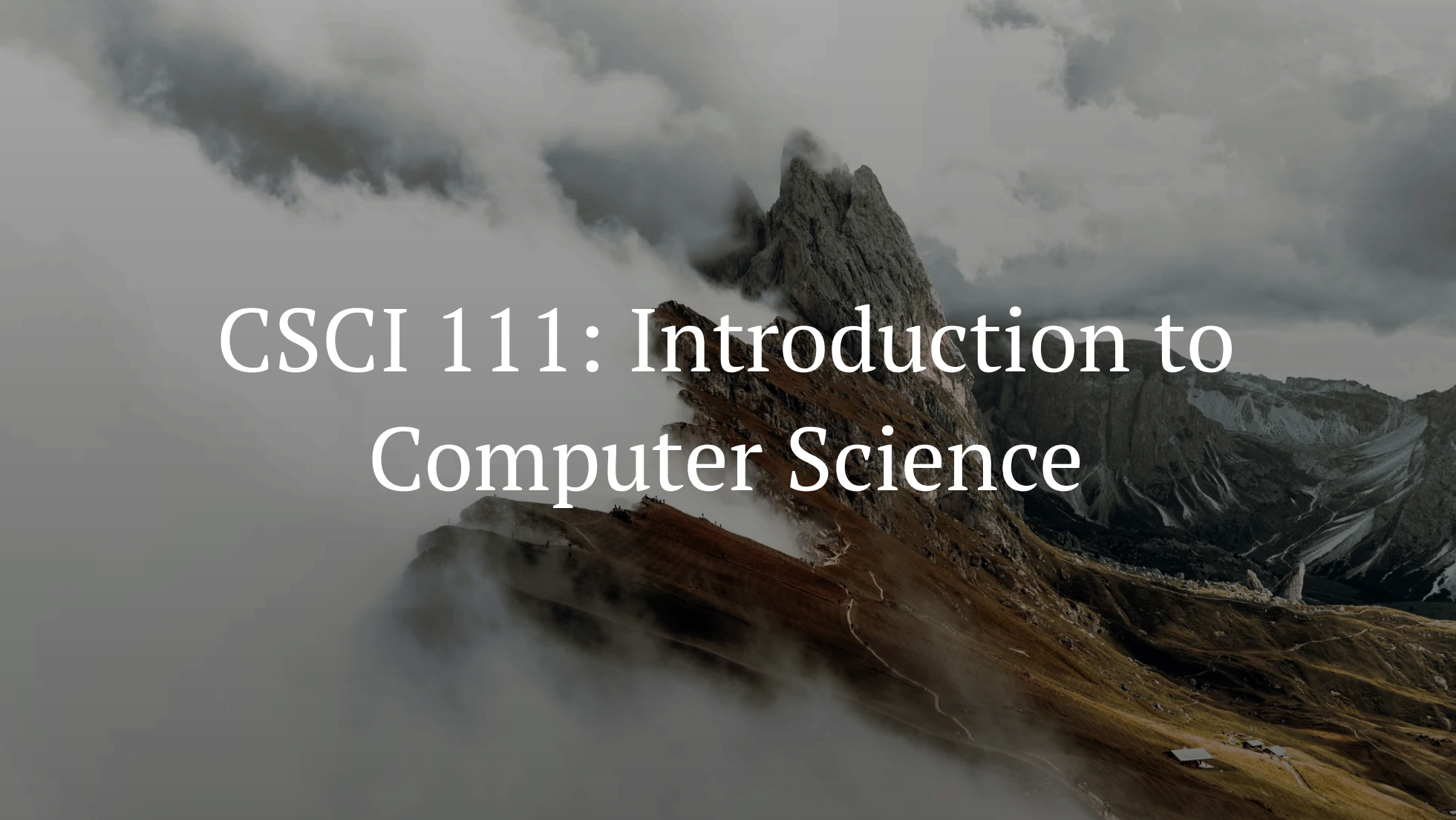

# CSCI 111: Introduction to Computer Science

# Administrivia

# Discord

- **Reminder**: If you haven't already, **please** make sure to join the class Discord server.
  - **Why?**: This is where we'll communicate, share resources, and ask questions outside of class.

# Runestone Academy Setup

- **Create an account** on Runestone Academy to access *How to Think Like a Computer Scientist*.
  - **Course Name**: `tolley_csci111_f24`
  - **Goal**: Start working through the book. We'll cover up to **Chapter 3** this week.

# Reading Assignment Reminder

- **Don't forget** to read *Do Artifacts Have Politics?* and write your short response paragraph for **Wednesday**'s discussion.

