CS134:

Conditionals and Modules

Announcements & Logistics

- **Homework 2** is due tonight 10 pm
- **Lab 2** due Wed 10pm / ***Thurs 9 pm (due to power outage)***
    - Will discuss in lab sections Mon/Tues
- Note that you can always work on lab machines any time
- Make sure to keep your work consistent with what is on evolene
- Always push to evolene when done with a work session
- If restarting work on a different machine:
    - If working on a machine on this lab for the 1st time: clone the repository just like you would when starting
    - Otherwise, make sure to git pull first

Do You Have Any Questions?

Last Time

- Wrapped up functions
- Discussed return statements and variable scope
- Start learning about conditionals
    - Boolean data type
    - Making decisions in Python using `if else` statements

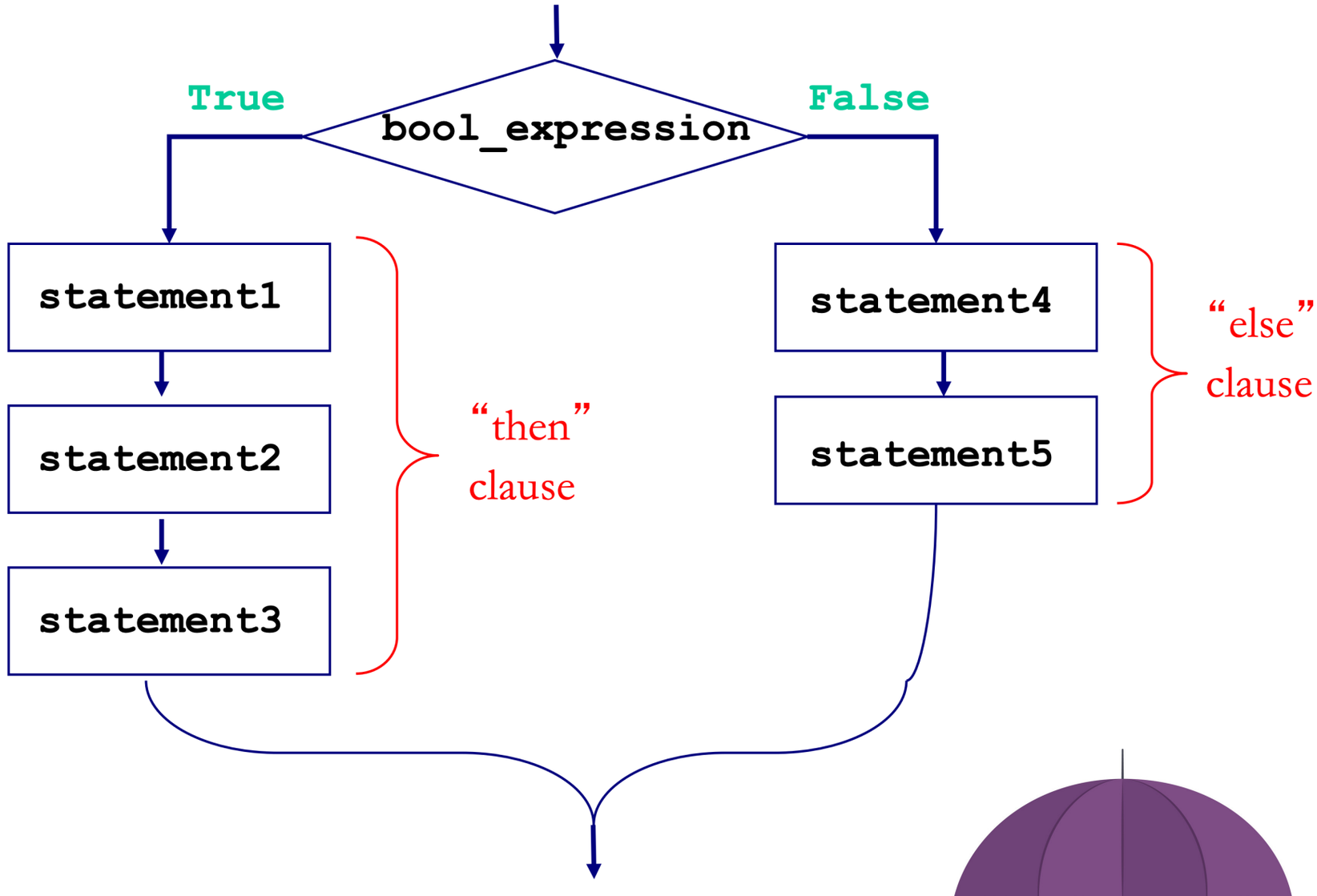
Today's Plan

- Look at more complex decisions in Python
    - Boolean expressions with **and**, **or**, **not**
- Choosing between many different options in our code
    - `If elif else` chained conditionals

