

CS 200 Final Exam Review

Fall 2024

Agenda

- Finals Logistics
- Unix, including bash shell scripts
- Python review; Recursion, List Comprehensions, Decorators, Memoization, & Exceptions
- Regular Expressions, Expressions & Procedures
- Data Structures; Stacks, Queues, Hash Tables and Heaps
- Object Oriented Programming
- SQL
- Python Byte Code & Machine Architecture
- Cryptography
- Final Tips
- Practice Problems

Agenda

- Finals Logistics
- Unix, including bash shell scripts
- Python review; Recursion, List Comprehensions, Decorators, Memoization, & Exceptions
- Regular Expressions, Expressions & Procedures
- Data Structures; Stacks, Queues, Hash Tables and Heaps
- Object Oriented Programming
- SQL
- Python Byte Code & Machine Architecture
- Cryptography
- Final Tips
- Practice Problems

Logistics for Final Exam

- Wednesday, December 18th at 9:00 am
- Watson, Room A51 (across from Pauli Murray)
- One 8.5 x 11 page of notes is allowed
 - Must hand in cheat-sheet with your exam
 - Must bring your Yale ID


***For special accommodations, please reach out to
Professor Slade ASAP***

Available Resources (same as before)

- The Basics → [CPSC 200 – Course Site](#); This include the lecture notes
- Google's Python Class Intro → [Overview](#)
- ["THE" Python Guide](#)
- Practice material for Midterm 2
 - Practice [Exam](#) / [Solutions](#)
 - UNIX Principles (1 though 6)
 - ssh into the Zoo; then in your home folder, type the following command:
 - `python3 /c/cs201/www/unixtutorial.py`
- Ed Discussion board → Ask questions and call out typos (great way to earn Errata points)

Agenda

- Finals Logistics
- Unix, including bash shell scripts
- Python review; Recursion, List Comprehensions, Decorators, Memoization, & Exceptions
- Regular Expressions, Expressions & Procedures
- Data Structures; Stacks, Queues, Hash Tables and Heaps
- Object Oriented Programming
- SQL
- Python Byte Code & Machine Architecture
- Cryptography
- Final Tips
- Practice Problems

A cartoon illustration of Mr. Garrison from South Park, wearing his signature glasses and a blue suit with a red tie. He is looking at a computer monitor. The monitor displays green text on a black background, which appears to be a terminal window. The text reads: 'INITIALIZE MODEM', 'USRobotics, Sportster 56K X2, AT&F1&C1&D2510=20515=128527=64', and a small green cursor bar on the line below. The background of the scene is a dimly lit office with a desk and some papers visible.

```
INITIALIZE MODEM
USRobotics, Sportster 56K X2, AT&F1&C1&D2510=20515=128527=64
█
```

UNIX Scenarios

UNIX will cover through Principle Six

- 1) UNIX tutorial on the Zoo → ssh into the Zoo; then in your home folder, type the following command:

```
python3 /c/cs201/www/unixtutorial.py
```

- 2) Another resource for additional practice is Tutorial Point's Unix/Linux Tutorial

<https://www.tutorialspoint.com/unix/index.htm>

General tips:

- Take the time to work on the principles. Even if you completed the first two, go back and practice.

UNIX Example Transcript 1

```
$ ls
data.txt script.sh test.py logs backup

$ XXXX

$ cd projects

$ pwd
/home/user/projects

$ XXXX
data.txt script.sh test.py logs backup projects

$ file script.sh
script.sh: Bourne-Again shell script, ASCII text executable

$ echo "Sample Log Entry" > log.txt

$ XXXX
total 4
-rw-r--r-- 1 user user 18 Dec  7 12:00 log.txt

$ chmod +x log.txt

$ ./log.txt
Sample Log Entry

$ XXXX
```

```
[1] 12345

$ ps
  PID TTY          TIME CMD
 12343 pts/1    00:00:00 bash
 12345 pts/1    00:00:00 sleep
 12346 pts/1    00:00:00 ps

$ jobs
[1]+  Running                  sleep 500 &

$ XXXX

$ ps
  PID TTY          TIME CMD
 12343 pts/1    00:00:00 bash
 12346 pts/1    00:00:00 ps
[1]+  Terminated              sleep 500

$ XXXX

$ echo $NOW
Thu Dec  7 12:01:30 PM EST 2024

$ XXXX

$ myuser
user
```

UNIX Example Transcript 1

```
$ ls
data.txt script.sh test.py logs backup

$ mkdir projects

$ cd projects

$ pwd
/home/user/projects

$ ls ..
data.txt script.sh test.py logs backup projects

$ file script.sh
script.sh: Bourne-Again shell script, ASCII text executable

$ echo "Sample Log Entry" > log.txt

$ ls -l
total 4
-rw-r--r-- 1 user user 18 Dec  7 12:00 log.txt

$ chmod +x log.txt

$ ./log.txt
Sample Log Entry

$ sleep 500 &
```

```
[1] 12345

$ ps
  PID TTY          TIME CMD
 12343 pts/1    00:00:00 bash
 12345 pts/1    00:00:00 sleep
 12346 pts/1    00:00:00 ps

$ jobs
[1]+  Running                  sleep 500 &

$ kill %1

$ ps
  PID TTY          TIME CMD
 12343 pts/1    00:00:00 bash
 12346 pts/1    00:00:00 ps
[1]+  Terminated              sleep 500

$ NOW=$(date)

$ echo $NOW
Thu Dec  7 12:01:30 PM EST 2024

$ alias myuser="whoami"

$ myuser
user
```

UNIX Example Transcript 2

```
$ pwd  
/home/user
```

```
$ mkdir project_dir
```

```
$ XXXX
```

```
$ XXXX  
/home/user/project_dir
```

```
$ touch file1.txt file2.txt file3.log
```

```
$ ls  
file1.txt file2.txt file3.log
```

```
$ echo "apple" > file1.txt
```

```
$ echo "banana" >> file1.txt
```

```
$ echo "carrot" > file2.txt
```

```
$ XXXX  
2
```

```
$ grep "apple" file1.txt  
apple
```

```
$ XXXX
```

```
$ cp file1.txt file1_copy.txt
```

```
$ sort file1.txt > sorted_file.txt
```

```
$ XXXX  
$ diff file1.txt sorted_file.txt  
1c1  
< apple  
---  
> banana
```

```
$ ls  
file1.txt file1_copy.txt file2.txt file3_backup.log sorted_file.txt
```

```
$ rm file1_copy.txt
```

```
$ XXXX  
1 pwd  
2 mkdir project_dir  
3 cd project_dir  
4 touch file1.txt file2.txt file3.log  
5 ls
```

UNIX Example Transcript 2

```
$ pwd  
/home/user
```

```
$ mkdir project_dir
```

```
$ cd project_dir
```

```
$ pwd  
/home/user/project_dir
```

```
$ touch file1.txt file2.txt file3.log
```

```
$ ls  
file1.txt file2.txt file3.log
```

```
$ echo "apple" > file1.txt
```

```
$ echo "banana" >> file1.txt
```

```
$ echo "carrot" > file2.txt
```

```
$ cat file1.txt | wc -w  
2
```

```
$ grep "apple" file1.txt  
apple
```

```
$ mv file3.log file3_backup.log
```

```
$ cp file1.txt file1_copy.txt
```

```
$ sort file1.txt > sorted_file.txt
```

```
$ diff file1.txt file1_copy.txt
```

```
$ diff file1.txt sorted_file.txt
```

```
1c1
```

```
< apple
```

```
---
```

```
> banana
```

```
$ ls  
file1.txt file1_copy.txt file2.txt file3_backup.log sorted_file.txt
```

```
$ rm file1_copy.txt
```

```
$ history | head -n 5
```

```
1 pwd
```

```
2 mkdir project_dir
```

```
3 cd project_dir
```

