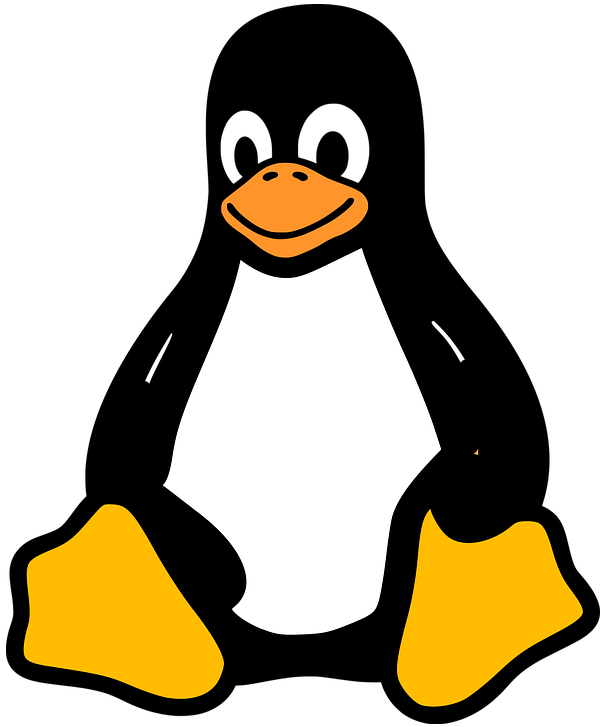


CS 201 Midterm 2 Review
Fall 2024

Agenda

- Midterm Logistics
- Reviewing resources
- Unix
- Python review
  - Python Virtual Machine (PVM)
  - OOP Methods and Basic Data Structures (Stacks, Queues, and Hash Tables)
  - Exceptions
  - Iterators
  - Decorators
  - Recursion
- Final Tips

Agenda

- Midterm Logistics
- Reviewing resources
- Unix
- Python review
  - Python Virtual Machine (PVM)
  - OOP Methods and Basic Data Structures (Stacks, Queues, and Hash Tables)
  - Exceptions
  - Iterators
  - Decorators
  - Recursion
- Final Tips

Logistics - Midterm 2

Thursday, November 7th at 7pm in ML 211

2-hour hand-written exam

No computers, notes, or books

For special accommodations, please reach out to Professor Slade ASAP

Agenda

- Midterm Logistics
- Reviewing resources
- Unix
- Python review
  - Python Virtual Machine (PVM)
  - OOP Methods and Basic Data Structures (Stacks, Queues, and Hash Tables)
  - Exceptions
  - Iterators
  - Decorators
  - Recursion
- Final Tips

Available Resources

- 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 Transcript / Solutions
    - 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)
- Reviewing problem sets, *especially Pset 4 (hint, hint)*, and each other (it never hurts to make study groups)!

Agenda

- Midterm Logistics
- Reviewing resources
- Unix
- Python review
  - Python Virtual Machine (PVM)
  - OOP Methods and Basic Data Structures (Stacks, Queues, and Hash Tables)
  - Exceptions
  - Iterators
  - Decorators
  - Recursion
- Final Tips

UNIX will cover through Principle Three

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.

Two examples of UNIX scenarios

UNIX

- Principles 1-3 → Some highlights of III:
  - `diff`
  - `touch`
  - `grep`
  - `file`
  - `whoami`
  - `id`
  - `uptime`
- Difference between piping and redirection:
  - Piping (`|`) takes output and sends to to another **program**
  - Redirection (`>`, `&>`, `2>`, `>>`, etc.) takes output and sends to to either a **file** or a **stream**.

UNIX

```
[crb84@scorpion tmp]$ ls
Friday  test  test_01  test_02
[crb84@scorpion tmp]$ XXXX1
[crb84@scorpion tmp]$ ls
files  Friday  test  test_01  test_02
[crb84@scorpion tmp]$ ls > files_new
[crb84@scorpion tmp]$ XXXX2
0a1,3
> files
> files_new
> Friday
[crb84@scorpion tmp]$
```

UNIX

```
[crb84@scorpion tmp]$ ls
Friday  test  test_01  test_02
[crb84@scorpion tmp]$ ls | grep test > files
[crb84@scorpion tmp]$ ls
files  Friday  test  test_01  test_02
[crb84@scorpion tmp]$ ls > files_new
[crb84@scorpion tmp]$ diff files files_new
0a1,3
> files
> files_new
> Friday
[crb84@scorpion tmp]$
```

UNIX

```
[crb84@scorpion tmp]$ ls -l
total 0
[crb84@scorpion tmp]$ XXXX1
[crb84@scorpion tmp]$ ls -l
total 4
-rw-rw-r-- 1 crb84 crb84 204 Apr 15 21:49 1
[crb84@scorpion tmp]$ XXXX2
       total        used        free      shared  buff/cache   available
```

UNIX

```
[crb84@scorpion tmp]$ ls -l
total 0
[crb84@scorpion tmp]$ free > 1
[crb84@scorpion tmp]$ ls -l
total 4
-rw-rw-r-- 1 crb84 crb84 204 Apr 15 21:49 1
[crb84@scorpion tmp]$ head -n1 1
              total        used        free      shared  buff/cache   available
```

Take the time to complete the tutorial

This is the best way to learn the content

Agenda

- Midterm Logistics
- Reviewing resources
- Unix
- Python review
  - Python Virtual Machine (PVM)
  - OOP Methods and Basic Data Structures (Stacks, Queues, and Hash Tables)
  - Exceptions
  - Iterators
  - Decorators
  - Recursion
- Final Tips

Python... so far

- **Midterm 1 covered:**
  - Procedure Examples
  - Regular Expressions
  - Python Expressions
  - Recursion
  - Comprehensions
  - Object-oriented Programming
- **For Midterm 2, the scope include:**
  - Python Virtual Machine (PVM), OOP Methods, Basic Data Structures (Stacks, Queues, and Hash Tables), Exceptions, Iterators, Decorators, and Recursion.

