Bonus Slide

List Comprehensions Condense Code

Very often we need to create a new list, but the way that list is created is straightforward. A **list comprehension** lets us condense loops and conditionals that create a list into one line.

```
updated = [ x ** 2 for x in orig if x > 0 ]
```

The first expression is the value being added

The order the loops/conditionals are written shows the order they're executed.

is the same as

```
updated = []
for x in orig:
    if x > 0:
        updated = updated + [ x ** 2 ]
```

Unit 1: Fundamentals of Programming

Testing

15-110 – Friday 09/18

Announcements

- Hw4 due Monday noon

- Don't forget to schedule an oral assessment! New timeslots now available.
  - https://www.cs.cmu.edu/~110/oral/oral1.html

Learning Goals

- Describe why **testing** is an important part of the software development process

- Use **system testing** to verify whether a Python program produces expected results.

- Use **unit testing** to thoroughly test a single function and verify that it works as intended.

Checking Our Goals

When we use a Python program, we usually have a **goal** in mind for what we want to accomplish.

Our goals may not align perfectly with the code we interact with. Algorithms may vary from our intentions, or bugs may cause unexpected behavior.

How can we ensure that our programs accomplish our goals correctly?

Case Study: Transcript Analysis

Case Study Description

You are a toy designer. You are currently testing variations on a new action figure toy. Figure #1 has detachable limbs that can be rearranged; Figure #2 has buttons that make it light up and make noises; Figure #3 can move its limbs normally.

You just concluded a set of product tests with children and want to analyze the transcripts of what they said while playing. Your main goal is to measure the **affect** of the children to determine which variation produces the most positive results.

Transcript Example

You've already collected the transcripts into three lists. Here's one:

```
toy1_transcripts = [
    "Whoa this is so cool, I can put the arm on backwards, that's awesome!",
    "Ugh, the leg won't click back in, this is so annoying and frustrating.",
    "Look, I made a monster with four arms, this is amazing, I love it!",
    "I don't get it, why would I want to take the arms off.",
    "Yes! I switched all the parts around, this is the best toy ever, so much fun!"
 ]
```

Codebase

And you've acquired a codebase that (supposedly) can analyze the transcripts and find which set has the most positive affect:

```
def clean_word(word):
    """
    Removes common punctuation from the edges of a word so that
    "cool," or "awesome!" can still be matched against our word lists.
    """
    punctuation = ".,!?;:\"'"
    return word.strip(punctuation)


def get_transcript_score(transcript):
    """
    Computes an affect score for a single transcript (a string of text).
    Adds 1 for each positive word found, subtracts 1 for each negative word found.
    """
    score = 0
    words = transcript.split()

    for word in words:
        cleaned = clean_word(word)

        if cleaned in positive_words:
            score = score + 1
```

You Do: Does this fulfill the goal?

You do: what can we do to prove to ourselves that this code actually fulfills our goal? How can we determine if it always works correctly?

Reminder: Your main goal is to measure the **affect** of the children to determine which variation produces the most positive results.

Importance of Testing

Testing Checks Code

Testing is the process of checking a set of code to ensure it meets expectations and works correctly.

Testing is a crucial part of the software development process. Well-designed tests ensure that a set of code works correctly now and in the future when changes are made.