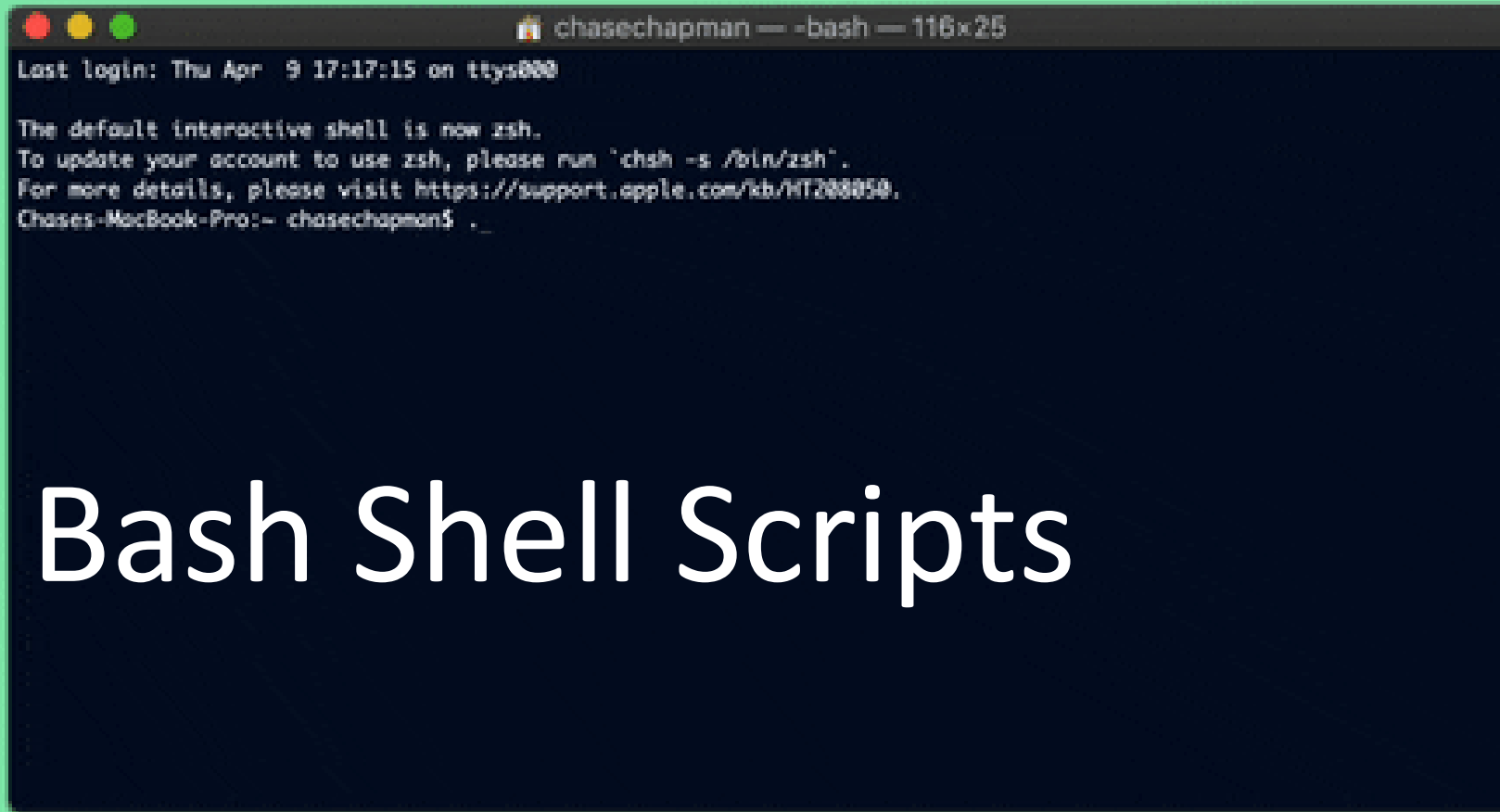
```
4 touch file1.txt file2.txt file3.log
```

```
5 ls
```

Agenda

- Finals Logistics
- Unix, including bash shell scripts
- Python review; Recursion, List Comprehensions, Decorators, Memoization, & Exceptions
- Regular Expressions, Expressions & Procedures
- Data Structures; Stacks, Queues, Hash Tables and Heaps
- Object Oriented Programming
- SQL
- Python Byte Code & Machine Architecture
- Cryptography
- Final Tips
- Practice Problems

UNIX, Bash Shell Scripts

A terminal window with a dark blue background and white text. The window title bar shows a red, yellow, and green window control icon, followed by the text 'chasechapman — -bash — 116x25'. The terminal content shows a login message, a notification about the default shell changing to zsh, instructions on how to update the account, a URL for more details, and a command prompt.

```
chasechapman — -bash — 116x25
Last login: Thu Apr  9 17:17:15 on ttys000

The default interactive shell is now zsh.
To update your account to use zsh, please run 'chsh -s /bin/zsh'.
For more details, please visit https://support.apple.com/kb/HT208050.
Chases-MacBook-Pro:~ chasechapman$ .
```

UNIX, Bash Shell Scripts

What is a Bash Shell Script?

- A text file containing a series of commands executed by the Bash shell
- Automates repetitive tasks and simplifies complex operations
- Commonly used in system administration, file processing, and software development

Key Features of Bash Scripts:

- **Variables:** Store data for reuse (e.g., `my_var="hello"`)
- **Control Structures:** Include if, for, while loops for decision-making and iteration
- **Command-Line Arguments:** Access user inputs with `$1`, `$2`, `$*`, etc
- **Redirection and Pipes:** Redirect output/input (`>`, `<`) and pass data between commands (`|`)

Bash Scripts Problem - Example

Understand the Requirements:

- Identify what the script needs to accomplish (e.g., process input, iterate over files)

Set Up the Environment:

- Start with the shebang (`#!/bin/bash`) to specify the shell interpreter
- Use comments to document the purpose of the script

Handle Input:

- Check for command-line arguments using `$1`, `$2`, etc
- Validate input (e.g., using `[-z "$1"]` for missing arguments)

Develop Logic:

- Use loops (`for`, `while`) to process data
- Apply conditional statements (`if`, `elif`, `else`) for decision-making

Debug and Test:

- Use `echo` to print variable values for debugging
- Test the script with various input cases to ensure accuracy

Steps to Solve Bash Script Problems

Below is a shell script z; what is the output of the following commands? Describe this script.

```
#!/bin/bash

# Check if at least one argument is provided
if [ -z "$1" ]; then
    echo "usage: $0 positive integers"
    exit
fi

sum=0

# Iterate through all arguments
for num in "$@"; do
    # Validate if the argument is a positive integer
    if ! [[ "$num" =~ ^[0-9]+$ ]]; then
        echo "error: input must be positive integers"
        exit
    fi
    # Add the valid number to the sum
    sum=$((sum + num))
done

# Output the sum
echo $sum
```

```
$ ./z
[REDACTED]

# Valid input:
$ ./z 1 2 3 4
[REDACTED]

# Invalid input (negative numbers):
$ ./z -1 2 3
[REDACTED]

# Invalid input (non-integer values):
$ ./z 2 a 3
[REDACTED]
```

Behavior: ?

Steps to Solve Bash Script Problems

Below is a shell script z; what is the output of the following commands? Describe this script.

```
#!/bin/bash

# Check if at least one argument is provided
if [ -z "$1" ]; then
    echo "usage: $0 positive integers"
    exit
fi

sum=0

# Iterate through all arguments
for num in "$@"; do
    # Validate if the argument is a positive integer
    if ! [[ "$num" =~ ^[0-9]+$ ]]; then
        echo "error: input must be positive integers"
        exit
    fi
    # Add the valid number to the sum
    sum=$((sum + num))
done

# Output the sum
echo $sum
```

```
$ ./z
usage: ./z positive integers

# Valid input:
$ ./z 1 2 3 4
10

# Invalid input (negative numbers):
$ ./z -1 2 3
error: input must be positive integers

# Invalid input (non-integer values):
$ ./z 2 a 3
error: input must be positive integers
```

Behavior: The script calculates and prints the sum of all positive integer arguments passed to it

If no arguments are provided, it prints a usage message

Non-positive integers are ignored in the calculation.