Agenda

- Midterm Logistics
- Reviewing resources
- Unix
- Python review
  - Python Virtual Machine (PVM)
  - OOP Methods and Basic Data Structures (Stacks, Queues, and Hash Tables)
  - Exceptions
  - Iterators
  - Decorators
  - Recursion
- Final Tips

Python Virtual Machine (PVM)

1. What is it?

The PVM is the core component of the Python interpreter

When you write Python code, it's first converted into **bytecode**, a lower-level, platform-independent representation of your code

This bytecode is then executed by the PVM, which interprets each bytecode instruction and performs the corresponding operations on your machine

Essentially, the PVM acts as the bridge between Python code and your hardware

Python Virtual Machine (PVM)

2. Why does it matter?

Understanding the PVM is important because it helps explain Python's behavior, performance, and certain limitations. Here are a few reasons why it's significant:

- **Portability**: Bytecode can run on any machine with a Python interpreter, making Python code highly portable
- **Memory Management**: The PVM handles memory allocation, garbage collection, and resource cleanup, so understanding it helps developers write efficient, memory-safe programs
- **Execution Flow**: Knowing that Python code goes through a compile phase (to bytecode) and then an interpret phase (by the PVM) helps with understanding debugging, performance optimization, and troubleshooting runtime errors

Python Virtual Machine (PVM)

3. Where can you expect to see this?

- **Debugging and Profiling Tools**: Tools like pdb (Python debugger) and profiling libraries interact with the PVM to monitor and manipulate Python code execution. If you're troubleshooting code performance or runtime errors, you're indirectly working with the PVM

- **Compiled Bytecode Files**: You might see .pyc files generated by Python. These are cached bytecode files created to speed up subsequent executions of the same script

- **Error Messages**: Certain error messages refer to bytecode or the internals of the PVM, especially if you're dealing with complex or low-level programming in Python

Example 1: PVM

Define a Python function y() that generates the following bytecode in Python version 3.10:

```
>>> dis.dis(y)
  10           0 LOAD_CONST               1 (15)
               2 STORE_FAST               0 (a)

  11           4 LOAD_FAST                0 (a)
               6 LOAD_CONST               2 (3)
               8 BINARY_MODULO
              10 POP_JUMP_IF_FALSE       18

  12          12 LOAD_FAST                0 (a)
              14 RETURN_VALUE

  13      >>  16 LOAD_FAST                0 (a)
              18 LOAD_CONST               3 (5)
              20 BINARY_ADD
              22 RETURN_VALUE
```

How to approach?

To approach this type of bytecode-related problem, follow these steps:

1. **Understand the Bytecode Instructions**: Analyze each bytecode line and understand what operation it performs (e.g., LOAD_CONST, BINARY_MODULO, POP_JUMP_IF_FALSE). Recognize common instructions for assignments, arithmetic operations, and conditionals
2. **Identify Control Flow and Conditions**: Look for jump instructions like POP_JUMP_IF_FALSE to determine where the code branches or checks conditions, helping you identify if-else or loop structures
3. **Map Bytecode to Python Constructs**: Based on the bytecode, infer the high-level Python operations (e.g., variable assignments, conditional statements) that would produce the same bytecode
4. **Determine Variable Values and Expressions**: Identify the constants and operations (like 15 and % 3 in the example) and how they relate to the bytecode operations
5. **Write the Python Function**: Draft a function that matches the inferred structure, ensuring it uses the correct constants, operators, and control flow to generate the expected bytecode

Answer:

The function y() that produces this bytecode is:

```
def y():
    a = 15
    if a % 3:
        return a
    else:
        return a + 5
```

1. **Variable Initialization**: The line a = 15 assigns the value 15 to a. This corresponds to LOAD_CONST and STORE_FAST instructions in the bytecode
2. **Modulo Condition**: The if a % 3: line checks if a modulo 3 is non-zero. If it is, the condition is true, and a is returned directly. This matches the LOAD_FAST, BINARY_MODULO, and POP_JUMP_IF_FALSE instructions
3. **Return Statements**:
   - If the condition is true (meaning a modulo 3 is non-zero), the function returns a directly, which corresponds to the RETURN_VALUE instruction at line 12
   - If the condition is false (meaning a modulo 3 is zero), the function returns a + 5. This is represented by LOAD_FAST, LOAD_CONST, BINARY_ADD, and RETURN_VALUE instructions

Example 2: PVM

Define a Python function y() that generates the following bytecode in Python version 3.10:

```
>>> dis.dis(z)
  10           0 LOAD_CONST               1 (8)
               2 STORE_FAST               0 (b)

  11           4 LOAD_FAST                0 (b)
               6 LOAD_CONST               2 (4)
               8 BINARY_TRUE_DIVIDE
              10 STORE_FAST               1 (c)

  12          12 LOAD_FAST                1 (c)
              14 LOAD_CONST               3 (5.0)
              16 COMPARE_OP               0 (<)
              18 POP_JUMP_IF_FALSE       26

  13          20 LOAD_CONST               4 ('Less than 5')
              22 RETURN_VALUE

  14     >>   26 LOAD_CONST               5 ('5 or more')
              28 RETURN_VALUE
```

Answer:

