Booleans, Conditionals, and Errors

15-110 – Monday 09/09

Announcements

- Hw1 was due today
  - Start homeworks early!
- Check2 now released, HW2 coming out on Wednesday
  - Note that Check2 has written and programming components!
- Quizlet1 is on Wednesday
- Go to recitation; it contains material that won't be in the slides!

Quizlets

- There are 8 quizlets throughout the semester on Wednesdays
  - Lowest **two** scores dropped
- Procedure:
  - Bring a full-sized piece of paper (printer paper preferred)
  - You'll have 5 minutes to answer the question displayed on the screen
  - No computers, phones, notes, or collaboration etc.
  - When time is up, you **must** stop writing immediately and hand all papers **face-down** to the nearest aisle
- You must be *silent* during the quizlet and while it is being collected. You also must not talk to anyone about the quizlet until it has been graded and released.

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

Python can evaluate whether certain expressions are true or false. These types of values are called **Booleans**.

Booleans are either `True` or `False`

We get a Boolean when we do a comparison with **comparison operators**:

less than `<`	greater than `>`	equal `==`
less or equal `<=`	greater or equal `>=`	not equal `!=`

```
>> “Hello” == “World”
False
```

We can combine boolean values using **logical operators**.

or `or`	not `not`	and `and`

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

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

and operator takes two Boolean values and evaluates to **True if both values are True**.

We use **true tables** to show the output for a boolean expression given all possible inputs.

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

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

`or` operator takes two Boolean values and evaluates to **True if either value is True**.

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

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

`not` operator takes a Boolean value **negates it**, switching it to the opposite value.

a	not a
True	False
False	True

We use `not` to **switch the result** of a Boolean expression.

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))
```

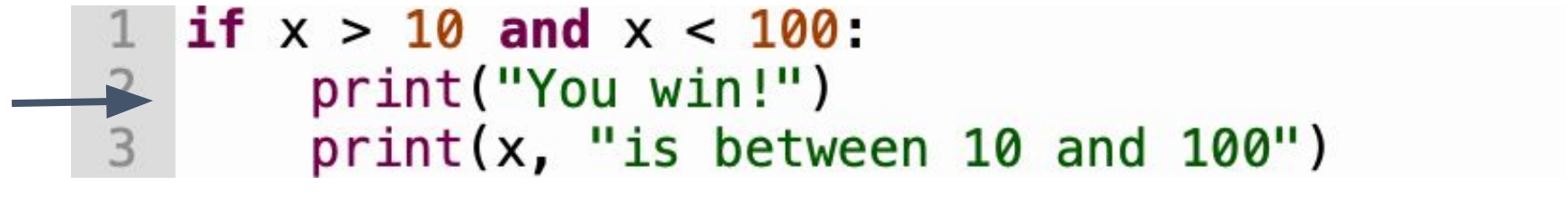
Conditionals

Conditionals are a **control structure**: they let us make decisions based on **boolean expressions**.

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

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

indentation (tab)