# Variables, Expressions, and Statements

*The Foundation of Programming*

- **Variables**: Stores data.
  - Example: `x = 10`
- **Expressions**: Combine values and operators to create new values.
  - Example: `x + 5`
- **Statements**: Do something, like assigning a value.
  - Example: `y = x + 5`

# Basic Arithmetic Operations

**Let's revisit the usual suspects:**

- `+` : Addition (*cake + more cake = happiness*)
- `-` : Subtraction (*paycheck - rent and bill = sadness*)
- `*` : Multiplication (*2 * pizza_slices = more pizza*)
- `/` : Division (*cake / 2 = less cake more sad*)
- `//` **: Integer Division** (*but what if you don't like cake crumbs?*)

# Integer Division vs. Floating Point Division

- **Floating Point Division** ( `/` ): Gives you the exact answer (including decimals).
  - Example: `10 / 3 = 3.3333...`
- **Integer Division** ( `//` ): Only cares about the whole number part.
  - Example: `10 // 3 = 3`
- *Why use integer division?* When you only care about the whole part or are counting objects that can't be split (no half pizzas here).

# When Would You Use Integer Division?

- **Scenarios**:
  - Splitting items evenly among groups.
  - Calculating how many complete sets of something fit.
  - Breaking large problems into smaller chunks, e.g. batch processing.
- **Example**: How many teams of 4 can we form from 13 people?

```
teams = 13 // 4
# teams = 3
```

# Edge Cases with Integer Division

### What if the result isn't a whole number?

- Integer division always rounds down, even if the result isn't a perfect whole number.
  - Example:

```
13 // 4 = 3  # Left with 1 remainder
13 / 4 = 3.25  # Floating-point result
```

- **Negative Numbers**: When negative numbers are involved, it rounds down toward negative infinity.
  - Example:

```
-13 // 4 = -4  # Rounds down to -4
-13 / 4 = -3.25  # Floating-point result
```

# Modulo: The Remainder Specialist

**Modulo ( `%` )**: It's the remainder after division.

- Example

```
10 % 3  # Remainder is 1
```

- Real-World Use Cases: Checking if a number is even or odd:

```
if x % 2 == 0:
    print("Even")
else:
    print("Odd")
```

- Rotating through a set of values (e.g., days of the week).

# Modulo Edge Cases

## What happens with negatives?

- Positive number % negative number:

```
10 % -3 = -2
```

- Negative number % positive number:

```
-10 % 3 = 2
```

- Why? Modulo gives you the "remainder" that would make the division round to the closest lower multiple of the divisor.

# Negative Modulo

**When dealing with negative numbers, things get a bit more nuanced**

## Negative Dividend:

- If the dividend (the number being divided) is negative, Python still returns a positive result as long as the divisor is positive.

**Example:**

```
-10 % 3  # Result: 2
```

**Why?**

- In this case, `-10 ÷ 3` gives a quotient of `-4` (rounding toward negative infinity), and `-4 * 3 = -12`. So, the remainder is `-10 - (-12) = 2`

# Negative Modulo

**When dealing with negative numbers, things get a bit more nuanced**

## Negative Divisor:

- If the divisor is negative, Python returns a result with the same sign as the divisor.

**Example:**

```
10 % -3  # Result: -2
```

**Why?**

- In this case, `10 ÷ -3` gives a quotient of `-4`, and `-4 * -3 = 12`. So, the remainder is `10 - 12 = -2`.

# Negative Modulo

**When dealing with negative numbers, things get a bit more nuanced**

## Both Negative:

- When both the dividend and divisor are negative, the result is negative.

**Example:**

```
-10 % -3  # Result: -1
```

**Why?**

- Here, `-10 ÷ -3` gives a quotient of `3`, and `3 * -3 = -9`. The remainder is `-10 - (-9) = -1`.

# Integer Division + Modulo: A Power Couple

- **Fact**: You can use `//` and `%` to break down any division into "how many times" and "what's left over."
  - Example: `13 // 4 = 3` (three teams) and `13 % 4 = 1` (one person left over).
- **Think of it this way**: You split a group, and the remainder tells you who's hanging out alone.

# Practice Problem: Integer Division and Modulo

*You have 15 cookies, and you're splitting them among 4 friends.*

1. How many cookies does each friend get?
2. How many cookies are left over?

- Write a function:

```
def split_cookies(ARGUMENTS): # What arguments does our function take?
    return RETURNVALUE # What do we return?
```

# Real-World Use Case: Modulo for Days of the Week

- Imagine today is Wednesday (day 3 of the week).
- What day will it be in 10 days? (*Hint: Use modulo!*)
  - `(3 + 10) % 7 = 6` (*Saturday*).
- **Why?** Modulo wraps around the week (or any cycle).
- Useful in calendars, circular queues, game loops.

# Debugging Integer Division & Modulo

- Common mistakes:
  - Forgetting that `//` always rounds down.
  - Misunderstanding how `%` handles negatives.
- **Pro Tip**: Test with edge cases (negative numbers, small values).
  - Example: `-5 % 3 = 1` (*Wait, what?*)

# Closing Thoughts on Integer Division and Modulo

- Integer Division is your friend when you only care about whole numbers.
- Modulo helps you find what's leftover.
- Edge cases are where things get tricky (and interesting).

# Functions

## The Building Blocks of Reusable Code

- Functions let you encapsulate logic into reusable blocks.
- You can define a function once and call it many times.
- Functions can take input (parameters) and return output.
- Example: Writing and calling functions is like giving your code reusable instructions.