The function z() that produces this bytecode is:

```
def z():
    b = 8
    c = b / 4
    if c < 5.0:
        return "Less than 5"
    else:
        return "5 or more"
```

1. **Variable Initialization**: b = 8 assigns the value 8 to b, which corresponds to LOAD_CONST and STORE_FAST instructions in the bytecode
2. **Division Operation**: c = b / 4 divides b by 4 and stores the result in c, corresponding to the BINARY_TRUE_DIVIDE and STORE_FAST instructions
3. **Comparison Condition**: if c < 5.0: checks if c is less than 5.0. This comparison is represented by COMPARE_OP with the < operation.
4. **Return Statements**:
    - If c < 5.0 is true, the function returns "Less than 5", matching the RETURN_VALUE instruction at line 20
    - If c >= 5.0, the function returns "5 or more", represented by the RETURN_VALUE instruction at line 28

Example 3: PVM

Provide the bytecode generated for the following Python function, compute_product().

Use dis.dis() format, but without the source code line numbers from the first column. Assume Python version 3.10.

```
def add_numbers():
    a = 5
    b = 10
    result = a + b
    return result
```

How to approach?

1. Break Down Each Line of the Python Function:

- Write out each line of the function separately, including assignments, calculations, and the return statement
- Label each line with what it does (e.g., “assigns a constant,” “performs addition,” “returns a variable”)

2. List Common Bytecode Instructions and Their Purposes:

- Create a small reference guide on the side for common bytecode instructions:
  - **LOAD_CONST**: Loads a constant onto the stack
  - **STORE_FAST**: Stores a variable
  - **LOAD_FAST**: Loads a variable onto the stack
  - **Arithmetic Operations**: Use BINARY_ADD for addition, BINARY_SUBTRACT for subtraction, etc
  - **RETURN_VALUE**: Returns the value at the top of the stack

How to approach (continued)?

3. Assign Offsets Sequentially:

- Start with an offset of 0 for the first instruction and increase by 2 for each subsequent instruction
- Write down the offset before each instruction to maintain the flow (e.g., 0, 2, 4, etc.)

4. Translate Each Line to Bytecode:

- For each line in the function:
  - Determine if it's an assignment, operation, or return
  - Use the reference list to pick the appropriate bytecode instructions
  - For example, if a line is a = 5, write down LOAD_CONST followed by STORE_FAST, increasing the offset by 2 for each step
- Don't forget to include any constants or variable names in parentheses after each instruction (e.g., LOAD_CONST 1 (5) or STORE_FAST 0 (a))

How to approach (continued)?

5. Organize the Bytecode:

- Write the bytecode in a clear list, each line including the offset, instruction, and arguments if applicable
- Leave space between bytecode segments to make it easier to check each section

6. Review for Common Patterns:

- Check that you've included:
  - **Assignments** with LOAD_CONST and STORE_FAST
  - **Arithmetic or Boolean operations** with BINARY_ instructions if needed
  - **Loading and returning** variables correctly with LOAD_FAST and RETURN_VALUE
- Make sure there's a final RETURN_VALUE instruction if the function has a return statement

7. Double-Check Each Offset and Instruction:

- Go through each line to ensure offsets increase correctly, each instruction is appropriate, and all variables/constants are accurately represented in parentheses

Answer

Here's how each line of bytecode corresponds to the original function:

1. **Variable Initialization**:
   - a = 5: LOAD_CONST 1 (5) loads the constant 5 onto the stack, and STORE_FAST 0 (a) stores it in the variable a.
   - b = 10: LOAD_CONST 2 (10) loads the constant 10 onto the stack, and STORE_FAST 1 (b) stores it in the variable b.
2. **Addition Operation**:
   - result = a + b: LOAD_FAST 0 (a) loads a onto the stack, LOAD_FAST 1 (b) loads b onto the stack, and BINARY_ADD adds them together. The result is then stored in result with STORE_FAST 2 (result).
3. **Return Statement**:
   - return result: LOAD_FAST 2 (result) loads result onto the stack, and RETURN_VALUE returns it as the output of the function.

```
 0 LOAD_CONST               1 (5)
 2 STORE_FAST               0 (a)

 4 LOAD_CONST               2 (10)
 6 STORE_FAST               1 (b)

 8 LOAD_FAST                0 (a)
10 LOAD_FAST                1 (b)
12 BINARY_ADD
14 STORE_FAST               2 (result)

16 LOAD_FAST                2 (result)
18 RETURN_VALUE
```

Agenda

- Midterm Logistics
- Reviewing resources
- Unix
- Python review
  - Python Virtual Machine (PVM)
  - OOP Methods and Basic Data Structures (Stacks, Queues, and Hash Tables)
  - Exceptions
  - Iterators
  - Decorators
  - Recursion
- Final Tips

OOP and Basic Data Structures

1. What is it?

- **OOP (Object-Oriented Programming) Methods**: OOP is a programming paradigm that allows you to model real-world entities as objects. In Python, OOP concepts such as classes, methods, inheritance, and encapsulation enable the creation of modular and reusable code. These OOP methods provide structure, organization, and a way to represent data and behavior together
- **Basic Data Structures**:
  - **Stacks**: A stack is a Last-In, First-Out (LIFO) structure, where elements are added and removed from the same end. Operations include push (add) and pop (remove)
  - **Queues**: A queue is a First-In, First-Out (FIFO) structure, where elements are added at one end and removed from the other. Operations include enqueue (add) and dequeue (remove)
  - **Hash Tables**: A hash table is a data structure that stores key-value pairs, enabling fast data retrieval by mapping keys to values through a hashing function

