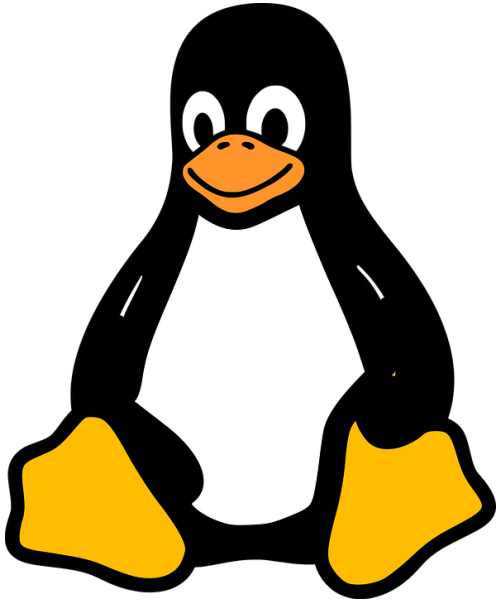


CS 201 Midterm 1 Review
Fall 2024

Agenda

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

Agenda

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

Logistics - Midterm 1

Thursday October 10 at 7pm in ML 211

2-hour hand written exam

No computers, notes, or books

Agenda

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

Resources available to you as you prepare

- The Basics → CPSC 200 – Course Site; This include the lecture notes

- Professor Slade's YouTube Video Recommendations → Python Tutorials from Socratica
- Google's Python Class Intro → Overview
- "THE" Python Guide

- Practice material
  - Practice Exam / Solutions
  - UNIX Transcript / Solutions → python3 /c/cs201/www/unixtutorial.py
    - ssh into the Zoo; then in your home folder, type the following command:

- Ed Discussion board
- Reviewing problem sets and each other (it never hurts to make study groups)!

Agenda

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

How can I get better at UNIX?

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) Practice typing commands on the Zoo. This is the best way to learn.

General tips:

- Be familiar with the *output* of each command (important in context of the transcript)

A few UNIX scenarios

scenario 1

```
[jlv34@peacock midterm1_prep]$ pwd
/home/accts/jlv34/midterm1_prep
[jlv34@peacock midterm1_prep]$ ls -l
total 12
drwxrwxr-x 2 jlv34 jlv34 4096 Feb 22 20:16 test1
drwxrwxr-x 2 jlv34 jlv34 4096 Feb 22 20:16 test2
drwxrwxr-x 2 jlv34 jlv34 4096 Feb 22 20:16 test3
[jlv34@peacock midterm1_prep]$ [redacted] cd test2
[jlv34@peacock test2]$ pwd
/home/accts/jlv34/midterm1_prep/test2
[jlv34@peacock test2]$ [redacted] cd..
[jlv34@peacock midterm1_prep]$ pwd
/home/accts/jlv34/midterm1_prep
```

cd.. 3 Change Parent Directory

scenario 1

```
[jlv34@peacock midterm1_prep]$ pwd
/home/accts/jlv34/midterm1_prep
[jlv34@peacock midterm1_prep]$ ls -l
total 12
drwxrwxr-x 2 jlv34 jlv34 4096 Feb 22 20:16 test1
drwxrwxr-x 2 jlv34 jlv34 4096 Feb 22 20:16 test2
drwxrwxr-x 2 jlv34 jlv34 4096 Feb 22 20:16 test3
[jlv34@peacock midterm1_prep]$ cd test2
[jlv34@peacock test2]$ pwd
/home/accts/jlv34/midterm1_prep/test2
[jlv34@peacock test2]$ cd ..
[jlv34@peacock midterm1_prep]$ pwd
/home/accts/jlv34/midterm1_prep
```

scenario 2

```
[[jlv34@peacock test2]$ ls -l
total 0
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:21 file1
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:21 file2
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:21 file3
[[jlv34@peacock test2]$ [redacted] touch file4
[[jlv34@peacock test2]$ ls -l
total 0
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:21 file1
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:21 file2
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:21 file3
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:21 file4
```

touch file4 3 Create or update a file's timestamp

scenario 2

```
[[jlv34@peacock test2]$ ls -l
total 0
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:21 file1
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:21 file2
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:21 file3
[[jlv34@peacock test2]$ touch file4
[[jlv34@peacock test2]$ ls -l
total 0
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:21 file1
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:21 file2
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:21 file3
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:21 file4
```

