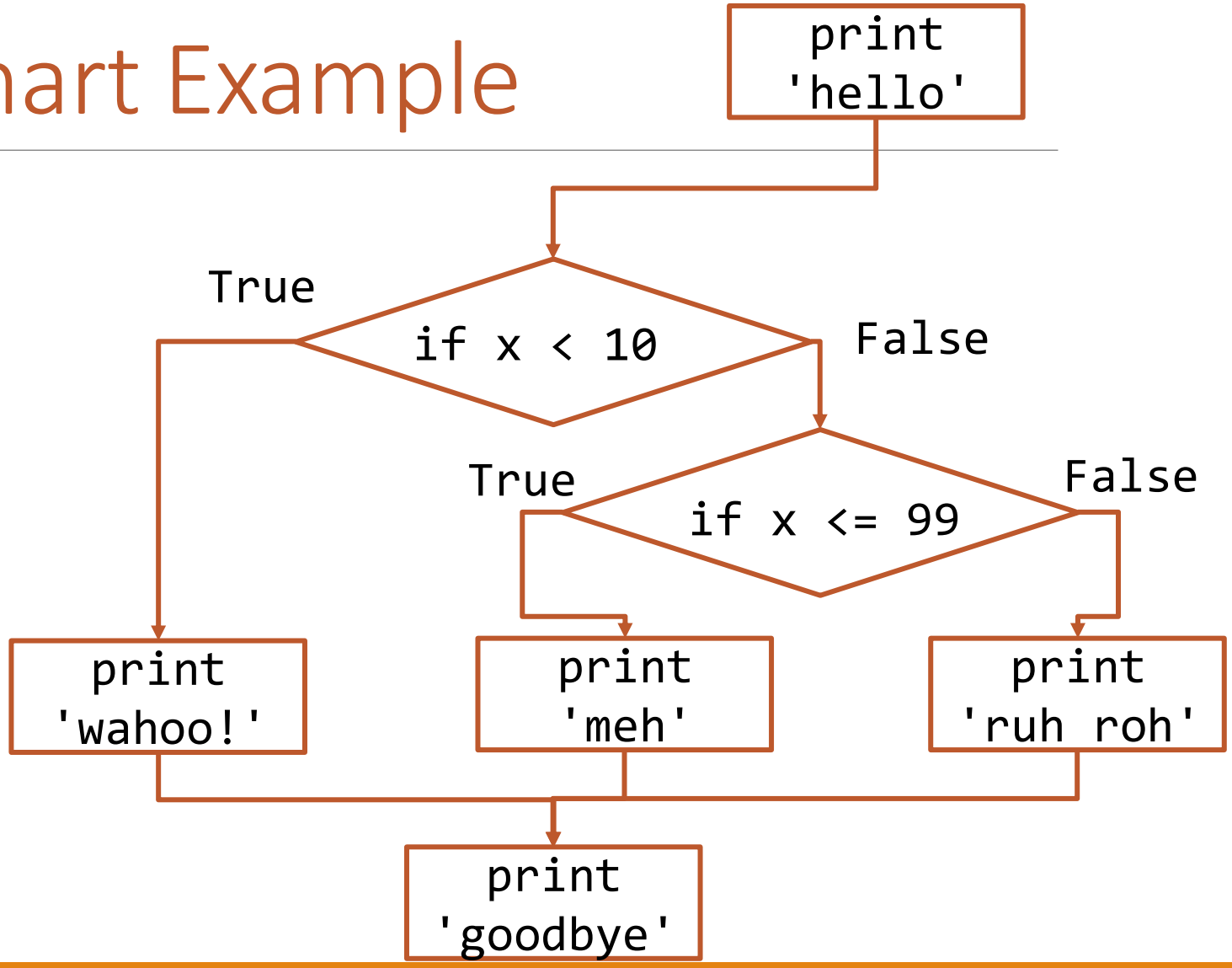
`Elif` Implements Multiple Alternatives

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

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`).

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

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 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. This can currently be done with `if` statements and function definitions.