OOP and Basic Data Structures

2. Why does it matter?

- **Organization and Modularity**: OOP allows for a more structured codebase where data and functions are organized within classes. This modularity makes code easier to maintain, debug, and extend

- **Efficient Data Handling**: Stacks and queues offer efficient ways to manage ordered data, which is essential in applications like backtracking algorithms (for stacks) or scheduling tasks (for queues)

- **Fast Data Retrieval**: Hash tables allow for nearly constant-time complexity (O(1)) for lookups, making them highly efficient for cases where quick access to data is crucial, such as dictionaries, caching, and managing unique identifiers

OOP and Basic Data Structures

3. Where can you expect to see this?

- **Application Development**: You'll see OOP used in software applications to create models for users, products, transactions, etc. Data structures like stacks and queues are also used to manage tasks, events, or command histories

- **Web Development and APIs**: OOP principles help design classes that encapsulate API functionalities. Hash tables (or dictionaries in Python) are common in storing configuration settings, managing user sessions, or caching responses

- **Algorithm Design and Data Processing**: Stacks, queues, and hash tables are fundamental in algorithm design. For example, stacks are used in depth-first search (DFS) algorithms, while hash tables help manage unique data items

OOP and Basic Data Structures

What is it?

- **OOP (Object-Oriented Programming) Methods**: OOP is a programming paradigm that allows you to model real-world entities as objects. In Python, OOP concepts such as classes, methods, inheritance, and encapsulation enable the creation of modular and reusable code. These OOP methods provide structure, organization, and a way to represent data and behavior together.
- **Basic Data Structures**:
  - **Stacks**: A stack is a Last-In, First-Out (LIFO) structure, where elements are added and removed from the same end. Operations include push (add) and pop (remove).
  - **Queues**: A queue is a First-In, First-Out (FIFO) structure, where elements are added at one end and removed from the other. Operations include enqueue (add) and dequeue (remove).
  - **Hash Tables**: A hash table is a data structure that stores key-value pairs, enabling fast data retrieval by mapping keys to values through a hashing function.

A stack follows the Last-In, First-Out (LIFO) principle. It can be implemented using a list with append (for pushing items) and pop (for removing items)

Use Cases:

- **Backtracking**: Stacks are useful in algorithms that involve exploring all possibilities and then backtracking, such as depth-first search (DFS) in graphs or mazes
- **Undo/Redo Functionality**: Applications like text editors use stacks to keep track of changes, allowing users to undo or redo actions
- **Expression Parsing**: Stacks help evaluate mathematical expressions, especially when handling parentheses or converting infix expressions to postfix

```python
class Stack:
    def __init__(self):
        self.stack = []

    def push(self, item):
        self.stack.append(item)  # Adds an item to the top

    def pop(self):
        if not self.is_empty():
            return self.stack.pop()  # Removes the item from the top
        else:
            return None

    def peek(self):
        if not self.is_empty():
            return self.stack[-1]  # Returns the top item without removing it
        else:
            return None

    def is_empty(self):
        return len(self.stack) == 0
```

Use append and pop for adding and removing items, respectively.

A queue follows the First-In, First-Out (FIFO) principle. It can be implemented using collections.deque for efficient enqueue and dequeue operations.

Use Cases:

- **Task Scheduling**: Queues are used to manage tasks that need to be processed in a specific order, such as jobs waiting to be printed or CPU scheduling in operating systems
- **Breadth-First Search (BFS)**: BFS in graphs and trees uses a queue to explore nodes layer by layer
- **Asynchronous Data Processing**: Queues are useful in message-based systems and asynchronous processing, where data is processed in the order it arrives.

```python
from collections import deque

class Queue:
    def __init__(self):
        self.queue = deque()

    def enqueue(self, item):
        self.queue.append(item)  # Adds an item to the end of the queue

    def dequeue(self):
        if not self.is_empty():
            return self.queue.popleft()  # Removes the item from the front of the
        else:
            return None

    def is_empty(self):
        return len(self.queue) == 0

    def peek(self):
        if not self.is_empty():
            return self.queue[0]  # Returns the front item without removing it
        else:
            return None
```

Use deque with append for enqueue and popleft for dequeue.

A hash table stores key-value pairs and uses a hash function to map keys to values. In Python, you can use a dictionary to implement a hash table

Use Cases:

- **Fast Data Retrieval**: Hash tables enable fast lookups, making them ideal for implementing dictionaries, caches, and databases
- **Counting and Frequency Tracking**: Hash tables are often used to count occurrences of items, such as words in a document or elements in a dataset
- **Implementing Sets**: Since hash tables store unique keys, they're often used to implement sets, which require that each element is unique

```python
class HashTable:
    def __init__(self):
        self.table = {}

    def insert(self, key, value):
        self.table[key] = value  # Inserts a key-value pair

    def get(self, key):
        return self.table.get(key, None)  # Retrieves the value for a key, or None

    def delete(self, key):
        if key in self.table:
            del self.table[key]  # Removes the key-value pair

    def contains(self, key):
        return key in self.table  # Checks if the key exists in the hash table
```

Use a dictionary to store key-value pairs with insert, get, and delete methods.

Agenda

- Midterm Logistics
- Reviewing resources
- Unix
- Python review
  - Python Virtual Machine (PVM)
  - OOP Methods and Basic Data Structures (Stacks, Queues, and Hash Tables)
  - Exceptions
  - Iterators
  - Decorators
  - Recursion
- Final Tips

Exceptions

1. What is it?

- An exception is an event that disrupts the normal flow of a program. In Python, exceptions are raised when the interpreter encounters an error that prevents it from continuing.