Python Conditionals (`if` Statements)

```
if <boolean expression>:
    statement1
    statement2
    statement3
else:
    statement4
    statement5
```

Note: (syntax) Indentation and colon after if and else



If it is raining, then bring an umbrella.
Else, bring your sunglasses.

Image Source: (http://cs111.wellesley.edu/spring19)

Conditional Statements: If Else

- Consider the following functions that check if a number is even or odd

```
def printEven(num):
    """Takes a number as input, prints Even if
    it is even, else prints Odd"""
    if num % 2 == 0: # if even
        print("Even")
    else:
        print("Odd")
```

```
def isEven(num):
    """Takes a number as input, returns True if
    it is even, else returns False"""
    return num % 2 == 0
```

Logical Operators

- Logical operators **and**, **or**, **not** are used to combine Boolean values
- For two expressions exp1 and exp2
  - **not** exp1 (! in other languages) returns the opposite of the truth value for exp1
  - exp1 **and** exp2 (&& in other languages) evaluates to True iff both exp1 and exp2 evaluate to True
  - exp1 **or** exp2 (|| in other languages) evaluates to True iff either exp1 or exp2 evaluate to True

Truth Table for or

exp1	exp2	exp1 or exp2
True	True	**True**
True	False	**True**
False	True	**True**
False	False	**False**

Truth Table for and

exp1	exp2	exp1 and exp2
True	True	**True**
True	False	**False**
False	True	**False**
False	False	**False**

Source: (http://cs111.wellesley.edu/spring19)

Nested Conditionals

- Sometimes, we may encounter a more complicated conditional structure with more than 2 options
- Example: Write a function that takes a temp value in Fahrenheit
    - If temp is above 80, print "It is a hot one out there."
    - If temp is between 60 and 80, print "Nice day out, enjoy!"
    - If temp is below 60, print "Chilly day, don't forget a jacket."
- Notice that temp **can only be in one of those** multiple ranges
    - If we find that temp is greater than 80, no need to check the rest!

Nested Conditionals

```
if booleanExpression1:
    statement 1
    ...
else:
    if booleanExpression2:
        statement 2
        ...
    else:
        statement 3
        ...
```

Attempt 1: Chained Conditionals

- We can nest if-else statements (using indentation to distinguish between matching if-else blocks)
- However, this can quickly become unnecessarily complex (and hard to read)

```
def weather1(temp):
    if temp > 80:
        print("It is a hot one out there.")
    else:
        if temp >= 60:
            print("Nice day out, enjoy!")
        else:
            if temp >= 40:
                print("Chilly day, wear a sweater.")
            else:
                print("Its freezing out, bring a winter jacket!")
```

Attempt 2: Chained Ifs

- What if we used a bunch of if statements (w/o else) one after the other to solve this problem?
- What is the advantage/disadvantage of this approach?

```
def weather2(temp):
    if temp > 80:
        print("It is a hot one out there.")
    if temp >= 60 and temp <= 80:
        print("Nice day out, enjoy!")
    if temp <60 and temp >= 40:
        print("Chilly day, wear a sweater")
    if temp < 40:
        print("Its freezing out, bring a winter jacket!")
```