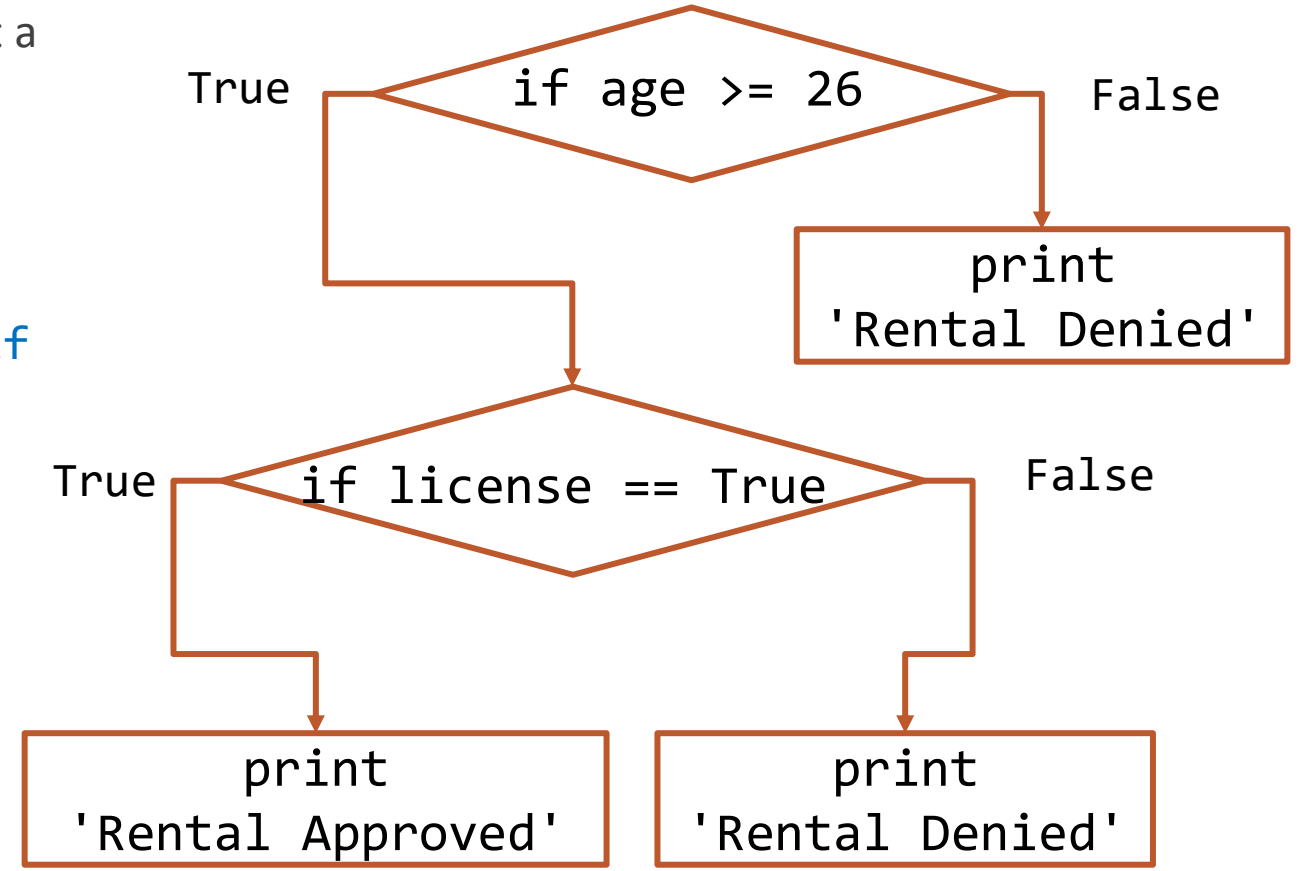
In program syntax, we demonstrate that a control structure is nested by **indenting the code** so that it's in the outer control structure's body.