- Common exceptions include TypeError, ValueError, IndexError, and FileNotFoundError, among many others

- Python provides a way to handle these exceptions gracefully through try, except, else, and finally blocks, allowing developers to manage unexpected events and control program flow even when errors occur

Exceptions

2. Why does it matter?

- **Robustness and Error Handling**: Exceptions are crucial for creating robust programs that can handle errors without crashing unexpectedly. By anticipating possible errors, developers can ensure their code behaves predictably and provide meaningful feedback to the user

- **Debugging and Troubleshooting**: Exceptions make it easier to identify where issues arise in code. Python provides detailed error messages and stack traces that help developers locate the source of the problem, facilitating faster debugging

- **Maintaining Program Flow**: Exception handling allows a program to manage errors while maintaining continuity, rather than terminating abruptly. For example, if an error occurs while reading a file, the program can catch the exception, log the issue, and continue with other tasks

Exceptions

3. Where can you expect to see this?

- **File I/O**: When working with files, exceptions are often raised if a file is missing, inaccessible, or corrupted. For instance, trying to open a non-existent file raises a FileNotFoundError

- **User Input Validation**: Programs that require user input use exceptions to handle invalid inputs. For example, a program might raise a ValueError if a user enters non-numeric data where a number is expected

Exceptions

3. Where can you expect to see this (continued)?

- **API and Network Requests**: When connecting to external servers or APIs, exceptions handle cases like timeouts, connectivity issues, or unexpected responses. This allows the program to retry the connection or gracefully alert the user if the resource is unavailable

- **Algorithmic and Mathematical Operations**: When performing calculations, exceptions can handle scenarios like division by zero (ZeroDivisionError) or out-of-bound index access (IndexError), helping maintain program stability

Exception 1 example:

Here, we define several functions, which may or may not raise an exception when executed.

For each function, indicate if an exception is raised **always**, **sometimes**, or **never**.

For **always** and **sometimes**, indicate which exception is raised and, for sometimes, specify the conditions under which the exception occurs.

```
def g1():
    import nonexistent_module

def g2():
    lst = [1, 2, 3]
    return lst[5]

def g3():
    with open("data.txt", "w") as f:
        f.write(42)

def g4():
    return b"hello".decode("utf-8")

def g5():
    return 10 / 0
```

How to approach this?

When approaching this type of exception identification problem, follow these steps:

1. **Read Each Function Carefully**: Analyze the code in each function to understand what it's doing and identify potential issues
2. **Consider Possible Errors**: Think about common Python exceptions (e.g., ImportError, IndexError, TypeError) and whether the function might trigger any of them
3. **Determine Exception Frequency**:
   - Decide if the exception will be raised **always**, **sometimes**, or **never**
   - **Always**: The exception occurs every time the function is called
   - **Sometimes**: The exception depends on external factors or specific conditions
   - **Never**: No exception will occur under typical circumstances
4. **Identify the Conditions for "Sometimes"**: If an exception is raised sometimes, specify the exact conditions under which it occurs (e.g., missing files, invalid module names)
5. **Test Your Understanding**: If possible, think through test cases or mentally simulate running the code to check if your predictions make sense
6. **Use Python Documentation**: Refer to the documentation for unfamiliar functions or methods to understand their typical behavior and associated exceptions

Answer Key:

g1()

- **Sometimes**: Raises ImportError when nonexistent_module is not found in the Python environment. This happens if the module is not installed

g2()

- **Always**: Raises IndexError because the list lst has only three elements, and accessing index 5 is out of bounds

g3()

- **Always**: Raises TypeError because f.write(42) expects a string argument, but an integer (42) is provided

g4()

- **Never**: The decode("utf-8") method on b"hello" (a valid byte sequence) executes without any issues

g5()

- **Always**: Raises ZeroDivisionError because division by zero is undefined in Python

Exception 2 example:

Here, we define several functions, which may or may not raise an exception when executed.

For each function, indicate if an exception is raised **always**, **sometimes**, or **never**.

For **always** and **sometimes**, indicate which exception is raised and, for sometimes, specify the conditions under which the exception occurs.

```
def h1():
    result = int("abc")

def h2():
    my_dict = {'x': 10}
    return my_dict['y']

def h3():
    numbers = [1, 2, 3]
    return sum(numbers) / len(numbers)

def h4(filepath):
    with open(filepath, "r") as f:
        data = f.read()
    return data

def h5():
    return "hello".index("z")
```

Answer Key:

h1()

- **Always**: Raises ValueError because converting the string "abc" to an integer is invalid

h2()

- **Always**: Raises KeyError because the dictionary my_dict does not contain the key 'y'

h3()

- **Never**: This function calculates the average of a list of numbers, which will work correctly as long as the list is non-empty and contains numeric values. Here, it only has valid numbers, so no exception is raised

h4(filepath)

- **Sometimes**: Raises FileNotFoundError if the specified filepath does not exist or is inaccessible. This depends on whether the file is present and readable

h5()

- **Always**: Raises ValueError because "hello" does not contain the substring "z", so calling index("z") on it fails

Agenda

- Midterm Logistics
- Reviewing resources
- Unix
- Python review
  - Python Virtual Machine (PVM)
  - OOP Methods and Basic Data Structures (Stacks, Queues, and Hash Tables)
  - Exceptions
  - Iterators
  - Decorators
  - Recursion
- Final Tips

Iterators

1. What is it?

- An **iterator** is an object that allows sequential access to elements in a collection (like a list, tuple, or dictionary) without exposing the underlying data structure

- In Python, an iterator is an object that implements the __iter__() and __next__() methods