Agenda

- Finals Logistics
- Unix, including bash shell scripts
- Python review; Recursion, List Comprehensions, Decorators, Memoization, & Exceptions
- Regular Expressions, Expressions & Procedures
- Data Structures; Stacks, Queues, Hash Tables and Heaps
- Object Oriented Programming
- SQL
- Python Byte Code & Machine Architecture
- Cryptography
- Final Tips

Python, Recursion



Python, Recursion

What is Recursion?

- A method where a function calls itself to solve a problem
- Used to break a problem into smaller, manageable parts

Key Components:

- **Base Case:** Defines the stopping condition for recursion. Without it, recursion leads to infinite loops
- **Recursive Case:** The logic that breaks the problem into smaller subproblems and calls the function on them

Benefits of Recursion:

- Simplifies solutions for problems like tree traversal, mathematical sequences, and navigating nested structures
- Often more elegant than iterative solutions (e.g., using loops)

Python, Recursion

Why Use Recursion?

- Breaks down complex problems into smaller, repeatable tasks
- Useful for working with hierarchical or tree-like structures

Examples of Recursive Problems:

- **Factorial Calculation:**
 $n! = n \times (n-1) \times (n-2) \times \dots \times 1$
- **Fibonacci Sequence:**
 $F(n) = F(n-1) + F(n-2)$
- **Tree Traversal:** Navigating nodes in hierarchical data

Key Considerations:

- Ensure the base case is defined to avoid stack overflow
- Monitor memory usage as recursive calls can consume significant stack space

Factorial Calculation:

$$n! = n \times (n - 1) \times (n - 2) \times \dots \times 1$$

python

```
def factorial(n):  
    if n == 1:  
        return 1  
    return n * factorial(n - 1)
```

Fibonacci Sequence:

$$F(n) = F(n - 1) + F(n - 2)$$

python

```
def fibonacci(n):  
    if n <= 1:  
        return n  
    return fibonacci(n - 1) + fibonacci(n - 2)
```

Agenda

- Finals Logistics
- Unix, including bash shell scripts
- Python review; Recursion, List Comprehensions, Decorators, Memoization, & Exceptions
- Regular Expressions, Expressions & Procedures
- Data Structures; Stacks, Queues, Hash Tables and Heaps
- Object Oriented Programming
- SQL
- Python Byte Code & Machine Architecture
- Cryptography
- Final Tips
- Practice Problems

Python, List Comprehensions

Python, List Comprehensions

What are List Comprehensions?

- A concise way to create lists in Python
- Syntax: [expression for item in iterable if condition]
- Combines loops and conditional logic into a single line

Key Benefits:

- More readable and compact compared to traditional for-loops
- Efficient and faster execution for list creation tasks

Key Considerations:

- Use for simple, readable tasks
- Avoid overly complex logic for maintainability

Using Conditions:

- Add conditions with `if` to filter items.

python

```
even_numbers = [x for x in range(10) if x % 2 == 0]  
# Output: [0, 2, 4, 6, 8]
```

Nested List Comprehensions:

- Handle complex iterations, like matrix transformations.

python

```
matrix = [[1, 2], [3, 4]]  
flat = [num for row in matrix for num in row]  
# Output: [1, 2, 3, 4]
```

With Functions:

- Apply functions to each item.

python

```
words = ["hello", "world"]  
upper_words = [word.upper() for word in words]  
# Output: ['HELLO', 'WORLD']
```

Agenda

- Finals Logistics
- Unix, including bash shell scripts
- Python review; Recursion, List Comprehensions, **Decorators**, Memoization, & Exceptions
- Regular Expressions, Expressions & Procedures
- Data Structures; Stacks, Queues, Hash Tables and Heaps
- Object Oriented Programming
- SQL
- Python Byte Code & Machine Architecture
- Cryptography
- Final Tips

Python, Decorators

Decorators

Definition: Decorators are functions in Python that modify the behavior of another function or method

Purpose: They add functionality to existing functions without altering their code

How Do Decorators Work?

Wrapper Function: A decorator wraps another function and executes additional code before or after the original function

@ Syntax: Decorators use the `@decorator_name` syntax above the function definition

Key Benefits

Reusability: Write functionality once and apply it to multiple functions

Readability: Makes code cleaner and easier to understand

Dynamic Behavior: Add behavior to functions during runtime

Decorators – How to approach?

Understand the Problem Requirements:

- Identify what behavior the decorator should add (e.g., logging, argument validation, caching)
- Check if the decorator modifies the function's behavior, its return value, or both

Analyze Input and Output:

- Understand the inputs the decorated function will take
- Know what the decorator is expected to do with those inputs or outputs

Design the Wrapper Function:

- Write a nested function (wrapper) inside the decorator that wraps the original function
- Use `*args` and `**kwargs` to handle a flexible number of arguments

Use `functools.wraps` to Preserve Metadata:

- Always decorate the wrapper function with `@wraps(func)` to retain the original function's metadata like `__name__` and `__doc__`

Test Incrementally:

- First, ensure the function works without the decorator
- Apply the decorator and confirm that the desired behavior is added without breaking the original function

Decorators - Example

Write a Python decorator function
timing(func)
that times how long a function takes to execute.

If the debug parameter is set to True, it should
print the execution time and the function's
name.

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

@timing
def factorial(n, debug=True):
    if n == 0:
        return 1
    return n * factorial(n - 1, debug=debug)

>>> add(5, 10)
15

>>> add(5, 10, debug=True)
Function 'add' executed in 0.00001 seconds.
15

>>> factorial(5)
120

>>> factorial(5, debug=True)
Function 'factorial' executed in 0.0001 seconds.
120
```

Decorators - Explanation

- The timing decorator uses `@wraps(func)` to preserve the original function's metadata (e.g., `__name__` and `__doc__`)
- The wrapper function:
 - Starts the timer using `time.time()`
 - Calls the wrapped function (`func`) with the provided arguments
 - Calculates the elapsed time after execution
 - If the `debug` parameter is `True`, it prints the function name and execution time
- The decorator is applied to functions `add` and `factorial` to demonstrate its functionality

Decorators - Answer

```
import time
from functools import wraps

def timing(func):
    @wraps(func)
    def wrapper(*args, debug=False, **kwargs):
        start_time = time.time() # Start the timer
        result = func(*args, **kwargs) # Call the function
        elapsed_time = time.time() - start_time # Calculate execution time
        if debug:
            print(f"Function '{func.__name__}' executed in {elapsed_time:.5f} seconds.")
        return result
    return wrapper
```

Explanation:

- The timing decorator measures a function's execution time
- `@wraps(func)` preserves the original function's metadata
- It calculates execution time using `time.time()`
- The debug flag controls whether timing info is printed

Agenda

- Finals Logistics
- Unix, including bash shell scripts
- Python review; Recursion, List Comprehensions, Decorators, Memoization, & Exceptions
- Regular Expressions, Expressions & Procedures
- Data Structures; Stacks, Queues, Hash Tables and Heaps
- Object Oriented Programming
- SQL
- Python Byte Code & Machine Architecture
- Cryptography
- Final Tips

Python, Memoization

Python, Memoization

What is Memoization?

- A technique to optimize functions by storing the results of expensive function calls and reusing them when the same inputs occur again
- Saves computation time for recursive or repetitive tasks

How Does It Work?

- Stores results in a cache (e.g., dictionary) with inputs as keys and outputs as values
- Checks the cache before computing a result

Why Use Memoization?

- Efficiently handles overlapping subproblems, especially in recursion
- Prevents redundant calculations, speeding up performance

Python, Memoization

Using `functools.lru_cache`:

- Python's built-in decorator for automatic memoization
- No need to manage cache manually

Advantages of `lru_cache`:

- Simple to implement
- Automatically handles cache management and size limits
- Improves readability and reduces code complexity

When to Use Memoization:

- Recursive problems like Fibonacci, factorial, or dynamic programming
- Scenarios where function inputs and outputs are deterministic (no side effects)

Example with `@lru_cache`:

