Booleans, Conditionals, and Errors

15-110 – Monday 1/29

Announcements

- Hw1 was due today
- Check2/Hw2 now released
- Quizlet1 is on Wednesday

Quizlets

- There are 9 quizlets throughout the semester on Wednesdays
  - Lowest **two** scores dropped
- Procedure:
  - Bring a piece of paper
  - You’ll have 5 minutes to answer the question displayed on the screen
  - No computers, phones, notes, or collaboration
  - When time is up, take a picture and upload to Gradescope
    - If you have trouble getting it uploaded, you can hand in your paper instead
- **Demo**: Practice uploading to Gradescope

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
- 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 or x > 15

not (x > 5 and x <= 10)

(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 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:

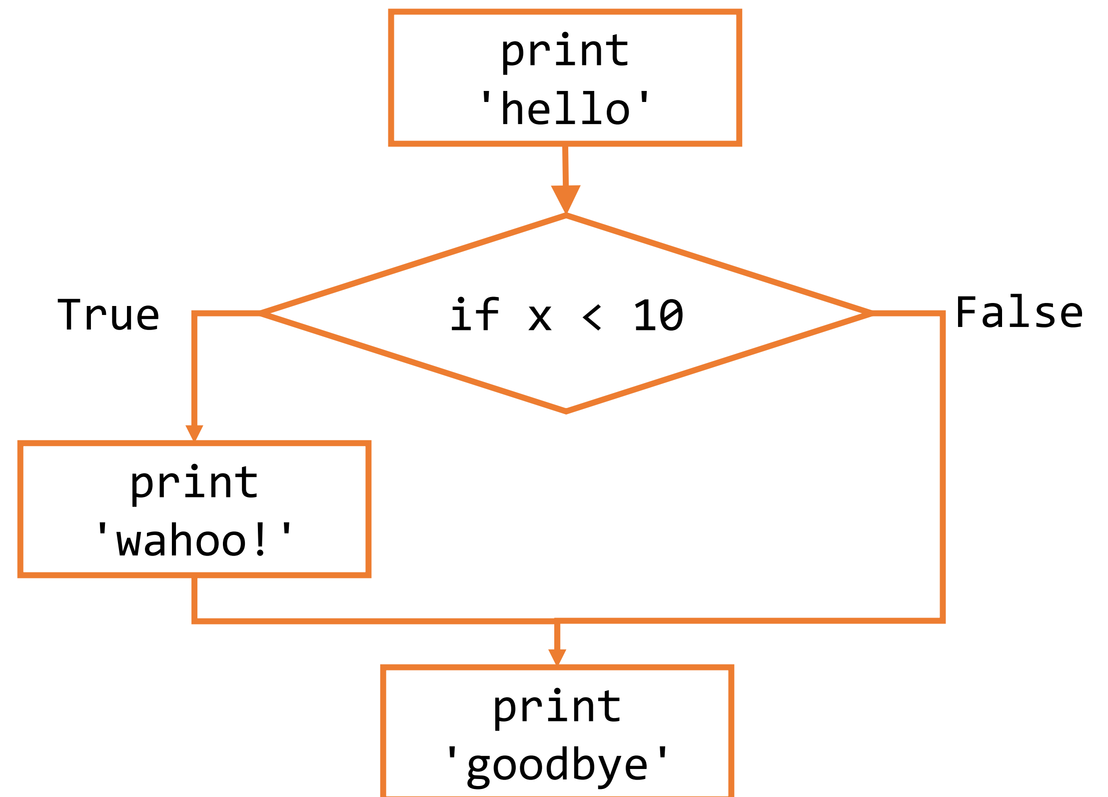
```
if <BooleanExpression>:
    <bodyIfTrue>
```

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. 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`.