- The __iter__() method returns the iterator object itself, and __next__() returns the next element in the sequence. When there are no more items to return, __next__() raises a StopIteration exception, signaling the end of the iteration

Iterators

2. Why does it matter?

- **Memory Efficiency**: Iterators allow you to work with data collections without loading the entire collection into memory. This is particularly useful for handling large datasets or streams of data, as iterators fetch items one at a time only when needed
- **Lazy Evaluation**: Iterators enable lazy evaluation, meaning values are generated on-the-fly and only computed when accessed. This can greatly improve performance and reduce memory usage for operations on large or infinite sequences
- **Enhanced Code Readability and Maintainability**: Using iterators often leads to cleaner and more readable code. They are foundational for Python's loop constructs (like for loops), making it easier to write concise and expressive code
- **Custom Iteration Patterns**: Python allows developers to create custom iterator classes, which can be useful for implementing specific traversal or access patterns, such as iterating over data in a custom order or implementing a filter

Iterators

3. Where can you expect to see this?

- **For Loops**: Every time you use a for loop in Python, you're working with an iterator. Python automatically converts iterables (such as lists, tuples, and dictionaries) into iterators for loop processing
- **Data Streams and File I/O**: Iterators are common in handling data streams and file I/O. For example, using file_object with for line in file_object creates an iterator that reads each line of a file one at a time, conserving memory
- **Generators**: Python's generator functions (using the yield keyword) create iterators. Generators are a convenient way to build iterators without the need to implement __iter__() and __next__() directly, and they're commonly used for tasks requiring lazy evaluation
- **Custom Classes**: By defining __iter__() and __next__() in a custom class, developers can implement tailored iteration patterns for specific applications, like traversing complex data structures or simulating a sequence of events

Iterators

3. Where can you expect to see this?

- **For Loops**: Every time you use a for loop in Python, you're working with an iterator. Python automatically converts iterables (such as lists, tuples, and dictionaries) into iterators for loop processing
- **Data Streams and File I/O**: Iterators are common in handling data streams and file I/O. For example, using file_object with for line in file_object creates an iterator that reads each line of a file one at a time, conserving memory
- **Generators**: Python's generator functions (using the yield keyword) create iterators. Generators are a convenient way to build iterators without the need to implement __iter__() and __next__() directly, and they're commonly used for tasks requiring lazy evaluation
- **Custom Classes**: By defining __iter__() and __next__() in a custom class, developers can implement tailored iteration patterns for specific applications, like traversing complex data structures or simulating a sequence of events

Example

Define a Python function iterator_example() that returns an iterator for a list of numbers [1, 2, 3, 4] and then iterates through it using the next() function

The function should retrieve and return each item in the list one at a time using next()

For each call to next(), if there are no more items, the function should handle the StopIteration exception and return "End of Iterator" instead

Write the function iterator_example() and provide the output of each call to next() on the iterator

Expected output:

```
1
2
3
4
End of Iterator
```

How to approach?

- To approach this problem, first understand that an iterator allows sequential access to elements in a collection
- Begin by creating an iterator from the given list using iter()
- Then, use next() to retrieve each element one-by-one
- Anticipate the StopIteration exception, which signals the end of the iterator, and handle it with a try-except block to print "End of Iterator" when no more items are available
- This structure ensures that all elements are accessed, and the end of the iteration is managed gracefully

Answer → The function iterator_example() that meets these requirements is:

Iterator Creation:

- numbers = [1, 2, 3, 4]: A list is created with four elements
- iter_numbers = iter(numbers): The iter() function converts the list into an iterator object, allowing us to traverse the list one item at a time

Using next() to Retrieve Items:

- The function uses next(iter_numbers) to retrieve each item from the iterator, moving to the next item with each call
- For the first four calls to next(), it returns each item in the list in order: 1, 2, 3, and 4

Handling StopIteration:

- After the last item is retrieved, a subsequent call to next() raises a StopIteration exception
- The function catches this exception in an except block, printing "End of Iterator" to indicate that there are no more items

```
def iterator_example():
    numbers = [1, 2, 3, 4]
    iter_numbers = iter(numbers)  # Create an iterator from the list

    # Fetch and print each item one at a time
    try:
        print(next(iter_numbers))  # Output: 1
        print(next(iter_numbers))  # Output: 2
        print(next(iter_numbers))  # Output: 3
        print(next(iter_numbers))  # Output: 4
        print(next(iter_numbers))  # Raises StopIteration
    except StopIteration:
        print("End of Iterator")

iterator_example()
```

Agenda

- Midterm Logistics
- Reviewing resources
- Unix
- Python review
  - Python Virtual Machine (PVM)
  - OOP Methods and Basic Data Structures (Stacks, Queues, and Hash Tables)
  - Exceptions
  - Iterators
  - Decorators
  - Recursion
- Final Tips

Decorators

1. What is it?

- A **decorator** is a higher-order function that "wraps" another function or method to modify or extend its behavior. In Python, decorators are functions that take another function as an argument, add some functionality, and return a new function or the original function with added behavior

- Decorators are applied to functions or methods using the @decorator_name syntax placed above the function definition. This syntax is shorthand for passing the function to the decorator

Decorators

2. Why does it matter?

- **Code Reusability**: Decorators allow you to add functionality to multiple functions without repeating code. This makes code more modular and reusable
- **Separation of Concerns**: By using decorators, you can separate core function logic from auxiliary functionality (like logging, authentication, or caching). This keeps your code clean and focused
- **Readability and Maintenance**: Decorators provide a concise, readable way to add behavior to functions, making the code easier to understand and maintain. For example, adding @log or @authenticate at the top of a function definition clearly indicates the function's added behavior without cluttering its main logic
- **Enhanced Functionality**: Decorators enable the implementation of cross-cutting concerns like timing, debugging, access control, and more, which are essential for building robust applications