If Elif Else Statements

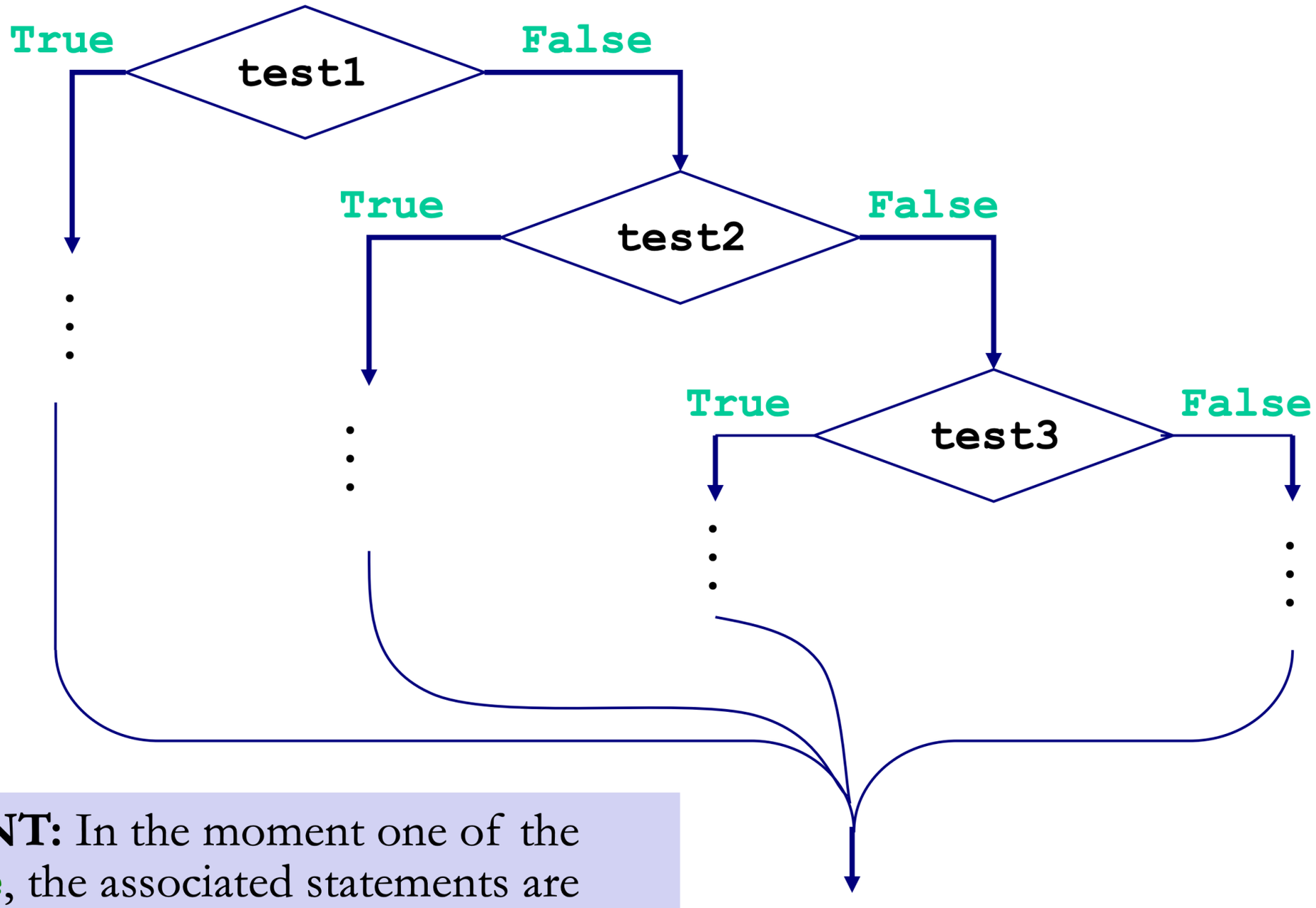
- Fortunately, Python allows us a simpler way to choose one out of many options by **chaining** conditionals

```
if booleanExpression1:
    statement 1
    ...
elif booleanExpression2:
    statement 2
    ...
else:
    statement 3
    ...
```

A better approach that avoids too many indented blocks and improves code readability

Can have any number of `elif` conditions, but only one `else` (at the end)

Flow Diagram: Chained Conditionals



IMPORTANT: In the moment one of the tests is `True`, the associated statements are executed and the chained conditional is exited. Only in the case when tests are False, we continue checking to find a True test.