```
1 if x > 10 and x < 100:
2     print("You win!")
3     print(x, "is between 10 and 100")
```

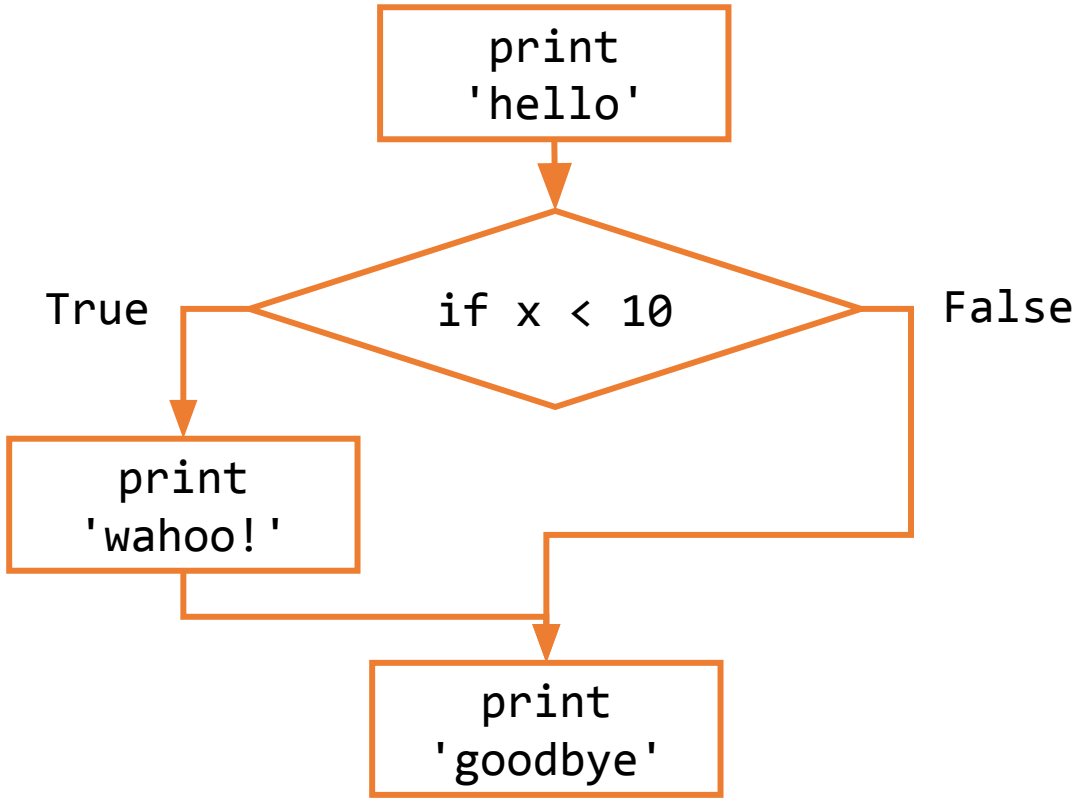
`if` is how Python knows this is an if statement

`:` and indentation is start of **if statement body**

We will use a **flowchart** 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")
```