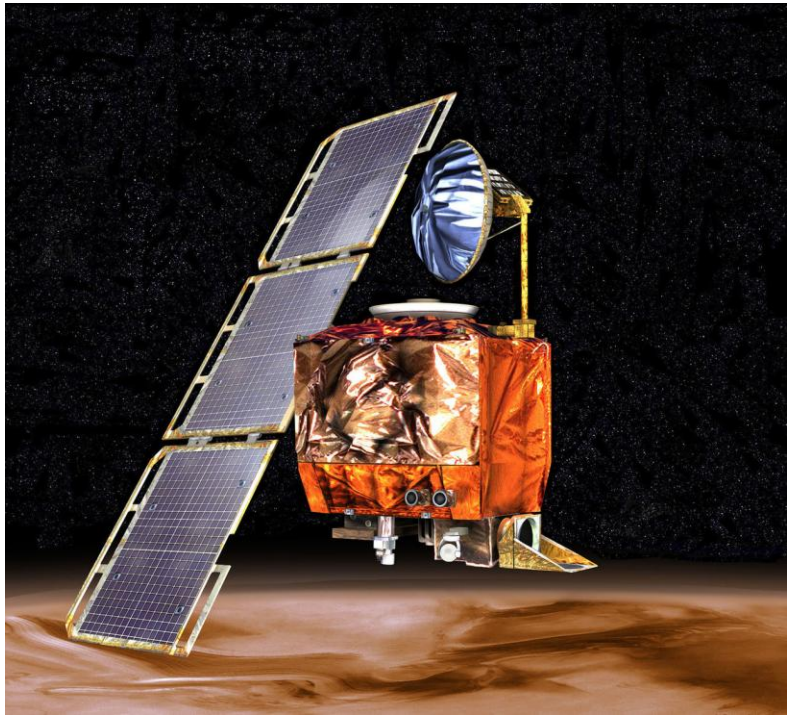
When code is not adequately tested, there can be major repercussions…

Mars Climate Orbiter

In 1998 the Mars Climate Orbiter was launched by NASA, with a goal of orbiting Mars to gather data and relay communications.

When the spacecraft attempted to enter orbit almost a year later, it used the wrong trajectory and burned up in Mars' atmosphere.

Why did this happen? Two software systems used by the spacecraft (one developed by NASA, the other by Lockheed Martin) used **different unit systems –** NASA used metric, Lockheed Martin used imperial.



https://science.nasa.gov/mission/mars-climate-orbiter/

Therac-25

The Therac-25 was designed in 1982 to provide radiation therapy to patients.

Between 1985-1987, at least six incidents occurred where the machine gave a massive overdose of radiation to a patient, *not* what was entered by the technician. Several patients died.

Previous versions of the machine had included hardware-level safety checks to ensure radiation overdoses would not occur. This version relied on software-level checks, but these checks had not been thoroughly tested.

https://ieeexplore.ieee.org/document/274940

British Post Office Scandal

The British Post Office introduced a new accounting system in 1999, Horizon.

Horizon had many design problems and bugs that caused faulty behavior and showed false imbalances in accounts. Subpostmasters who used the system reported these errors but were often blamed for the discrepancies.

More than 900 subpostmasters were wrongfully convicted of stealing money. This led to people serving time in prison, individual ruin, and multiple suicides.

https://cacm.acm.org/news/how-software-bugs-led-to-one-of-the-greatest-miscarriages-of-justice-in-british-history/

Regulation of Software Testing

Due to these events and more, some industries have implemented **regulations** requiring design and testing requirements for certain types of software.

For example, aviation software used in several countries (including the United States) must meet the DO-178C standard.

Regulations evolve over time as the industry standards for programming change.

https://www.do178.org/

System Testing

Test the Whole Thing

In an earlier lecture we introduced the concept of a **test case**. Run a function on a specific input and see if it produces the expected output.

We can use this same approach on an entire system. After considering the goal, identify an input for the system where **you know what the output should be**. Run the system on that input to see if it produces the expected result.

It often helps to start with a simple/small example, then build up complexity.

Case Study – A Simpler Input

What would be the simplest input for our case study?

Simplify to just two toys, each with just one transcript. One transcript is `"Good"`; the other `"Bad"`. We would expect the first toy to be ranked first.

Let's try it!

Unsure? Read the Code.

There may be times when you aren't sure why a result turned out a certain way.

Read the code to understand how it works. Comments are helpful, but verify them against the actual code. If something doesn't make sense, try running it with print statements to see what it's doing.

It's especially important to read and understand all the code in any circumstance where correctness is extra important, like if you're creating analysis code for a research study or an application that patients will interact with.