```
python

from functools import lru_cache

@lru_cache(maxsize=None)
def fib(n):
    if n <= 1:
        return n
    return fib(n - 1) + fib(n - 2)

print(fib(10))  # Output: 55
```

Agenda

- Finals Logistics
- Unix, including bash shell scripts
- Python review; Recursion, List Comprehensions, Decorators, Memoization, & Exceptions
- Regular Expressions, Expressions & Procedures
- Data Structures; Stacks, Queues, Hash Tables and Heaps
- Object Oriented Programming
- SQL
- Python Byte Code & Machine Architecture
- Cryptography
- Final Tips

Python, Exceptions

Python, Exceptions

What are Exceptions?

- Errors that occur during program execution, disrupting the normal flow of the program
- Examples: ZeroDivisionError, FileNotFoundError, ValueError, etc

Why Handle Exceptions?

- Ensures programs can respond gracefully to errors instead of crashing
- Helps maintain user experience and data integrity

Common Built-in Exceptions:

- ZeroDivisionError: Raised when dividing by zero
- IndexError: Raised when accessing an invalid list index
- KeyError: Raised when accessing a nonexistent dictionary key
- FileNotFoundError: Raised when a file or directory is not found

Basic Exception Handling:

- Use `try`, `except`, `else`, and `finally` blocks.
- Syntax:

```
python

try:
    # Code that might raise an exception
except SomeError:
    # Code to handle the exception
else:
    # Code to execute if no exception occurs
finally:
    # Code to execute regardless of what happens
```

Example:

```
python

try:
    num = int(input("Enter a number: "))
    print(10 / num)
except ZeroDivisionError:
    print("You can't divide by zero!")
except ValueError:
    print("Invalid input. Please enter a number.")
```

Agenda

- Finals Logistics
- Unix, including bash shell scripts
- Python review; Recursion, List Comprehensions, Decorators, Memoization, & Exceptions
- Regular Expressions, Expressions & Procedures
- Data Structures; Stacks, Queues, Hash Tables and Heaps
- Object Oriented Programming
- SQL
- Python Byte Code & Machine Architecture
- Cryptography
- Final Tips
- Practice Problems

Regular Expressions

Regular Expressions

Regular expressions will not be on the final exam.

Agenda

- Finals Logistics
- Unix, including bash shell scripts
- Python review; Recursion, List Comprehensions, Decorators, Memoization, & Exceptions
- Regular Expressions, **Expressions** & Procedures
- Data Structures; Stacks, Queues, Hash Tables and Heaps
- Object Oriented Programming
- SQL
- Python Byte Code & Machine Architecture
- Cryptography
- Final Tips
- Practice Problems

Python Expressions

What is a Python Expression?

- A combination of values, variables, operators, and function calls that is evaluated to produce another value
- Expressions are the building blocks of Python programs

Best Practices:

- Use parentheses () to clarify operator precedence
- Keep expressions simple and readable
- Break down complex expressions into multiple statements when needed

Arithmetic Expressions: Perform mathematical calculations.

```
python  
  
result = 3 + 5 * 2 # Output: 13
```

Comparison Expressions: Compare values and return a boolean.

```
python  
  
is_equal = (5 == 5) # Output: True
```

Logical Expressions: Combine boolean values using logical operators.

```
python  
  
result = (5 > 3) and (2 < 4) # Output: True
```

String Expressions: Operate on strings.

```
python  
  
greeting = "Hello, " + "World!" # Output: "Hello, World!"
```

Python Expressions – Practice Problem

Write the values of ONLY 3 of the following underlined Python expressions. No errors occur.

>>> list(reversed([5, 4, 3, 2])).pop()

>>> sorted(['apple', 'pie', 'is', 'amazing'], key=len)

>>> {x**2 for x in range(5)}

>>> [x.upper() for x in 'python' if x in 'aeiou']

>>> {i: chr(97 + i) for i in range(5)}

Python Expressions - Answers

```
>>> list(reversed([5, 4, 3, 2])).pop() → 2
```

reversed() reverses the list [5, 4, 3, 2] to [2, 3, 4, 5]

The pop() method removes and returns the last element, which is 2

```
>>> sorted(['apple', 'pie', 'is', 'amazing'], key=len) → ['is', 'pie', 'apple', 'amazing']
```

The sorted() function arranges the strings by their length using the key=len argument

```
>>> {x**2 for x in range(5)} → {0, 1, 4, 9, 16}
```

The set comprehension computes the square of each number in range(5) (0 through 4), resulting in {0, 1, 4, 9, 16}

```
>>> [x.upper() for x in 'python' if x in 'aeiou'] → [ ]
```

The list comprehension checks each letter in 'python' to see if it is in 'aeiou'

Since no vowels are present, the result is an empty list []

```
>>> {i: chr(97 + i) for i in range(5)} → {0: 'a', 1: 'b', 2: 'c', 3: 'd', 4: 'e'}
```

The dictionary comprehension maps each number in range(5) to its corresponding ASCII character (chr(97 + i)), starting from 'a' for 97

Agenda

- Finals Logistics
- Unix, including bash shell scripts
- Python review; Recursion, List Comprehensions, Decorators, Memoization, & Exceptions
- Regular Expressions, Expressions & Procedures
- Data Structures; Stacks, Queues, Hash Tables and Heaps
- Object Oriented Programming
- SQL
- Python Byte Code & Machine Architecture
- Cryptography
- Final Tips
- Practice Problems

Python Procedures

Python Procedures

Python procedures are functions that encapsulate reusable code for a specific task. They can take inputs (arguments) and may return an output.

Purpose:

- Simplify complex problems by breaking them into smaller tasks
- Promote code reuse and readability

Key Characteristics:

- Defined using the `def` keyword
- Can accept multiple arguments
- May include conditionals, loops, and other logical constructs
- Can return a single value, multiple values, or nothing (`None`)

Python Procedures - Steps to Solve Python Procedure Problems

Understand the Problem:

- Read the problem statement carefully; identify inputs, outputs, and required operations

Plan Your Solution:

- Break the task into smaller steps & write pseudocode or comments to outline your logic

Define the Procedure:

- Use the def keyword followed by the procedure name and parameters
- Include meaningful parameter names

Implement the Logic:

- Use Python constructs like loops, conditionals, and comprehensions to solve the problem
- Ensure edge cases are handled (e.g., empty lists, invalid inputs)

Test Your Procedure:

- Run the procedure with multiple test cases to verify correctness
- Check edge cases and expected outputs

Python Procedures - Example

Write a Python function `square_cubes(n)` that returns a list of all numbers less than `n` that are perfect squares or perfect cubes (or both).

Use a list comprehension for full credit.

