CS134:
Conditionals and Modules
Announcements & Logistics

- **Homework 2** is due tonight 11 pm
- **Lab 2** due Wed 11 pm / Thur 11 pm
- You can work on lab machines any time
- Make sure to keep your work consistent with what is on evolene
  - Best practice: 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
- No class on Friday (Winter Carnival)

Do You Have Any Questions?
Last Time

- Wrapped up functions
- Discussed return statements and variable **scope**
- Started 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

• We are going to cover a lot of material in the next 3 lectures
  • Make sure you are keeping up and getting help if needed!
Python Conditionals (if Statements)

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

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

Note: (syntax) Indentation and colon after if and else
Note: else clause is optional!
Conditional Statements: If Else

- Consider the following functions that check if a number is even or odd
- (More examples in today's notebook)

```python
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 \( \text{exp1} \) and \( \text{exp2} \):
  - **not \( \text{exp1} \) (! in other languages)** returns the opposite of the truth value for \( \text{exp1} \).
  - \( \text{exp1} \) **and** \( \text{exp2} \) (&& in other languages) evaluates to True iff both \( \text{exp1} \) and \( \text{exp2} \) evaluate to True.
  - \( \text{exp1} \) **or** \( \text{exp2} \) (|| in other languages) evaluates to True iff either \( \text{exp1} \) or \( \text{exp2} \) evaluate to True.

### Truth Table for or

<table>
<thead>
<tr>
<th>( \text{exp1} )</th>
<th>( \text{exp2} )</th>
<th>( \text{exp1} ) <strong>or</strong> ( \text{exp2} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

### Truth Table for and

<table>
<thead>
<tr>
<th>( \text{exp1} )</th>
<th>( \text{exp2} )</th>
<th>( \text{exp1} ) <strong>and</strong> ( \text{exp2} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>
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

```python
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)

```python
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 are the advantages/disadvantages of this approach?

```python
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.

```python
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 (optional) `else` (at the end).
Attempt 3: Chained Conditionals

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

```python
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
Exercise: `leapYear` Function

```python
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:

Not a leap year

<table>
<thead>
<tr>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>1900</td>
</tr>
<tr>
<td>1940</td>
</tr>
<tr>
<td>1980</td>
</tr>
<tr>
<td>1988</td>
</tr>
<tr>
<td>2024</td>
</tr>
</tbody>
</table>

Next leap year

<table>
<thead>
<tr>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
</tr>
<tr>
<td>2004</td>
</tr>
<tr>
<td>2008</td>
</tr>
<tr>
<td>2016</td>
</tr>
<tr>
<td>2020</td>
</tr>
<tr>
<td>2040</td>
</tr>
<tr>
<td>2044</td>
</tr>
<tr>
<td>2048</td>
</tr>
<tr>
<td>2052</td>
</tr>
<tr>
<td>2056</td>
</tr>
<tr>
<td>2060</td>
</tr>
</tbody>
</table>
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
  - Modules are used in interactive python when we import functions/code
- Python allows any program we write in a `.py` file to serve both as a module and a script
- To provide a way to distinguish between these two modes of operation, every module has a special variable called `__name__`
- Note: If a variable starts/ends with double `__` in Python, it’s a special 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?
  • 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 in interactive Python
Example: Script vs Module

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

```
bash-3.2$ python3 name.py
__name__ is set to __main__
```

```
bash-3.2$ python3
Python 3.9.7 (v3.9.7:1016ef3790, Aug 30 2021, 16:25:35)
[Clang 12.0.5 (clang-1205.0.22.11)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
[>>> import name
__name__ is set to name
```
Running leap as a Script and Module

- (Jeannie added this slide after lecture!)
Running leap as a Script and Module

- (Jeannie added this slide after lecture!)
- Running leap.py as a script (notice the code in the if block runs!)

```
bash-3.2$ python3 leap.py
Enter a year: 1900
1900 is not a leap year.
bash-3.2$ python3 leap.py
Enter a year: 2040
2040 is a leap year!
```

- Running leap.py as a module in interactive Python

```
bash-3.2$ python3
Python 3.9.7 (v3.9.7:1016ef3790, Aug 30 2021, 16:25:35)
[Clang 12.0.5 (clang-1205.0.22.11)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from leap import *
>>> isLeap(1900)
False
>>> isLeap(2040)
True
>>> exit()
```
Lab 2
Lab 2: Goals

• In this lab, you will be writing a non-trivial Python script to compute the current day of the week in Williamstown.

• High-level learning goals:
  • Defining and calling functions.
  • Using arithmetic operators in Python.
  • Testing your code in interactive Python.
  • Writing conditional (if else) statements to make decisions in your code.
How Computers Keep Track of Time

- On Unix machines time is represented by the **number of seconds**, starting from the beginning of Thursday, January 1, 1970
  - The date is arbitrary, but is called the Unix "epoch"
- In Python we can access this value using the time module()
- The time value is in UTC (current time in England)
- While the value is a float, we only need the integer part for this lab

```bash
$ python3
>>> from time import time
>>> time()
1612800680.9091752
```
Figuring Out the Day of the Week

• The time module gives us the total number of seconds since the Epoch

• Our goal: Use this value to figure out what the current day of the week is in England (for now, later we will deal with timezones)

• Approach (break down the problem):
  • How many minutes have elapsed since the Epoch?
  • How many hours? Days?
  • Suppose the number of days divide evenly by 7. What day of the week is it? What if they do not divide evenly?

• How do we do this using arithmetic operations?
  • Hint: Think about our example involving numCoins from last week
UTCDay(timeval)

• This function takes a floating point number as a parameter, timeval
  • timeval represents the UTC time in England
  • timeval is the total number of seconds since the Epoch
• This function should return:
  • A number between 0-6, which is the day of the week corresponding to timeval
  • Where 0 is Sunday, 1 is Monday, ..., 6 is Saturday
Interactively Testing Functions

• Enter interactive Python by typing `python3` at the Terminal

• Import the function and any modules you need

```python
>>> from day import UTCDay
>>> from time import time
```

• Call your function and see if it returns the desired output

• If you need to make changes to the code in `day.py`:
  • Quit out of interactive python session (Ctrl-D or exit())
  • Restart interactive Python; re-import modules before resuming testing
  • Hint: Can press up on keyboard to see previously typed commands in interactive Python
Running as a Script

• To run a program as a script:
  • type `python3 (filename.py)` in the Terminal
  • This ensures that the code block within `if __name__ == '__main__'` block is executed

```python
if __name__ == "__main__": # run as a script?
    now = time() # UTC time
    dayNumber = localDay(now, -4) # Eastern day of week number
    dayName = dayOfWeek(dayNumber) # get day name
    print("It's "+ dayName +"!") # print it out
```

• Note: The code in this if block is not run in interactive Python