If we want a program to do one of **two alternative actions** based on a condition we can write an `else` **statement**.

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`.

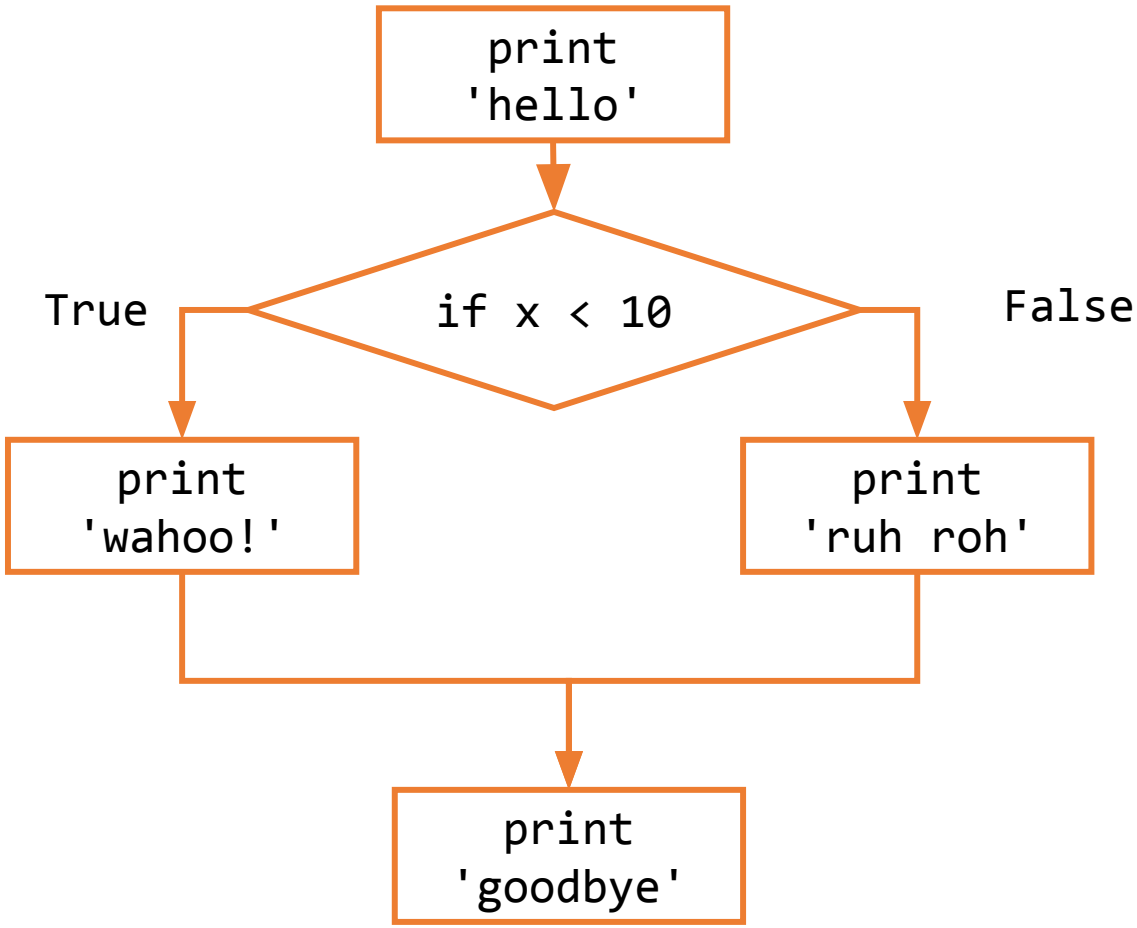
```
if x > 10 and x < 100:
    print("You win!")
    print(x, "is between 10 and 100")
else:
    print("You lose!")
```