```
>>> square_cubes(20)
[1, 4, 8, 9, 16]
>>> square_cubes(50)
[1, 4, 8, 9, 16, 25, 27, 36, 49]
>>> square_cubes(100)
[1, 4, 8, 9, 16, 25, 27, 36, 49, 64, 81]
```

Hint: A number x is a perfect square if \sqrt{x} is an integer, and a perfect cube if $x^{1/3}$ is an integer.

Python Procedures – How to approach?

Understand the Problem:

- The task is to identify numbers less than n that are either perfect squares or perfect cubes
- Recognize that a number n is:
 - A perfect square if \sqrt{x} is an integer
 - A perfect cube if $x^{1/3}$ is an integer

Plan the Solution:

- Iterate over all numbers from 1 to $n - 1$
- Check if each number meets either the perfect square or perfect cube condition
- Use a list comprehension to collect all valid numbers in one step

Implement the Logic:

- Use the mathematical properties:
 - $\text{int}(x ** 0.5) ** 2 == x$ for perfect squares
 - $\text{int}\left(\text{round}\left(x^{\frac{1}{3}}\right)\right) ** 3 == x$ for perfect cubes
- Combine these checks in the condition of the list comprehension

Python Procedures – Solution

```
def square_cubes(n):  
    return [x for x in range(1, n) if int(x ** 0.5) ** 2 == x or int(round(x ** (1/3))) ** 3 == x]
```

- The function iterates through all numbers from 1 to $n - 1$
- It checks if the number is a perfect square using $\text{int}(x ** 0.5) ** 2 == x$
- It also checks if the number is a perfect cube using $\text{int}\left(\text{round}\left(x^{\frac{1}{3}}\right)\right) ** 3 == x$
- If either condition is true, the number is added to the list

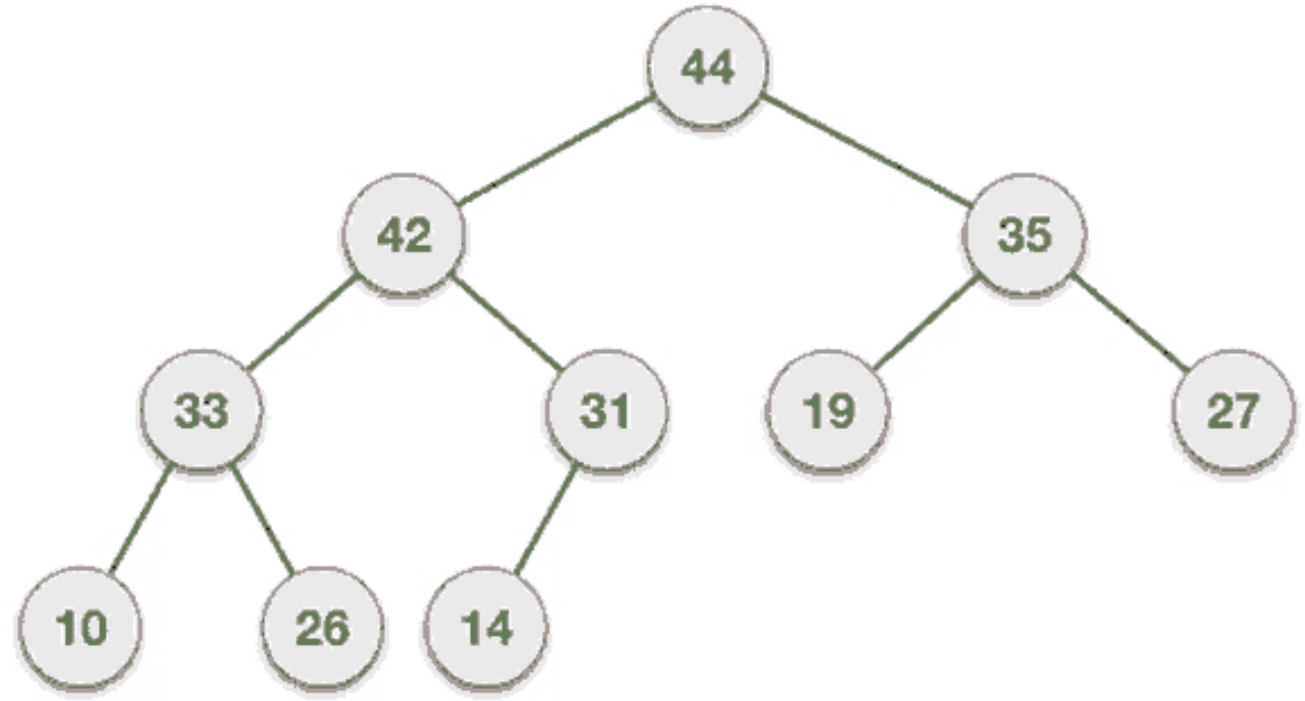
Remember for that to get full credit, you must write it as a list comprehension

Agenda

- Finals Logistics
- Unix, including bash shell scripts
- Python review; Recursion, List Comprehensions, Decorators, Memoization, & Exceptions
- Regular Expressions, Expressions & Procedures
- Data Structures; Stacks, Queues, Hash Tables and Heaps
- Object Oriented Programming
- SQL
- Python Byte Code & Machine Architecture
- Cryptography
- Final Tips
- Practice Problems

Data Structures

Stacks, Queues, Hash Tables and Heaps



Data Structures; Stacks, Queues, Hash Tables and Heaps

Why Use Data Structures?

Organize and manage data efficiently. Solve problems with optimal performance for different use cases.

Stacks

Definition: Last In, First Out (LIFO)

Operations:

- push: Add an item to the stack
- pop: Remove the top item
- peek: View the top item without removing it

Use Cases:

- Undo functionality
- Parsing expressions

```
stack = []
stack.append(1)  # Push
stack.append(2)
print(stack.pop())  # Output: 2
```

Queues

Definition: First In, First Out (FIFO)

Operations:

- enqueue: Add an item to the queue
- dequeue: Remove the front item

Use Cases:

- Task scheduling
- Breadth-first search (BFS)

```
from collections import deque
queue = deque()
queue.append(1)  # Enqueue
queue.append(2)
print(queue.popleft())  # Output: 1
```


Data Structures; Stacks, Queues, Hash Tables and Heaps

Hash Tables

Definition: Key-value pairs for fast lookups

Implemented Using: Python's dict

Use Cases:

- Database indexing
- Caching

```
hash_table = {"key1": "value1", "key2": "value2"}  
print(hash_table["key1"]) # Output: value1
```

Heaps

Definition: A binary tree where the parent is smaller (min-heap) or larger (max-heap) than its children

Use Cases:

- Priority queues
- Efficient sorting

Implemented Using: heapq module in Python

```
import heapq  
heap = []  
heapq.heappush(heap, 10)  
heapq.heappush(heap, 5)  
print(heapq.heappop(heap)) # Output: 5
```

Best Practices

- Choose the right structure for your use case
- Stacks and queues are simple but effective for sequential operations
- Hash tables are ideal for fast lookups with unique keys
- Use heaps for priority-based operations and efficient minimum/maximum retrieval

Agenda

- Finals Logistics
- Unix, including bash shell scripts
- Python review; Recursion, List Comprehensions, Decorators, Memoization, & Exceptions
- Regular Expressions, Expressions & Procedures
- Data Structures; Stacks, Queues, Hash Tables and Heaps
- Object Oriented Programming
- SQL
- Python Byte Code & Machine Architecture
- Cryptography
- Final Tips
- Practice Problems

Python, Object-Oriented Programming

Object-Oriented Programming (OOP)

A programming paradigm based on the concept of "**objects**" that combine **data** (attributes) and **behavior** (methods). Encourages modular, reusable, and organized code.

Core Principles of OOP

Encapsulation: Bundling data and methods that operate on the data into a single unit (class)

Inheritance: Creating new classes based on existing ones to promote code reuse

Polymorphism: Allowing objects to be treated as instances of their parent class, enabling flexibility and dynamic behavior

Abstraction: Hiding complex implementation details and exposing only the necessary functionalities

◦ OOP - Key Features in Python OOP

Classes: Define the blueprint for objects (e.g., class Car)

Attributes: Variables within a class that hold data (e.g., name, mpg, year)

Methods: Functions within a class that define behaviors (e.g., best(), pp())

Magic Methods: Special methods like `__init__`, `__repr__`, and `__str__` to customize object behavior

Static Methods: Defined using `@staticmethod` and do not require access to class or instance data.

◦ OOP – How to approach?

Understand the Problem Statement:

- Identify the entities (real-world objects) involved in the problem
- Recognize their attributes (data) and behaviors (actions)

Define the Classes:

- Determine the classes needed to model the entities in the problem
- Create a class for each entity with appropriate attributes and methods

Implement Core OOP Principles:

- Use **encapsulation** to bundle related data and methods into a single class
- Leverage **inheritance** if there are relationships between entities (e.g., a Truck class inherits from a Vehicle class)

- Apply **polymorphism** for flexible behavior (e.g., methods with the same name but different functionality across classes)

Write the Code in Steps:

- Start by defining the `__init__` method to initialize the attributes
- Add other necessary methods to define behavior (e.g., `best()` for comparing cars by specific criteria)
- Implement special methods like `__str__` or `__repr__` to customize the representation of objects

Test the Solution:

- Create objects (instances of classes) to test functionality
- Use the provided test cases or examples to verify correctness
- Debug and refine the code based on output

Agenda

- Finals Logistics
- Unix, including bash shell scripts
- Python review; Recursion, List Comprehensions, Decorators, Memoization, & Exceptions
- Regular Expressions, Expressions & Procedures
- Data Structures; Stacks, Queues, Hash Tables and Heaps
- Object Oriented Programming
- SQL
- Python Byte Code & Machine Architecture
- Cryptography
- Final Tips
- Practice Problems

Python, SQL