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")
```

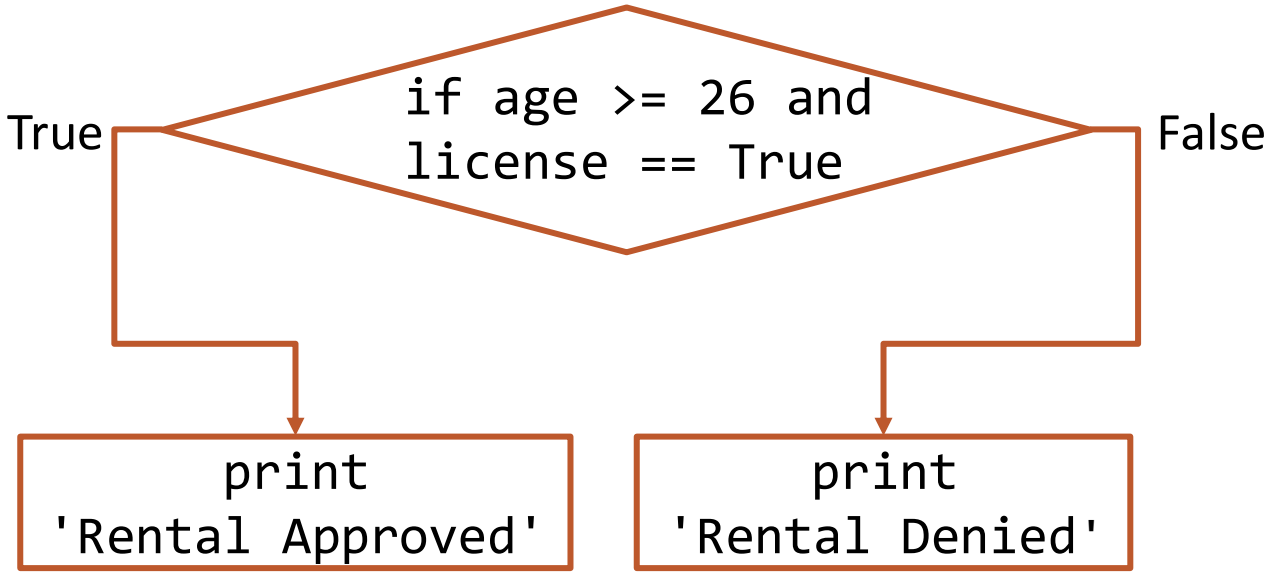


Alternative Car Rental Code

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 occurs between the outer clauses!

```
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?

In general, an outer `if`/`elif`/`else` statement **cannot** come between parts of an inner conditional.