} `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 value of `x` to print the other string instead?

Question: Can we change the value of `x` to make the if/else statement print out both statements?

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`.

```
1  if x > 10 and x < 100:
2      print("You win!")
3      print(x, "is between 10 and 100")
4  elif x > 100:
5      print("You almost win!")
6  else:
7      print("You lose!")
```

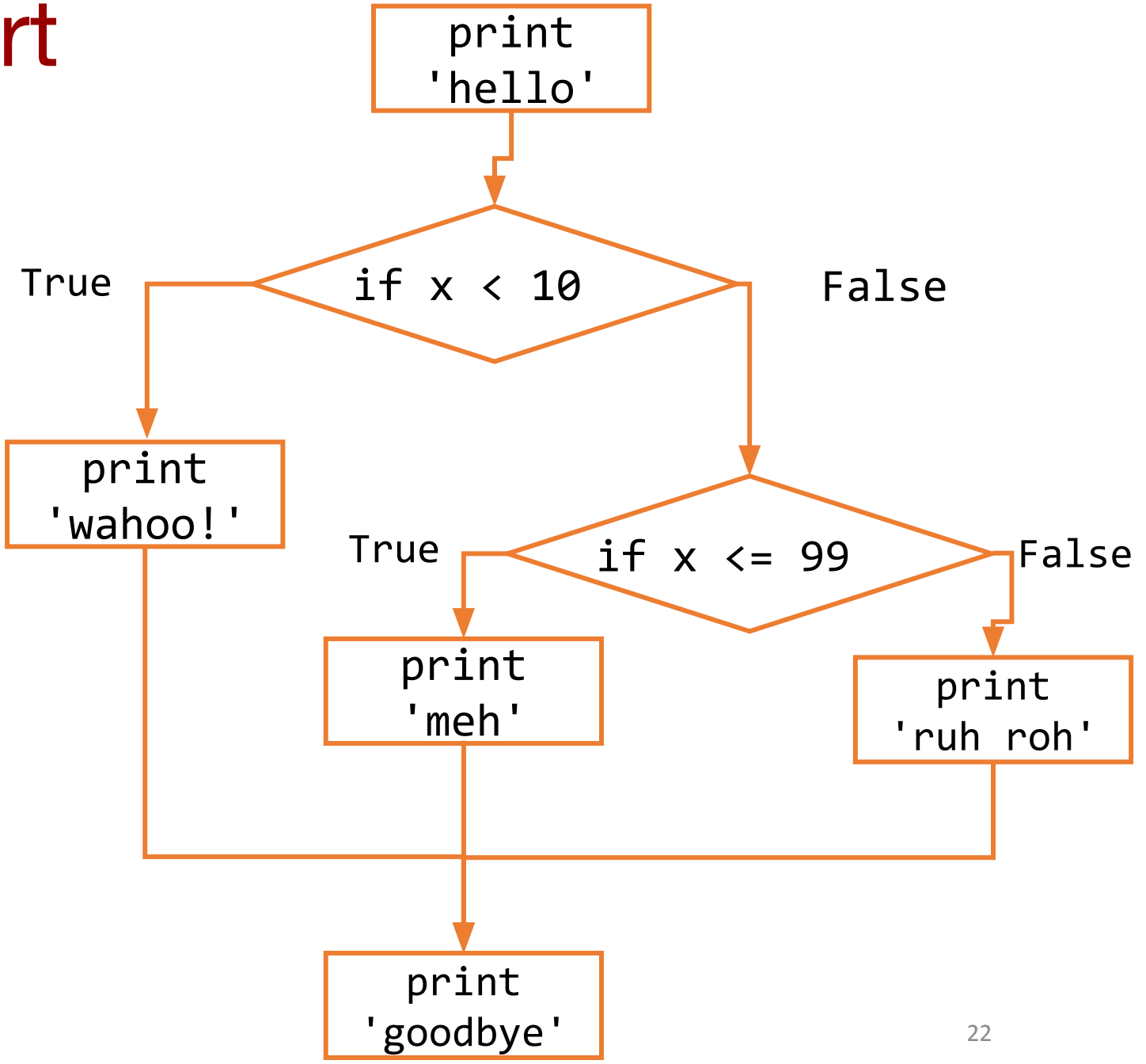
} `if` clause

} `elif` clause

} `else` clause

Updated Flow Chart Example

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



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)

It must start with an `if`. You cannot have an `elif` or `else` without an `if`.

These joined clauses can be considered a single control structure: **only one clause will execute**!

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 is when Python **does not evaluate the second part** of a boolean expression because it logically does not need to.

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

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

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

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

Short-circuit evaluation is handy for **keeping errors from happening**!