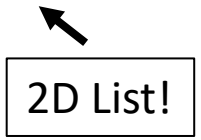
Example: Read `main`

Let's read through the `main` function to see what the code does at a high level.

```
def main():
    toy_names = ["Toy 1 (Detachable Parts)", "Toy 2 (Lights and Sounds)", "Toy 3
(Basic Figure)"]
    toy_transcript_lists = [toy1_transcripts, toy2_transcripts, toy3_transcripts]

    toy_scores = []
    for i in range(len(toy_names)):
        print_transcript_scores(toy_names[i], toy_transcript_lists[i])
        toy_scores = toy_scores + [get_average_score(toy_transcript_lists[i])]

    winner = find_most_positive_toy(toy_names, toy_scores)
    print("The toy with the most positive affect is: " + winner)
```



You Do: Read the Code!

You do: carefully read and comprehend one of the functions in the starter file. What is its goal? Does it accomplish that goal? Does anything about how it works surprise or concern you?

If your function calls another function, check the comment on the helper function to see what it does.

If you have time, try reading the other functions as well.

Redesign When Needed

While reading code, you might notice parts of the algorithm that you think could be done better.

Take initiative and improve the codebase when possible! Just make sure to document the changes and test them to make sure the code still works.

Sidebar: End-to-End Testing

Software engineers use **end-to-end testing** to check if an entire system works as expected when run from beginning to end.

This can get very complicated when a system has multiple parts that are all communicating with each other! But it's also a great way to ensure that each part works *in the context of the others*, not just on its own.

Unit Testing

Unit Testing Tests Functions

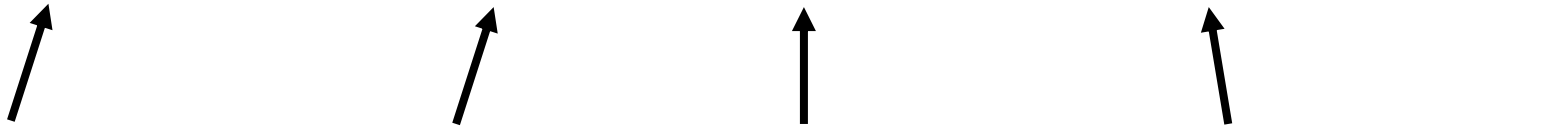
Some functions may be particularly sensitive and particularly important to get correct. Doing a few sample runs with the whole system might not be enough to identify problems.

We can use **unit testing** to verify that a small part of the codebase – like a single function – works exactly as expected.

Reminder: `assert` Statements

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, "avg(20, 4) -> 5"
```



Test Cases Work Together

When designing tests for a single unit, it's important to get good **coverage** of the unit. One test is not enough!

Write multiple tests. The tests should cover varied inputs, varied outputs, and test whether the unit works at scale.

```
assert findAverage(20, 4) == 5, "basic case"
assert findAverage(100, 10) == 10, "basic case"
assert findAverage(15, 2) == 7.5, "floating point result"
assert findAverage(20000, 800) == 25, "larger input"
```

Edge Cases Test Boundaries

It's also important to consider the **edge cases** of a unit. Check either side of a decision point to make sure the right choice is made each time, and check unusual inputs to make sure the function behaves appropriately.

```
assert findAverage(52, 1) == 52, "divide by 1"
assert findAverage(10, 0) == "n/a", "divide by 0"
assert findAverage(10.8, 2) == "5.4", "floats"
```

Case Study: Testing `find_most_positive_toy`

How can we test `find_most_positive_toy` to ensure it works correctly?

Simple input and output: `["A", "B"], [1, -1] -> "A"`

More complex: `["W", "X", "Y", "Z"], [0.3, 0.1, 0.7, -0.2] -> "Y"`

Last is best: `["A", "B", "C"], [0.1, 0.2, 0.3] -> "C"`

Just one: `["X"], [-0.4] -> "X"`

Should we test empty lists? Or is it okay for the function to not support that?

You do: test `get_average_score`

You do: consider the function `get_average_score` from the codebase. Write a test set to thoroughly test this function and ensure it works correctly.

Note: you'll want to call `get_transcript_score` on your test inputs to see what they score. If that function changes, the tests need to change too.

Learning Goals

- Describe why **testing** is an important part of the software development process

- Use **system testing** to verify whether a Python program produces expected results.

- Use **unit testing** to thoroughly test a single function and verify that it works as intended.