```
</>
92 JOIN inbound."in.system" sys ON sys."id" = cp."system_id"
93 LEFT JOIN public."pivoted_pairs" nvp ON nvp."contentpost_id" = cp."id"
94 LEFT JOIN inbound."in.rejectionreason" rr ON rr."id" = cp."rejection_reason_id"
95 LEFT JOIN public."public.overviewoption" iop ON iop."cust_application_id" = cp."id"
96 LEFT JOIN public."public.overviewoption" iop ON iop."topic" = cp."topic_id" AND iop."cust_application_id" = cp."id"
97 LEFT JOIN public."sort_number" sn ON sn."content_id" = cp."content_id" AND sn."cons_id" = cp."cons_id";
98
99 CREATE TABLE public."public.overviewoption" AS
100 SELECT
101     iop."id" AS "insight_option_id",
102     iop."field name",
103     cpiop."cust_application_id",
104     ins."customer consolidated",
105 FROM public."overviewoption" iop
106 JOIN inbound."in.overviewoption" iop ON iop."insight_id" = iop."insight_id"
107 JOIN inbound."in.article_option" iop ON iop."insight_option_id" = iop."id";
108
109 CREATE TABLE public."public.overviewoption" AS
110 SELECT
111     iop."id" AS "insight_option_id",
112     iop."field name",
113     cpiop."cust_application_id",
114     ins."customer consolidated",
115 FROM public."overviewoption" iop
116 JOIN inbound."in.overviewoption" iop ON iop."insight_id" = iop."insight_id"
117 JOIN inbound."in.article_option" iop ON iop."insight_option_id" = iop."id";
118
119 CREATE TABLE public."public.overviewoption" AS
120 SELECT
121     "cons_id",
122     COUNT("id") AS "post_count",
123 FROM public."bridge" b
124 WHERE "cons_id" <> ''
125 AND "mood" = 'Positive'
126 GROUP BY "cons_id";
127
128 CREATE TABLE tmp."
```


SQL

What is SQL?

- **Structured Query Language (SQL)** is a standard programming language designed for managing and querying relational databases
- Allows interaction with databases to:
 - **Create tables** to store structured data
 - **Insert, update, and delete records** in tables
 - **Retrieve data** using complex queries

Key Features:

- **Data Manipulation:** Insert, update, delete, and retrieve data
- **Data Definition:** Create and modify database schemas (tables, columns, etc.)
- **Data Querying:** Use SELECT statements with filters like WHERE, GROUP BY, and ORDER BY
- **Data Integrity:** Enforce constraints like primary keys, foreign keys, and unique fields
- **Scalability:** Widely used in applications of all sizes, from small apps to enterprise systems

SQL – How to solve?

Understand the Requirement:

- Analyze what the question is asking (e.g., creating tables, filtering rows, aggregating data)
- Identify the key tables and fields involved

Break Down the Problem:

- Split tasks into smaller steps:
 - Table creation (CREATE TABLE)
 - Data insertion (INSERT INTO)
 - Data querying (SELECT, WHERE, ORDER BY, etc.)

Write SQL Commands Incrementally:

- Start with basic SELECT queries, then add filters and sorting
- Test queries to verify intermediate results

Use SQL Tools:

- Use tools like sqlite3, MySQL Workbench, or database shells for testing queries
- Use SQL functions like SUM(), COUNT(), AVG(), etc., for calculations

Optimize Queries:

- Look for opportunities to improve performance using indexing, subqueries, or joins

SQL - Example

Create the SQL table **library** and related queries which have the following behavior. (Define table_creation, library_data, query1, and query2).

```
#!/usr/bin/env python3
import sqlite3

# Connect to SQLite database
connection = sqlite3.connect("library.db")
cursor = connection.cursor()

# SQL command to create the library table
table_creation = """

cursor.execute("DROP TABLE IF EXISTS library")
cursor.execute(table_creation)

# Sample data for library
library_data = [

```

```
# Insert data into library table
def load_books():
    for book in library_data:
        insert_command = """
        INSERT INTO library (id, title, author, genre, year)
        VALUES ({id}, "{title}", "{author}", "{genre}", {year});
        """.format(
            id=book[0], title=book[1], author=book[2], genre=book[3], year=book[4]
        )
        cursor.execute(insert_command)

load_books()

# Query examples
query1 = 
query2 = 

# Execute queries and print results
def try_query(command):
    try:
        cursor.execute(command)
        result = cursor.fetchall()
        for row in result:
            print(row)
    except Exception as e:
        print(f"Error: {e}")

# Execute the queries
try_query(query1)
try_query(query2)
```

SQL - Example

Tasks

1. Define table_creation to create the **library** table.
2. Populate the table with the given library_data.
3. Write and execute two queries:
 - Query 1: Select books published before 1950.
 - Query 2: Select books in the "Fiction" genre, ordered by year in descending order.

Query 1:

plaintext

 Copy code

```
SELECT [REDACTED]  
(2, '1984', 'George Orwell', 'Dystopian', 1949)  
(3, 'Pride and Prejudice', 'Jane Austen', 'Romance', 1813)  
(4, 'The Great Gatsby', 'F. Scott Fitzgerald', 'Fiction', 1925)
```

Query 2:

plaintext

 Copy code

```
SELECT [REDACTED]  
(4, 'The Great Gatsby', 'F. Scott Fitzgerald', 'Fiction', 1925)  
(1, 'To Kill a Mockingbird', 'Harper Lee', 'Fiction', 1960)
```

SQL - Answer

Understand the Requirements:

- Clearly define the objective (e.g., create a table, query specific data, update records)
- Identify constraints like filters, sorting, or groupings in the query

Break Down the Problem:

- Start with foundational steps (e.g., CREATE TABLE, INSERT)
- Gradually add more complexity with queries (SELECT, WHERE, JOIN)

Write Incrementally:

- Build small queries first and test them
- Add clauses like GROUP BY, ORDER BY, or conditions in WHERE after validating initial outputs

Test and Validate:

- Check results with edge cases (e.g., no data, null values)
- Use tools like sqlite3 or EXPLAIN to debug and optimize performance

Optimize and Refine:

- Use indexing for large datasets
- Write clean, readable SQL with proper formatting and comments

SQL - Answer

Create the SQL table **library** and related queries which have the following behavior. (Define table_creation, library_data, query1, and query2).