Decorators

3. Where can you expect to see this?

- **Logging and Debugging**: Decorators are often used to add logging to functions, allowing developers to track function calls and outputs. For example, a @log decorator might record when a function is called and what values it returns

- **Authentication and Authorization**: In web applications, decorators are frequently used to control access. For example, an @authenticate decorator could check if a user is logged in before allowing access to a function

- **Caching and Memoization**: Decorators can add caching to functions, storing the results of expensive calculations and returning the cached result on subsequent calls. This is common in performance optimization, where results of certain function calls are saved for reuse

Decorators

3. Where can you expect to see this (continued)?

- **Timing and Performance Monitoring**: A @timer decorator can be used to measure the time a function takes to execute, which is helpful for performance monitoring and optimization.

- **Class and Method Enhancements**: In classes, decorators like @staticmethod, @classmethod, and @property are built-in Python decorators that modify method behavior, such as making a method callable on the class itself rather than an instance.

Example: Decorator implementation

Problem Prompt:

Create a decorator named timed that measures the execution time of a function and prints out the time taken each time the function is called. The decorator should:

1. Print the function's name, arguments, and the time taken in seconds to complete the call.
2. Ensure that the decorated function behaves like the original function, preserving its name and docstring.

Then, apply the timed decorator to a function compute_factorial that computes the factorial of a given number n. Test the decorator by calling compute_factorial(5) and compute_factorial(10).

Example Output:

compute_factorial(5) took 0.0001 seconds

compute_factorial(10) took 0.0001 seconds

How could this be approached?

When approaching a problem that involves creating a decorator like timed, here's a structured approach and the key considerations someone should keep in mind:

1. **Clarify the Decorator's Purpose**: Identify the decorator's function—e.g., timing, logging, caching—and what it should output
2. **Set Up Structure**: Create an outer function that takes the original function as an argument, then define an inner wrapper function to add new behavior
3. **Plan Core Logic**: Place the necessary logic in the wrapper (e.g., capturing start and end times for timing)
4. **Preserve Metadata**: Use @wraps to retain the original function's name, docstring, and attributes
5. **Implement and Test**: Write the code, apply the decorator to a sample function, and test with different inputs
6. **Ensure Transparency**: Verify that the decorator doesn't alter the function's original behavior or output
7. **Check Reusability**: Confirm that the decorator works well with various functions and argument types
8. **Consider Edge Cases**: Test scenarios with different input sizes and types to ensure reliability

Detailed answer:

The timed decorator calculates and prints the execution time of a function by capturing start and end times with time.time()

It computes the elapsed time and displays it alongside te function's name and arguments

When applied to compute_factorial, the decorator logs the time taken for each call, which is particularly helpful for monitoring performance with recursive functions

Using @wraps(f) ensures the decorated function keeps its original name and docstring, maintaining its metadata

```python
import time
from functools import wraps

def timed(f):
    """Decorator that prints the execution time of a function."""
    @wraps(f)
    def wrapped(*args, **kwargs):
        start_time = time.time()       # Record start time
        result = f(*args, **kwargs)    # Call the function
        end_time = time.time()         # Record end time
        elapsed_time = end_time - start_time
        print(f"{f.__name__}{args} took {elapsed_time:.4f} seconds")
        return result
    return wrapped

@timed
def compute_factorial(n):
    """Compute factorial of n."""
    if n <= 1:
        return 1
    else:
        return n * compute_factorial(n - 1)

# Test the decorator
compute_factorial(5)
compute_factorial(10)
```

- **import time:** Imports Python’s built-in time module, which provides various time-related functions. Here, we’ll use it to measure the start and end times of the function's execution
- **from functools import wraps:** Imports the wraps function from the functools module. wraps is used in decorators to preserve the original function’s metadata (like its name and docstring) when wrapping it with additional functionality.

- **def timed(f):** Defines the decorator function timed, which takes a single argument f. This argument f represents the function that the decorator will wrap
- **"""Decorator that prints the execution time of a function.""":** This is a docstring describing the purpose of the timed decorator. It explains that timed will print how long a function takes to execute.
- **@wraps(f):** This line is a decorator for the wrapped function inside timed
- **def wrapped(*args, **kwargs):** Defines the inner function wrapped, which takes *args (positional arguments) and **kwargs (keyword arguments)

- **start_time = time.time():** Captures the current time in seconds since the epoch (a fixed point in time used as a reference)
- **result = f(*args, **kwargs):** Calls the original function f with the provided arguments *args and **kwargs
- **end_time = time.time():** Captures the current time again, marking the end of the function execution
- **elapsed_time = end_time - start_time:** Calculates the time taken to execute the function f by subtracting start_time from end_time
- **print(f"{f.__name__}{args} took {elapsed_time:.4f} seconds"): This** line formats and prints a message showing the function's name (f.__name__), its arguments (args), and the time taken to execute (elapsed_time). The :.4f format specifier rounds the elapsed time to four decimal places for readability
- **return result:** We should know what this does
- **return wrapped:** The outer timed function returns the wrapped function, which is now the decorated version of f. When @timed is used on a function, it replaces that function with wrapped, adding the timing functionality