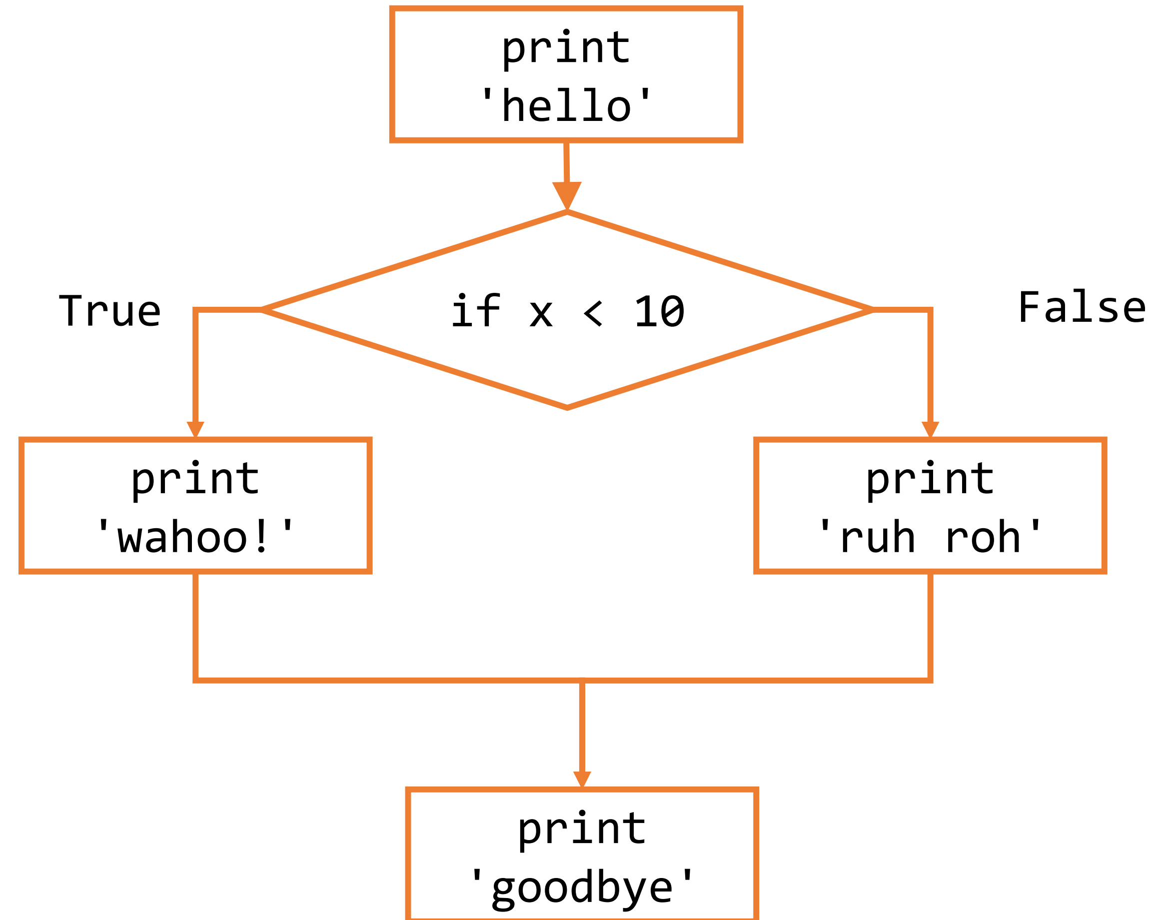
```
if <BooleanExpression>:
    <bodyIfTrue>
else:
    <bodyIfFalse>
```

`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?

