Errors

In particular NotImplementedError (Not Implemented Yet vs. Not Implemented Ever)
Either usage is valid, but a given program should be consistent

Errors can be raised manually in code

try/except blocks

Can specify blocks to be executed if a certain error is raised during execution

Before: Error leads to program crash

Now:

Error can be handled more quietly (e.g. If an input file is corrupted, log the error and keep going instead of crashing)

Recursion (examples in recursion.py)

Recursion is a tricky topic. You don't have to totally "get" it now, but seeing it now will help you later

Recursion is useful when you can solve a problem by breaking it into smaller versions of the same problem, and then combining the solutions

Recursion is generally implemented as a function that calls itself

(show first example, factorial)

Explanation: We know n! is n*(n-1)!, and that 0! is 1. We keep expressing it in terms of smaller and smaller factorials until we reach 0! because we know the answer to that

To use recursion, we break things into a **base case** and a **recursive case**

base case - we need to know how to solve the "smallest version" of a problem. For example, we know that 0! is 1. Common base cases include the numbers 0 and 1, an empty string, or an empty list

recursive case - given a problem, we need to know how to express it in terms of "smaller versions" of the same problem. "smaller versions" of a problem are ones that progress towards the base case. For example, n! = n*(n-1)!

Since the recursive case allows us to express any problem in terms of “smaller versions” of the same problem and the base case gives us the answer to the “smallest version” of the problem, this implies that we can solve a recursive problem by applying the recursive case repeatedly until we reach the smallest version of the problem

Go over remaining examples

<u>Thinking Like a Computer Scientist</u>

Not Reinventing The Wheel

It’s not just impossible to solve every problem at once, it’s bad form

If someone else has already solved a problem well, you should use their solution if possible

Most solutions are the result of integrating others’ solutions to sub-problems in order to solve another problem for a different group of people

The Essence of Problem Solving

1. Problem Definition - Think about what problem needs to be solved. Does it already have a solution? Are there limitations in existing solutions that need to be addressed? Problems don’t only relate to the ability to do something, they can also relate to the efficiency, cost, security, or ease at which something can be done. Who does the problem pertain to, who will use the solution, and what would they want out of a solution?
2. Solution Specification - What properties should the solution to this problem have? What parameters must it satisfy? What interface will a user of the solution use to interact with it? Do NOT describe how it works at this phase, only the black box properties it should satisfy. What requirements are essential to the solution being worthwhile? What requirements can be traded off if necessary to meet the essential requirements?
3. Solution Implementation - Break the solution into smaller pieces until the pieces can actually be built. Think about what tools you have at your disposal to implement the solution. Implement each piece to its requirements so that the overall requirements of the solution can be met.

Problems with the above methodology:

- We rarely know what properties our solution should have ahead of time
- Users don’t know what they want, especially ahead of time
- Requirements are usually poorly defined early on
- Requirements often need to be changed as tradeoffs are made in implementation

The above approach is a great way to approach a particular well defined problem, but most problems are not well defined. We deal with this by applying the methodology repeatedly

Iterative Development

Before: Waterfall methodology - clean path from requirements analysis to finished product

But:

- The path is rarely clean
- Requirements change
- The internet made it fast and cheap to update products and keep developing them after release

Now: Iteration (follow the above process, but repeatedly)

- Get the important features out now, instead of waiting until we have all the features
- Get feedback and refine requirements based on it, instead of trying to gather all requirements the first time
- Start getting users sooner, instead of waiting for a totally finished product first
- Build the solution incrementally, instead of trying to get it right all at once

The Iteration Loop

1. Design - The solution is designed using the current requirements as a guide, producing a specification for the solution
2. Implement - The solution is implemented using the current specification as a guide, producing a finished (but not necessarily perfect) version of the solution
3. Evaluate - The solution is evaluated by the eventual end-users of the product. Their evaluations are used to guide the creation of new requirements for a better solution

This loop is repeated, using the new requirements to guide another Design

<u>Computer Science is Everywhere</u>

Because computers are everywhere now

- The Cloud: Enabling small businesses and startups by renting them server space
- User Interfaces: Developing a user interface to solve a problem for some group of people
- Video Games: Literally software for entertainment purposes
- Databases: Helping businesses manage their data
- Device Drivers: Making devices actually work with your computer

- Programmer Tools: Making tools for other programmers to help them make better products
- Web Development: Pretty much any business needs a website
- Computational Biology: Modelling biological systems in computer programs and simulating them to help develop products
- Modelling Anything: Beyond biology, industries such as physics, chemistry, and industrial design need things simulated