- **start_time = time.time():** Captures the current time in seconds since the epoch (a fixed point in time used as a reference)
- **result = f(*args, **kwargs):** Calls the original function f with the provided arguments *args and **kwargs
- **end_time = time.time():** Captures the current time again, marking the end of the function execution
- **elapsed_time = end_time - start_time:** Calculates the time taken to execute the function f by subtracting start_time from end_time
- **print(f"{f.__name__}{args} took {elapsed_time:.4f} seconds"): This** line formats and prints a message showing the function's name (f.__name__), its arguments (args), and the time taken to execute (elapsed_time). The :.4f format specifier rounds the elapsed time to four decimal places for readability
- **return result:** We should know what this does
- **return wrapped:** The outer timed function returns the wrapped function, which is now the decorated version of f. When @timed is used on a function, it replaces that function with wrapped, adding the timing functionality

- **@timed:** This line applies the timed decorator to compute_factorial. It is equivalent to calling compute_factorial = timed(compute_factorial). From now on, whenever compute_factorial is called, it will run with the timing functionality
- **def compute_factorial(n):** Defines a function to compute the factorial of a given integer n.
- **"""Compute factorial of n.""":** The docstring describes the function's purpose.
- The **if and else statements calculate the factorial.** If n <= 1, the function returns 1 (base case). Otherwise, it recursively calls itself with n - 1, eventually reaching the base case.

Summary:

The **timed** decorator measures and prints the execution time of **compute_factorial**, allowing us to see how long it takes to compute factorials of different numbers. Each component in this code contributes to adding a timing layer around **compute_factorial**, making it easy to track performance without altering the function's core logic

Agenda

- Midterm Logistics
- Reviewing resources
- Unix
- Python review
  - Python Virtual Machine (PVM)
  - OOP Methods and Basic Data Structures (Stacks, Queues, and Hash Tables)
  - Exceptions
  - Iterators
  - Decorators
  - Recursion
- Final Tips

Recursion [Same Slides from Last Review]

Recursion is a method of solving problems where a function calls itself as a part of its computation. The key idea is to break a problem down into smaller, more manageable parts until you reach a base case, which stops the recursive calls.

Key Concepts of Recursion

- **Definition**:
  - Recursion occurs when a function calls itself during its execution.
- **Base Case**:
  - Every recursive function needs a base case that stops the recursion.
  - Without a base case, the recursion would go on indefinitely, leading to a "stack overflow" error.
- **Recursive Case**:
  - The part of the function where the function calls itself with a smaller or simpler input, moving the problem closer to the base case.

Recursion [Same Slides from Last Review]

Why Use Recursion?:

- **Breaks down complex problems**: Recursion is especially useful for problems that can be broken into similar subproblems, such as dividing a list, solving mathematical sequences, or working with tree-like structures.
- **Elegance and simplicity**: Recursive solutions can often be more elegant and easier to read than iterative solutions (loops).

Common Examples:

- **Factorial calculation**: $n! = n \times (n-1) \times (n-2) \times \cdots \times 1$
- **Fibonacci sequence**: $F(n) = F(n-1) + F(n-2)$
- **Navigating nested structures**: Recursion is great for navigating trees, directories, and nested lists.

Recursion [Same Slides from Last Review]

Key Steps in Writing Recursive Functions:

1. Identify the base case:
   - What is the simplest version of the problem where you know the answer immediately?
   - Example: For a factorial function, the base case is **n == 0**, where the result is **1**
2. Identify the recursive case:
   - How can you break the problem into a smaller version of itself?
   - Example: For factorials, you can reduce the problem to **n * factorial(n-1)**
3. Ensure Progress Toward the Base Case:
   - Every recursive call should bring the input closer to the base case
   - If there is no progress toward the base case, the recursion will continue indefinitely

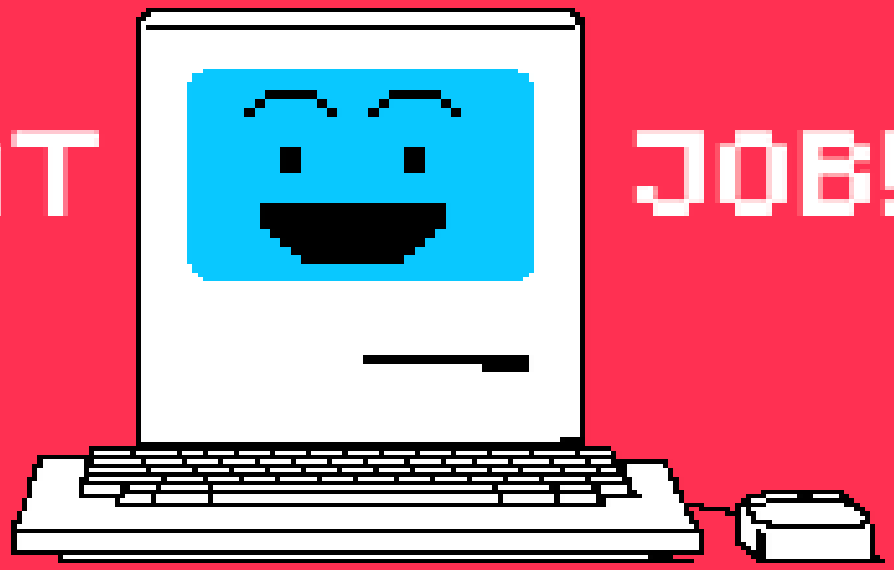
Agenda

- Midterm Logistics
- Reviewing resources
- Unix
- Python review
  - Python Background
  - Procedure Examples
  - Regular Expressions
  - Python Expressions
  - Recursion
  - Comprehensions
  - Object-oriented Programming
- Final Tips

Final Tips

- Start studying early
- Review your last two Psets (especially 4)
- Complete the Unix Tutorial, up to Three
- Post on Ed Discussions for clarifying answers

YOU'RE DOING A GREAT



JOB!