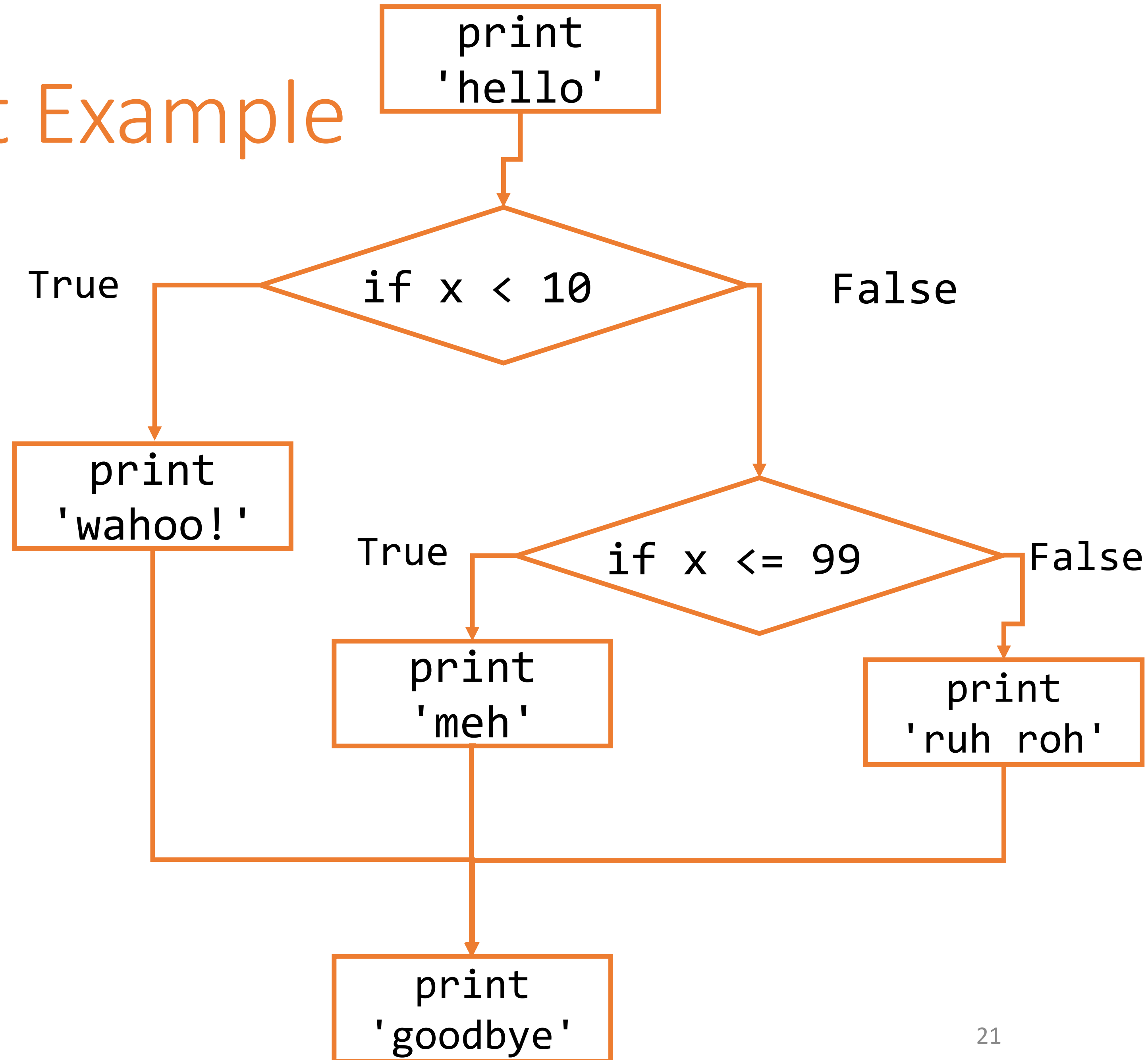
`Elif` Implements Multiple Alternatives

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.