# Defining a Simple Function

- Syntax of a function
  - Example:

```
def function_name():
    # Code goes here
    # Return, if applicable, goes here
```

  - A Simple Function:

```
def greet():
    print(Hello, world!")
```

# Calling a Function

- Functions are called by using their name followed by parentheses.
  - Example:

```
greet() # Outputs: Hello, world!
```

# Functions with Parameters

- Parameters allow you to pass values into a function.
- You can provide information to customize the output.
- Example:

```
def greet(name):
    print(f"Hello, {name}!")
```

- Calling the function with a parameter:

```
greet("Alice")  # Outputs: Hello, Alice!
```

# What are Brackets in Functions?

- Brackets ( `()` or `{}` ) have special meaning in Python.
- Round brackets `()` are used to pass arguments or parameters to a function.
- Curly braces `{}` are used inside f-strings for string formatting.

```
name = "Alice"
print(f"Hello, {name}!")  # Inserts 'Alice' into the string
```

# What is an f-string?

- f-strings (formatted strings) allow you to easily insert variables into strings.
- Place an f before the string and put variables inside curly braces `{}` .

```
age = 21
print(f"You are {age} years old.")  # Outputs: You are 21 years old.
```

# Returning a Value from a Function

- Functions can return data using the return keyword.
- Example:

```
def add(a, b):
    return a + b
```

- You can store the returned result in a variable:

```
result = add(5, 3)  # result is 8
```

# Returning Multiple Values

- You can return multiple values from a function by separating them with commas.
- Example:

```
def divide_remainder(a, b):
    quotient = a // b
    remainder = a % b
    return quotient, remainder
```

- Calling the function:

```
q, r = divide_remainder(10, 3)
print(f"Quotient: {q}, Remainder: {r}")  # Outputs: Quotient: 3, Remainder: 1
```

# Advanced: Using the eval() Function

- `eval()` allows you to evaluate Python expressions passed as strings.
- Example:

```
user_input = input("Enter a mathematical expression: ")
result = eval(user_input)
print(f"Result: {result}")
```

- Be cautious with `eval()` as it can execute arbitrary code.

# Practice Problem: Practice Problem: Calculate and Compare

- **Write a function called compare_numbers that:**

  1. Takes two numbers as input.
  2. Returns the sum, difference, and whether the first number is larger than the second.
  3. After calling the function, print the results in a readable format (f-string).

- **Expected Output:**

  - Given input from the user `10` and `7`

```
Sum: 17
Difference: 3
Is A larger: True
```

# Conditionals, Loops, and Flow Control

- Conditionals allow you to execute certain code only when specific conditions are met.
- The most common conditional statement is the `if` statement.

**Example:**

```
if x > 0:
    print("x is positive")
```

# The if-else Structure

- An if-else structure allows you to execute one block of code if a condition is true, and another block if it is false.
- Example:

```
if x > 0:
    print("x is positive")
else:
    print("x is not positive")
```

- The else block runs when the condition in the if statement is false.

# The if-elif-else Structure

- You can chain multiple conditions together using `elif` (else-if) and `else`.
- Example:

```
if x > 0:
    print("x is positive")
elif x == 0:
    print("x is zero")
else:
    print("x is negative")
```

# Comparison Operators

- Conditionals often use comparison operators to evaluate expressions.
- Common operators:
  - `>` : greater than
  - `<` : less than
  - `==` : equal to
  - `!=` : not equal to
  - `>=` : greater than or equal to
  - `<=` : less than or equal to

**Example:**

```
if age >= 18:
    print("You are an adult.")
```

# Logical Operators

- Logical operators let you combine multiple conditions.
  - `and` : True if both conditions are true.
  - `or` : True if at least one condition is true.
  - `not` : Inverts the result of the condition.

**Example:**

```
if age >= 18 and age < 65:
    print("You are an adult but not a senior.")
```

# Loops

- Loops let you repeat a block of code multiple times.
- The most common types of loops are for loops and while loops.

**Example:**

```
# For-loop
for i in range(5):
    print(i)
```

# The `while` Loop

- A while loop continues to run as long as its condition is true.

**Example:**

```
count = 0
while count < 5:
    print(count)
    count += 1
```

- The loop will stop when count reaches 5.

# Loop Control: `break` and `continue`

- You can control loops with break and continue:
  - `break` : Exit the loop entirely.
  - `continue` : Skip the current iteration and move to the next one.

**Example:**

```
for i in range(5):
    if i == 3:
        break
    print(i)  # Output: 0, 1, 2
```

# Combining Loops and Conditionals

- Loops and conditionals work together for more complex logic.

**Example:**

```
for i in range(10):
    if i % 2 == 0:
        print(f"{i} is even")
    else:
        print(f"{i} is odd")
```

# Nested Loops

- Loops can be nested inside other loops to handle multi-dimensional data or complex tasks.

**Example:**

```
for i in range(3):
    for j in range(3):
        print(f"i={i}, j={j}")
```

# Practice Problem: Number Guessing Game

**Problem:**

- Write a simple number guessing game:
  - 1. The program generates a random number between 1 and 20.
  - 2. The user has 5 tries to guess the number.
  - 3. After each guess, tell the user if they were too high, too low, or correct.