```
#!/usr/bin/env python3
import sqlite3

# Connect to SQLite database
connection = sqlite3.connect("library.db")
cursor = connection.cursor()

# SQL command to create the library table
table_creation = """
CREATE TABLE library (
    id INTEGER PRIMARY KEY,
    title TEXT,
    author TEXT,
    genre TEXT,
    year INTEGER
);
"""

cursor.execute("DROP TABLE IF EXISTS library")
cursor.execute(table_creation)

# Sample data for library
library_data = [
    (1, "To Kill a Mockingbird", "Harper Lee", "Fiction", 1960),
    (2, "1984", "George Orwell", "Dystopian", 1949),
    (3, "Pride and Prejudice", "Jane Austen", "Romance", 1813),
    (4, "The Great Gatsby", "F. Scott Fitzgerald", "Fiction", 1925),
]
```

```
# Insert data into library table
def load_books():
    for book in library_data:
        insert_command = """
INSERT INTO library (id, title, author, genre, year)
VALUES ({id}, "{title}", "{author}", "{genre}", {year});
""".format(
        id=book[0], title=book[1], author=book[2], genre=book[3], year=book[4]
    )
        cursor.execute(insert_command)

load_books()

# Query examples
query1 = "SELECT * FROM library WHERE year < 1950;"
query2 = "SELECT * FROM library WHERE genre = 'Fiction' ORDER BY year DESC;"

# Execute queries and print results
def try_query(command):
    try:
        cursor.execute(command)
        result = cursor.fetchall()
        for row in result:
            print(row)
    except Exception as e:
        print(f"Error: {e}")

# Execute the queries
try_query(query1)
try_query(query2)
```

SQL - Answer

Query 1:

plaintext

 Copy code

```
SELECT * FROM library WHERE year < 1950;  
(2, '1984', 'George Orwell', 'Dystopian', 1949)  
(3, 'Pride and Prejudice', 'Jane Austen', 'Romance', 1813)  
(4, 'The Great Gatsby', 'F. Scott Fitzgerald', 'Fiction', 1925)
```

Query 2:

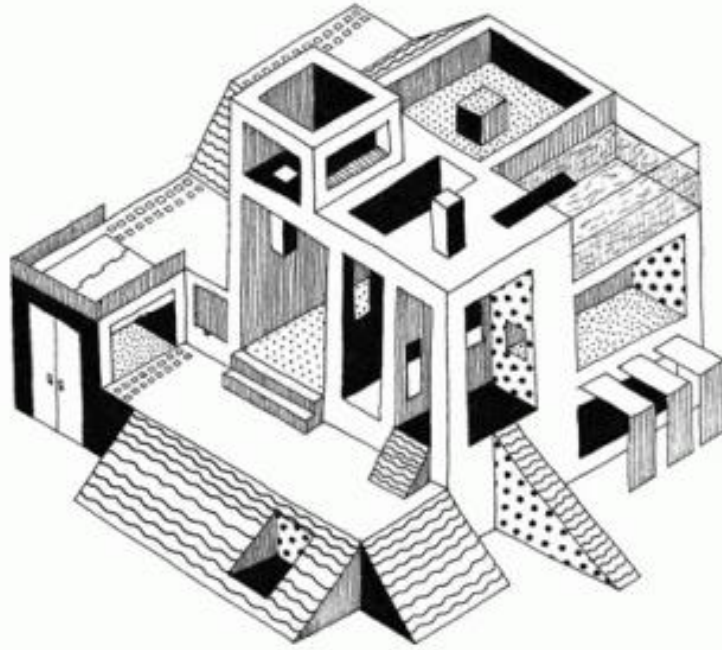
plaintext

 Copy code

```
SELECT * FROM library WHERE genre = 'Fiction' ORDER BY year DESC;  
(4, 'The Great Gatsby', 'F. Scott Fitzgerald', 'Fiction', 1925)  
(1, 'To Kill a Mockingbird', 'Harper Lee', 'Fiction', 1960)
```

Agenda

- Finals Logistics
- Unix, including bash shell scripts
- Python review; Recursion, List Comprehensions, Decorators, Memoization, & Exceptions
- Regular Expressions, Expressions & Procedures
- Data Structures; Stacks, Queues, Hash Tables and Heaps
- Object Oriented Programming
- SQL
- Python Byte Code & Machine Architecture
- Cryptography
- Final Tips
- Practice Problems



Python, Bytecode and Machine Architecture

Python, Bytecode

Intermediate Representation:

- Python code is first compiled into **bytecode**, a low-level, platform-independent representation
- Bytecode is stored as .pyc files in the `__pycache__` directory

Execution:

- The Python Virtual Machine (PVM) interprets the bytecode to execute the program

Efficient Execution:

- Bytecode bridges the gap between human-readable Python code and the underlying machine instructions

Analyzable:

- Tools like the `dis` module can disassemble Python bytecode, enabling performance optimizations and debugging

Python, Machine Architecture

Underlying Hardware:

- Machine architecture refers to the design and functionality of the physical computer hardware, such as CPUs, memory, and I/O

Execution Flow:

- Python bytecode ultimately interacts with the machine's architecture through system calls and the Python interpreter

Registers and Instructions:

- Low-level operations, such as comparisons and arithmetic, are carried out using the machine's instruction set (e.g., x86-64, ARM)

Performance Optimization:

- Understanding architecture helps optimize Python programs, especially when interfacing with compiled libraries or handling intensive tasks

Bytecode & Machine Architecture - Example

Define a Python procedure

calculate_bytecode()

that generates the following bytecode.

What is the name of this function?

```
dis.dis(calculate_bytecode)
2          0 RESUME                0
          2 LOAD_CONST             1 (20)
          4 STORE_FAST              0 (x)

3          6 LOAD_FAST              0 (x)
          8 LOAD_CONST             2 (5)
         10 BINARY_OP              6 (%)
         14 LOAD_CONST             3 (0)
         16 COMPARE_OP             2 (==)
         18 POP_JUMP_FORWARD_IF_FALSE 24 (to 48)

4          20 LOAD_FAST             0 (x)
         22 LOAD_CONST             2 (5)
         24 BINARY_OP             11 (/)
         28 RETURN_VALUE

5      >>  30 LOAD_CONST            4 (10)
          32 LOAD_FAST              0 (x)
          34 BINARY_OP             5 (*)
          36 RETURN_VALUE
```

Bytecode & Machine Architecture - Answer

Key Features of This Example

Bytecode Instruction: It breaks down into key operations like `LOAD_CONST`, `STORE_FAST`, and `BINARY_OP`

Logic Matching: The if-else statement checks divisibility and handles two separate return paths

Numeric Constants: Demonstrates constants used in mathematical operations

Control Flow: `POP_JUMP_FORWARD_IF_FALSE` handles conditional branching

Bytecode & Machine Architecture - Answer