Image Source: (http://cs111.wellesley.edu/spring19)

Attempt 3: Chained Conditionals

- Note that we can chain together any number of elif blocks
- The else block is still optional

```
def weather3(temp):
    if temp > 80:
        print("It is a hot one out there.")
    elif temp >= 60:
        print("Nice day out, enjoy!")
    elif temp >= 40:
        print("Chilly day, wear a sweater.")
    else:
        print("Its freezing out, bring a winter jacket!")
```

Takeaway of Conditionals

- Chained conditionals can avoid having to nest conditionals. Chaining reduces complexity and improves readability
- Since only one of the branches in a chained `if, elif, else` conditionals evaluates to True, using them avoids unnecessary checks incurred by chaining if statements one after the other

Exercise: `leapYear` Function

- Let us write a function `leapYear` that takes a year as input, and returns `True` if it is a leap year, else returns `False`
- When is a given year a leap year?
    - *"Every year that is exactly divisible by four is a leap year, except for years that are exactly divisible by 100, but these centurial years are leap years, if they are exactly divisible by 400."*

How do we structure this logic using booleans and conditionals?

Exercise: `leapYear` Function

- Let us write a function `leapYear` that takes a year as input, and returns `True` if it is a leap year, else returns `False`
- When is a given year a leap year?
  - *"Every year that is exactly divisible by four is a leap year, except for years that are exactly divisible by 100, but these centurial years are leap years, if they are exactly divisible by 400."*
- If year is not divisible by 4: is not a leap year
- Else (divisible by 4) and if not divisible by 100: is a leap year
- Else (divisible by 4 and by 100) and not divisible by 400: not a leap year
- Else (if we make it to here must be divisible by 400): is a leap year

Exercise: `leapYear` Function

```
def isLeap(year):
    """Takes a year (int) as input and returns
    True if it is a leap year, else returns False"""
    pass
```

Leap years between from 1900 to 2060:

1900	1904	1908	1912	1916	1920	1924	1928	1932	1936
1940	1944	1948	1952	1956	1960	1964	1968	1972	1976
1980	1984	1988	1992	1996	2000	2004	2008	2016	2020
2024	2028	2032	2036	2040	2044	2048	2052	2056	2060

https://www.calendar.best/leap-years.html

Exercise: `leapYear` Function

```
def isLeap(year):
    """Takes a year (int) as input and returns
    True if it is a leap year, else returns False"""

    # if not divisible by 4 return False
    if year % 4 != 0:
        return False

    # is divisible by 4 but not divisible by 100
    elif year % 100 != 0:
        return True

    # is divisible by 4 and divisible by 100
    # but is not divisible by 400
    elif year % 400 != 0:
        return False

    # is divisible by 400 (and also 4, and 100)
    return True
```

Moving on…

Modules and Scripts

- A **script** is generally any piece of code saved in a file, e.g., `leap.py`
- Scripts are meant to be directly executed with: `python3 leap.py`
- A **module** is generally a collection of statements and definitions that are meant to be imported and used by a different program
- Python allows any program we write in a `.py` file to serve both as a module, or script
- To provide a way to distinguish between these two modes, every module has a special variable called `__name__`
- If a variable starts/ends with double __, it's a special python variable

Modules and Scripts

- Consider for example, the code we wrote in `leap.py`
- When `leap.py` file is directly run as a script then the special variable called `__name__` is set to the string `"__main__"`
- When we are importing the code as a module, the `__name__` variable is set to to the name of the module `leap`

- Why does this matter?
    - Importing a module runs the program in it, and we often want different behavior when the code is run as a script vs when it's imported as a module

`if __name__ == '__main__'`

- This is just an if statement with an equality Boolean expression:
  - Checking whether the special variable `__name__` is set to the string `'__main__'`. That is, the code is being run as a script
- We can place code that we want to run when our module is executed as a script inside the `if __name__ == "__main__"`: block
- This is usually testing code and we do not want it to run when we are importing functions from the file

Example: Script vs Module

foo.py

```
1  # foo.py
2  # test the role of __name__ variable
3  print("__name__ is set to", __name__)
```

```
cs134 — Python — 72×11
shikhasingh@williams-194-127 cs134 % python3 foo.py
__name__ is set to __main__
shikhasingh@williams-194-127 cs134 % python3
Python 3.9.7 (default, Sep  3 2021, 12:37:55)
[Clang 12.0.5 (clang-1205.0.22.9)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import foo
__name__ is set to foo
>>>
```