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)
```

Two New Math Operators

When we write algorithms using control structures, we may want to check whether a number has certain properties (like being even or a multiple of ten). We can do this using some new operators.

Modulo, or **mod** (`%`) finds the remainder when one number is divided by another.

For example, `7 % 4` is equal to `3`.

Check if a number is even with `x % 2 == 0`.

Integer division, or **div** (`//`) divides numbers by rounding down to nearest whole number. This cuts off any digits after the decimal point.

For example, `7 // 4` is equal to `1`, not `1.75`.

Cut off the last digit of a number with `x // 10`.

Nesting Control Structures

Nesting Creates More Complex Control Flow

Now that we have another 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 conditional statements and function definitions.

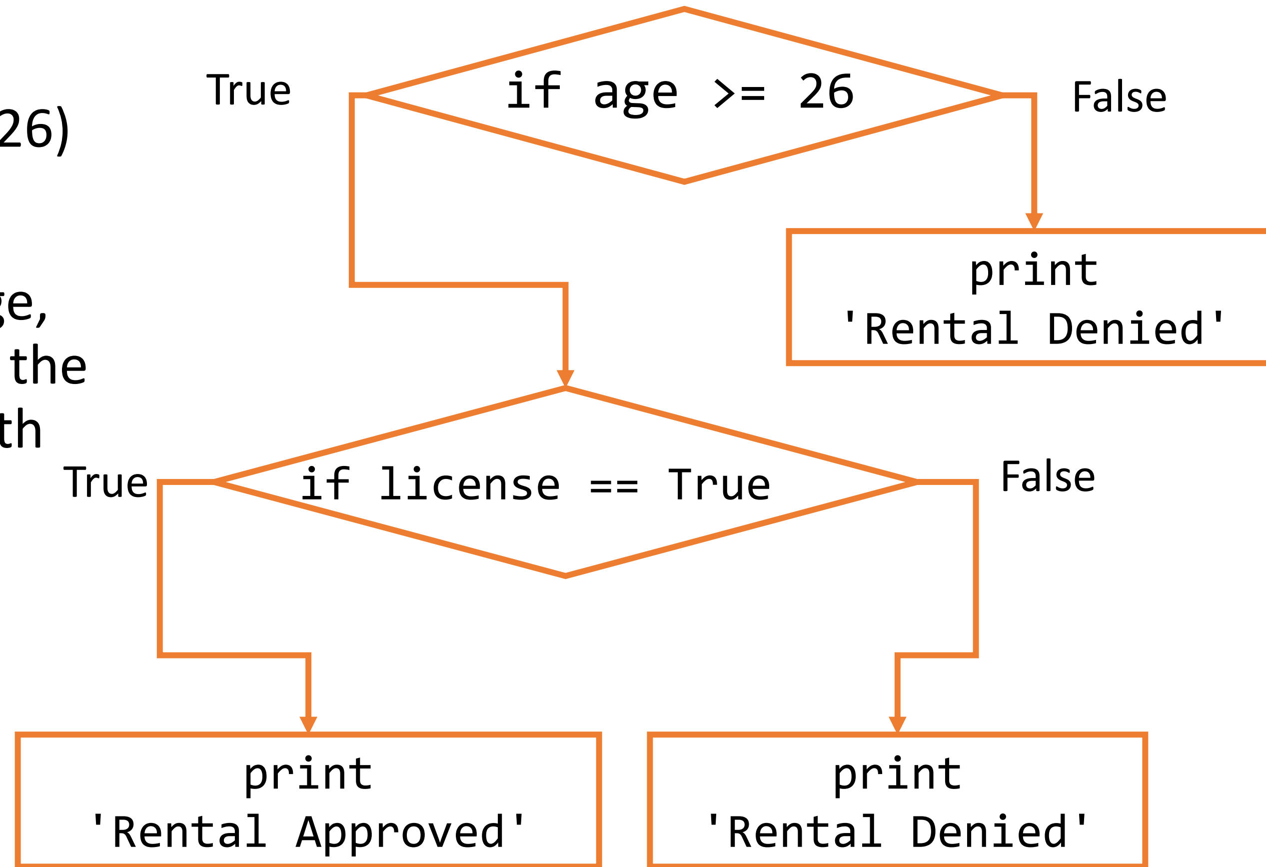
In program syntax, we demonstrate that a control structure is nested by **indenting the code** so that it's in the outer control structure's body.

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")
```