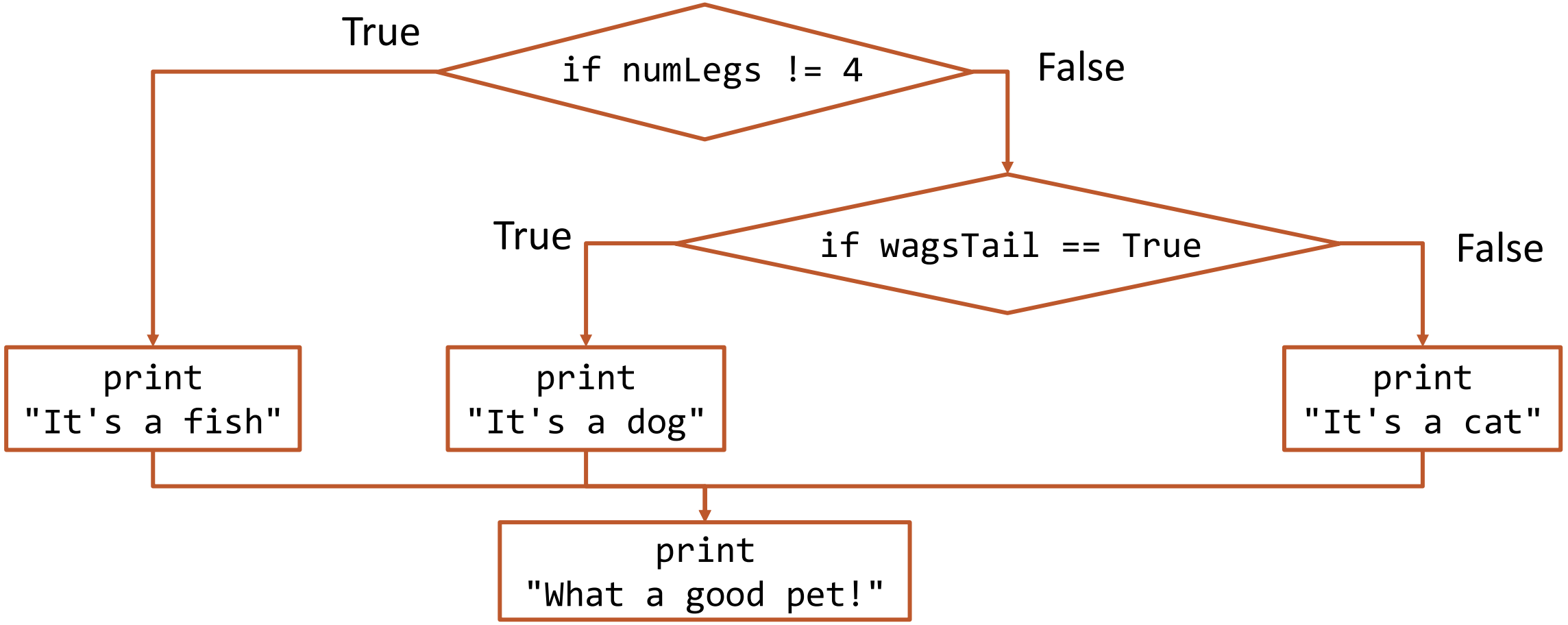
Nesting Conditionals in Functions

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



Python Errors

Syntax Errors Occur due to Bad Syntax

When Python executes your code, it first has to break your text down into **tokens**, then **structure** those tokens into a format that the computer can execute.

The programming language's **syntax** is a set of rules for how code instructions should be written. When syntax is correct, Python is able to tokenize and structure code without a problem.

If the interpreter runs into an error while tokenizing or structuring, it calls that a **syntax error**. In other words, you get a syntax error when the code you provide does not follow the rules of the Python language's syntax.

A syntax error means that **none of your code will run**, because the syntax can't be parsed.

Examples of Syntax Errors

Most syntax errors are called **SyntaxError**, which make them easy to spot. For example:

```
x = @      # @ is not a valid token
4 + 5 = x # the parser stops because it doesn't follow the rules
```

There are two special types of syntax errors: IndentationError and incomplete error.

```
    x = 4    # IndentationError: whitespace has meaning
print(4 + 5 # Incomplete Error: always close parentheses/quotes
```

Execution Errors are Runtime Errors

After Python tokenizes and structures the code, the interpreter runs through the control flow of the program line-by-line.

If an error occurs as the code is being executed, it's called a **runtime error**. Everything that happened before that error will execute just fine, but everything afterwards will not run.

Runtime errors have many different names in Python. Each name says something about what kind of error occurred, so reading the name and text can give you additional information about what went wrong.

Examples of Runtime Errors

```
print(Hello) # NameError: used a missing variable

print("2" + 3) # TypeError: illegal operation on types

x = 5 / 0 # ZeroDivisionError: can't divide by zero
```

We'll see more types of runtime errors as we learn more Python syntax.

Other Errors are Logical Errors

If we manage to run Python code completely, does that mean it's correct?

Not necessarily! **Logical errors** can occur if code runs but produces a result that was not what the user intended. The computer can't catch logical errors because the computer doesn't know what we intend to do.

To catch logical errors, you usually need to **test** your code. We'll do this mainly with `assert` statements.

Examples of Logical Errors

```
print("2 + 2 = ", 5) # no error message, but wrong!


def double(x):
    return x + 2 # adding instead of multiplying

assert(double(3) == 6) # 6 is the intended result
```

`assert` Statements Check Correctness

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.

We use `assert` statements to check for logical errors by testing whether the output of a function call is equal to what we expect it to be. If the result is not correct, you get an `AssertionError`.

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

Activity: Predict the Error Type

Let's test your knowledge of error types with another Kahoot!

Given a line of code, predict whether it will result in a Syntax Error, Runtime Error, Logical Error, or no error.

If you aren't sure, try to think about whether the problem will occur during syntax parsing/structuring, or execution, or if it will run properly but still have a problem.

Join at kahoot.it

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