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

if x != 0 and 10 / x:
    print(x, "is not 0")
```

There are two math operators that are handy for **checking if numbers have certain properties**: mod `%` and div `//`

modulo `%`	Finds the remainder when one number is divided by another.

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

integer divide `//`	Divides numbers by rounding down to nearest whole number. This cuts off any digits after the decimal point.

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

Nesting Control Structures

We can **nest control structures** inside of other control structures.

Example: We can put `if` statements inside of `if` statements.

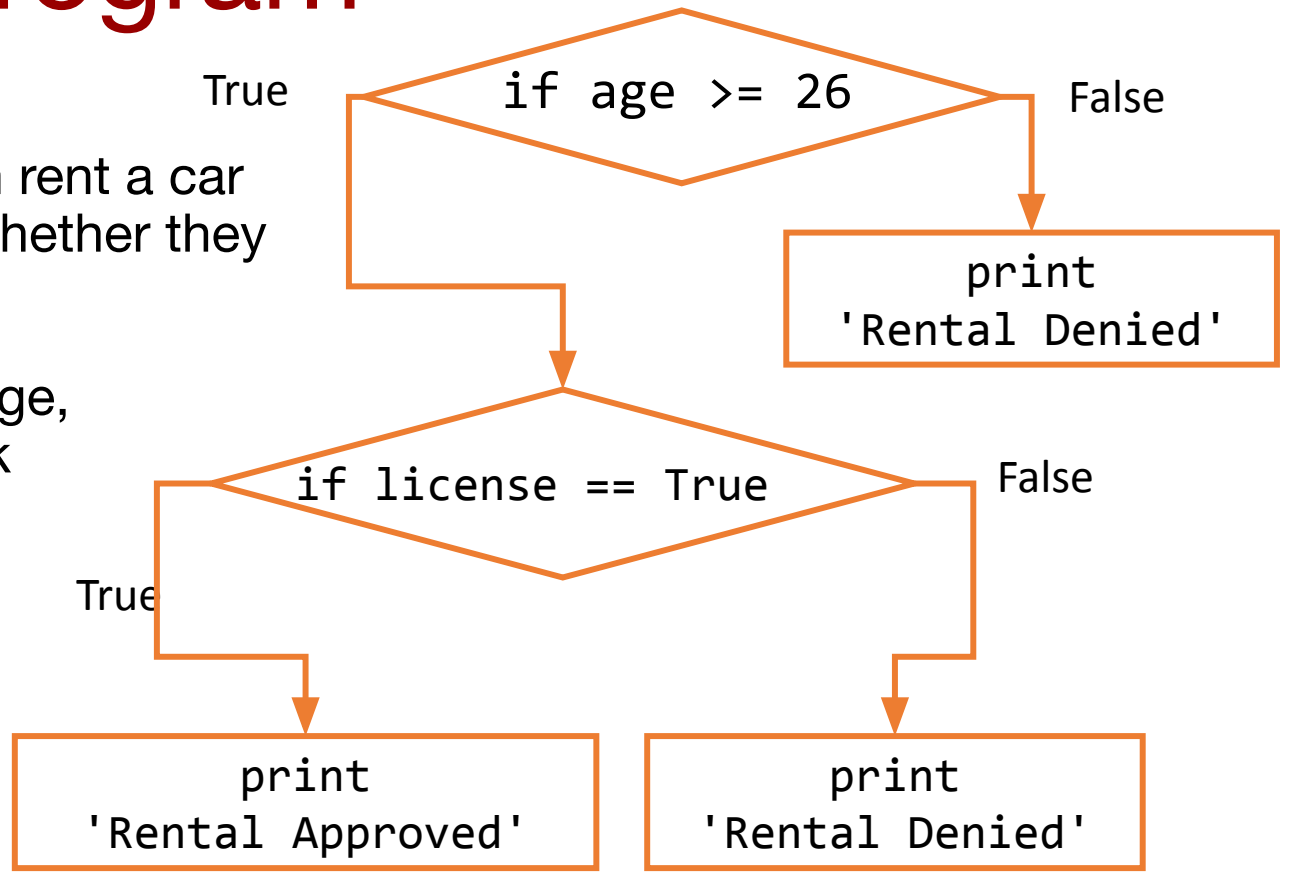
```
if age >= 26:
    if license == True:
        print("Rental Approved")
    else:
        print("Rental Denied: Invalid License”)
else:
    print("Rental Denied: Must be at least 26 years old")
```

In program syntax, we demonstrate that a control structure is nested **by indenting the code** so that it is 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 he 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: Invalid License")
else:
    print("Rental Denied: Must be at least 26 years old")
```

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

When we **nest a conditional inside a function definition**, we can return values early instead of only returning on the last line.

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
```

A function will **always end as soon as it reaches a `return` statement**, even if more lines of code follow it.

Python Errors

Errors happen when syntax is not valid Python code.

```
>> Print(“Hello World”)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'Print' is not defined

>> print “Hello World”
  File "<stdin>", line 1
    print "Hello World"
    ^^^^^^^^^^^^^^^^^^^
SyntaxError: Missing parentheses in call to
'print'. Did you mean print(...)?
```

There are **3 kinds of errors**:

Syntax Errors: When code does not follow Python grammar rules

```
print(1,2
```

Runtime Errors: When code crashes during program execution

```
print(1/0)
```

Logical Errors: When code runs but output is not what we expect

```
def printHello():
    print(“Goodbye”)
```

The programming language's **syntax** is a set of rules for how code instructions should be written.

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.

A **syntax error** occurs when the interpreter runs into an error while tokenizing or structuring, the code does not follow Python language's syntax rules.

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

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

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

There is a special type of syntax error: **IndentationError**

```
    x = 4     # IndentationError: whitespace has meaning
if x > 4:     # IndentationError: missing if statement body
```

If an error occurs as the code is being executed, it's called a **runtime error**.

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

Runtime errors have many different names in Python. Examples:

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

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

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

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.

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.

The third kind of error: **Logical errors** can occur if code runs but produces a result that was not what the user intended.

The computer cannot catch logical errors because the computer doesn't know what we intend to do!

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

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**.

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 with an `AssertionError`.

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

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

```
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
```

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