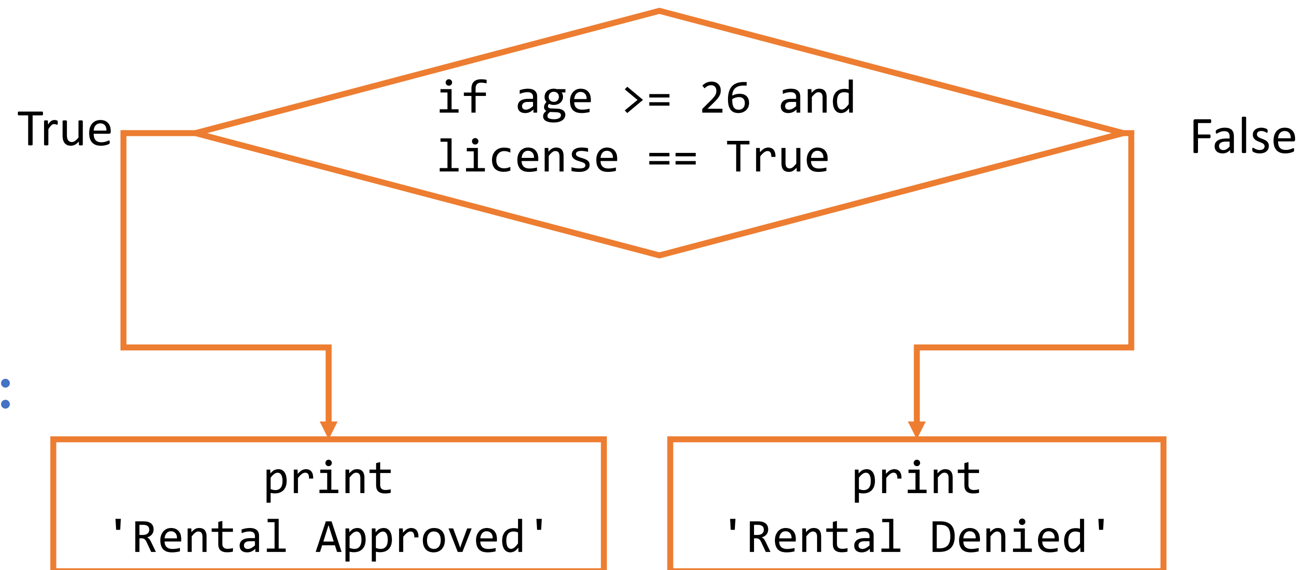
Note that each else is paired with the if at the **same indentation level**.

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
```

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 can 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, either because an invalid token is used or because tokens are structured in a way Python cannot understand.

A syntax error means that **none of your code will run**, because the syntax can't be parsed. Syntax errors should be fixed as quickly as possible!

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.

Syntax vs Runtime

What's the difference between syntax and runtime errors?

Syntax errors: Python cannot correct parse the syntax of the text, so none of the code will run.

- Example: saying "I take the snarkledoo to work". Snarkledoo is not a word, so the sentence doesn't parse.
- Another example: saying "bus I work to the take". The words are all valid but not in the right order, so the sentence doesn't parse.

Runtime errors: Python parses the code and starts to run, but gets to a point where the code cannot be computed. Anything after the non-working code will not run.

- Example: saying "I go to work from Monday to Friday. I take the giraffe to work. I get a hot chocolate before starting the day." All sentences parse. The first sentence runs fine, but the second would cause a runtime error (it doesn't make sense), so the third would not be processed.

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.

`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)
```

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
```

Logical errors are the hardest to find and fix. You'll learn more about how to debug them in recitation this week.

Demo: Programming Starter File

```
def testAll():
    testNumSign()
    testFlowChart()
    runInteractiveProgram()

testAll()
```

Starting in Check2, the programming starter files will contain test cases that use assert statements.

To run all the tests, click the **Run current script** button. This will run the whole file and call `testAll()` at the bottom, which will run every test function.

If you want to skip forward, you can turn off the tests for a single problem by commenting out the call to `testProblem` in the `testAll` definition body. Alternatively, if you want to test a single problem, you can run `testProblem()` in the interpreter to automatically see the results for just that problem.

Note that for some tests (like `runInteractiveProgram`) you need to check the results yourself! Read the test output to make sure your work is correct.

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