```
def calculate_bytecode():  
    # Initialize variable `x` with a constant value 20  
    x = 20  
    # Check if `x` is divisible by 5  
    # The condition `x % 5 == 0` uses modulus to verify divisibility  
    if x % 5 == 0:  
        # If divisible, return the result of x divided by 5  
        # The division result will be a float (e.g., 4.0 if x = 20)  
        return x / 5  
    else:  
        # If not divisible, return the product of 10 and `x`  
        # This handles the case where `x % 5 != 0`  
        return 10 * x
```

Agenda

- Finals Logistics
- Unix, including bash shell scripts
- Python review; Recursion, List Comprehensions, Decorators, Memoization, & Exceptions
- Regular Expressions, Expressions & Procedures
- Data Structures; Stacks, Queues, Hash Tables and Heaps
- Object Oriented Programming
- SQL
- Python Byte Code & Machine Architecture
- Cryptography
- Final Tips
- Practice Problems

Cryptography



Cryptography

Cryptography is the science and practice of secure communication in the presence of adversaries. It involves techniques to encode (encrypt) and decode (decrypt) information to ensure data confidentiality, integrity, and authenticity.

Key goals of Cryptography include:

- **Confidentiality:** Ensure that information is only accessible to those with the correct key/permission
- **Integrity:** Protect data from being altered without detection
- **Authentication:** Verify the identity of users or systems involved in communication
- **Non-repudiation:** Prevent parties from denying their actions, ensuring accountability

Applications Include:

- **Secure Communication:** Encrypting emails, chats, or messages
- **Data Protection:** Securing passwords, personal information, and financial data
- **Authentication:** Enabling technologies like digital signatures, multi-factor authentication, and secure logins
- **Blockchain & Cryptocurrency:** Ensuring integrity and security in decentralized systems

Cryptography

Basic Techniques:

- **Encryption:** Transform plaintext into unreadable ciphertext using an algorithm and a key
- **Decryption:** Reverse the encryption process using the same or a related key to retrieve the plaintext
- **Hashing:** Convert data into a fixed-size hash value, often used for verifying integrity
- **Digital Signatures:** A cryptographic method to verify the authenticity and integrity of digital data

Types of Cryptography:

- **Symmetric Cryptography:** A single key is used for both encryption and decryption
 - Example: AES, DES
- **Asymmetric Cryptography:** A pair of keys (public and private) is used
 - Example: RSA, ECC
- **Hash Functions:** One-way functions that produce a unique output for a given input
 - Example: SHA-256

Hint: You should review the topic from HW7

Cryptography - Questions

1. What is the difference between symmetric and asymmetric encryption? Provide examples of when each might be used.
2. Describe the role of a hash function in cryptography. Why is it important that hash functions are collision-resistant?
3. Compare and contrast hashing and encryption. Provide an example of when you would use each.
4. Define confidentiality, integrity, and availability in the context of cryptography. How do they collectively contribute to data security?

Cryptography - Answers

What is the difference between symmetric and asymmetric encryption? Provide examples of when each might be used.

Answer:

- **Symmetric Encryption:** Uses a single key for both encryption and decryption. Example: AES (Advanced Encryption Standard). Symmetric encryption is commonly used for encrypting data at rest, such as files or databases.
- **Asymmetric Encryption:** Uses a pair of keys (public and private). The public key encrypts, and the private key decrypts. Example: RSA. Asymmetric encryption is often used in secure key exchanges and digital certificates.

Describe the role of a hash function in cryptography. Why is it important that hash functions are collision-resistant?

Answer:

- A hash function maps input data to a fixed-length string, which is unique to the input. Example: SHA-256.
- **Collision resistance** ensures that two different inputs cannot produce the same hash. This property is critical for verifying data integrity and securing passwords, as collisions could allow attackers to substitute malicious data undetected.

Cryptography - Answers

Compare and contrast hashing and encryption. Provide an example of when you would use each.

Answer:

- **Hashing:** Converts data into a fixed-length hash, which cannot be reversed. Used for integrity checks and password storage (e.g., SHA-256).
- **Encryption:** Encodes data to make it unreadable without a decryption key. Used for securing communication and files (e.g., AES for file encryption).

Define confidentiality, integrity, and availability in the context of cryptography. How do they collectively contribute to data security?

Answer:

- **Confidentiality:** Ensures that only authorized parties can access information (e.g., encryption).
- **Integrity:** Ensures data is not altered or tampered with (e.g., hash functions).
- **Availability:** Ensures that authorized users have access to information when needed (e.g., robust systems with redundancy).
- Together, they form the foundation of the CIA triad, critical for protecting sensitive data.

Agenda

- Finals Logistics
- Unix, including bash shell scripts
- Python review; Recursion, List Comprehensions, Decorators, Memoization, & Exceptions
- Regular Expressions, Expressions & Procedures
- Data Structures; Stacks, Queues, Hash Tables and Heaps
- Object Oriented Programming
- SQL
- Python Byte Code & Machine Architecture
- Cryptography
- Final Tips
- Practice Problems

Final Tips

Understand Problem-Solving Patterns:

- Break down problems systematically (as shown in "How to Approach" sections)
- Follow steps like identifying base cases in recursion or designing SQL queries incrementally

Utilize the Cheat Sheet Wisely:

- Include complex syntax or uncommon functions, such as decorators, Python bytecode operations, or machine-level optimizations, that may not be easily remembered

Leverage Provided Resources:

- Revisit UNIX tutorials, Python guides, and the course's practice exams to solidify concepts
- Utilize the zoo server commands for hands-on UNIX practice

Focus on Key Code Components:

- Review annotated code examples provided in slides to ensure you understand the logic and behavior of sample solutions

Simulate Exam Conditions:

- Practice under time constraints to improve problem-solving speed and familiarity with the practice exam test

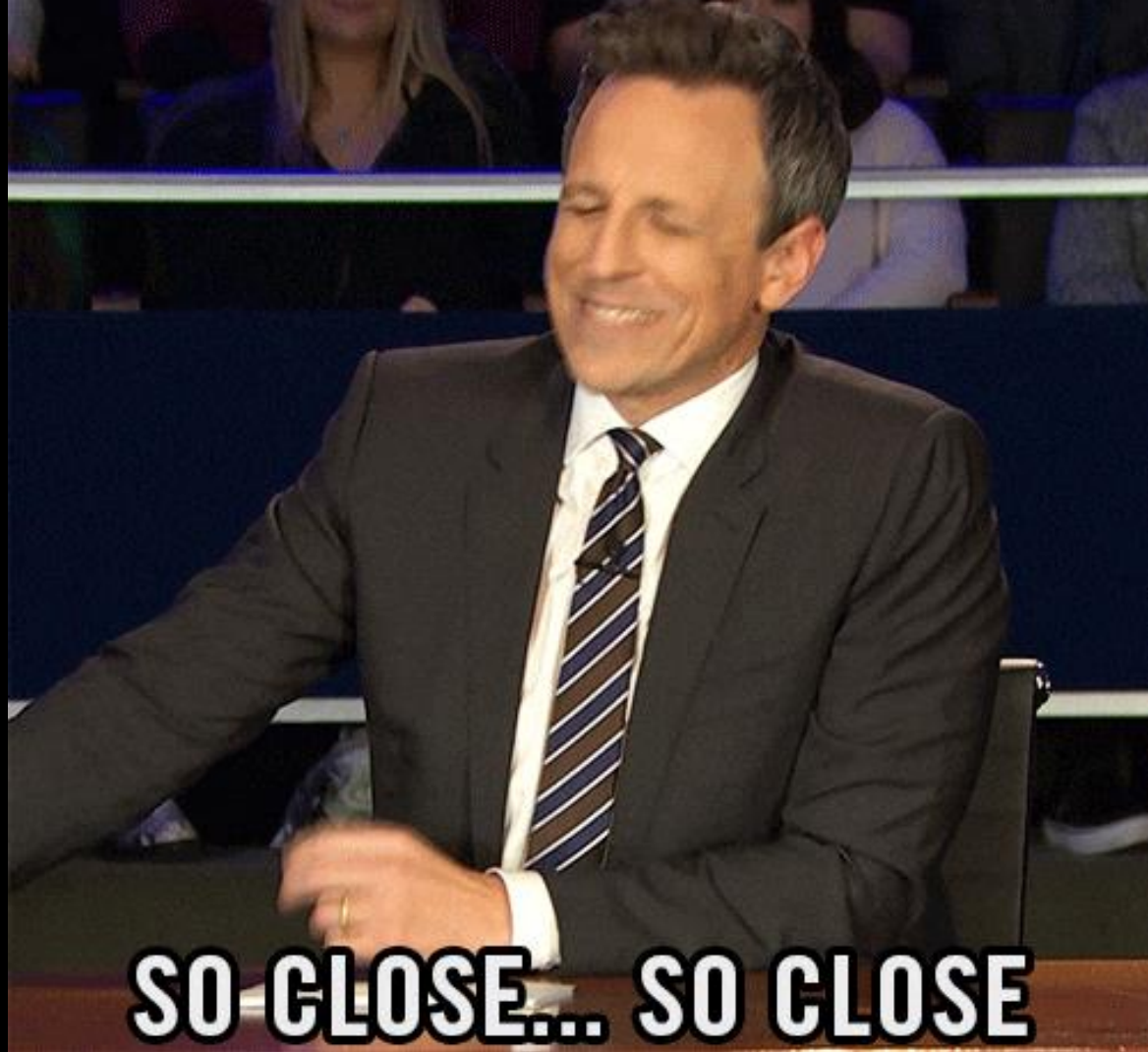
Practice Problems

Additional Practice Problems

Don't forget that there is still a practice exam that you can work on!

Practice [Exam](#) / [Solutions](#)

There's plenty of practice available to you.



You got this