ls -l ⇒ list files in long format

scenario 3

```
[jlv34@peacock test2]$ ls -l
total 0
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:25 file1
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:25 file2
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:25 file3
[jlv34@peacock test2]$ [redacted]
[jlv34@peacock test2]$ ls -l
total 0
-rw-rw-r-- 1 jlv34 jlv34  0 Feb 22 20:25 file1
-rw-rw-r-- 1 jlv34 jlv34  0 Feb 22 20:25 file2
-rw-rw-r-- 1 jlv34 jlv34  0 Feb 22 20:25 file3
-rw-rw-r-- 1 jlv34 jlv34 29 Feb 22 21:06 file4
[jlv34@peacock test2]$ cat file4
Fri Feb 22 21:06:18 EST 2019
[jlv34@peacock test2]$
```

date > file4 ⇒ Write the current date and time to file4

scenario 3

```
[jlv34@peacock test2]$ ls -l
total 0
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:25 file1
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:25 file2
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:25 file3
[jlv34@peacock test2]$ date > file4
[jlv34@peacock test2]$ ls -l
total 0
-rw-rw-r-- 1 jlv34 jlv34  0 Feb 22 20:25 file1
-rw-rw-r-- 1 jlv34 jlv34  0 Feb 22 20:25 file2
-rw-rw-r-- 1 jlv34 jlv34  0 Feb 22 20:25 file3
-rw-rw-r-- 1 jlv34 jlv34 29 Feb 22 21:06 file4
[jlv34@peacock test2]$ cat file4
Fri Feb 22 21:06:18 EST 2019
[jlv34@peacock test2]$
```

cat file4 3 concatenate and display the content of file4

scenario 4

```
[jlv34@peacock test2]$ ls -l
total 0
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:21 file1
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:21 file2
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:21 file3
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:21 file4
[jlv34@peacock test2]$ [redacted]
[jlv34@peacock test2]$ ls
[jlv34@peacock test2]$
```

=> Nothing here

rm * => Remove all files

scenario 4

```
[jlv34@peacock test2]$ ls -l
total 0
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:21 file1
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:21 file2
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:21 file3
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:21 file4
[jlv34@peacock test2]$ rm *
[jlv34@peacock test2]$ ls
[jlv34@peacock test2]$
```

scenario 5

```
[jlv34@peacock test1]$ ls -l
total 0
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:33 file.pdf
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:33 file.txt
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:33 hello.pdf
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:33 hello.txt
[jlv34@peacock test1]$ [redacted] rm file.*
[jlv34@peacock test1]$ ls -l
total 0
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:33 hello.pdf
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:33 hello.txt
[jlv34@peacock test1]$
```

rm file.* } remove all files with the name 'file' & any extension

scenario 5

```
[jlv34@peacock test1]$ ls -l
total 0
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:33 file.pdf
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:33 file.txt
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:33 hello.pdf
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:33 hello.txt
[jlv34@peacock test1]$ rm file.*
[jlv34@peacock test1]$ ls -l
total 0
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:33 hello.pdf
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:33 hello.txt
[jlv34@peacock test1]$
```

scenario 6

```
[jlv34@peacock test1]$ ls -l
total 0
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:35 file.pdf
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:35 file.txt
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:33 hello.pdf
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:33 hello.txt
[jlv34@peacock test1]$ [redacted] rm *.pdf
[jlv34@peacock test1]$ ls -l
total 0
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:35 file.txt
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:33 hello.txt
[jlv34@peacock test1]$
```

rm *.pdf ⇶ Remove all files with a pdf extension

scenario 6

```
[jlv34@peacock test1]$ ls -l
total 0
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:35 file.pdf
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:35 file.txt
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:33 hello.pdf
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:33 hello.txt
[jlv34@peacock test1]$ rm *.pdf
[jlv34@peacock test1]$ ls -l
total 0
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:35 file.txt
-rw-rw-r-- 1 jlv34 jlv34 0 Feb 22 20:33 hello.txt
[jlv34@peacock test1]$
```

scenario 7

```
[jlv34@peacock test1]$ ls -l
total 0
-rw-rw-r-- 1 jlv34 jlv34 13 Feb 22 20:40 foo
[jlv34@peacock test1]$ cat foo
hello, world
[jlv34@peacock test1]$ [redacted]
[jlv34@peacock test1]$ ls -l
total 0
-rw-rw-r-- 1 jlv34 jlv34 13 Feb 22 21:08 bar
-rw-rw-r-- 1 jlv34 jlv34 13 Feb 22 20:40 foo
[jlv34@peacock test1]$ cat bar
hello, world
[jlv34@peacock test1]$
```

cp foo bar

cp foo bar ⇒ Copies the contents of foo to a new file named bar

scenario 7

```
[jlv34@peacock test1]$ ls -l
total 0
-rw-rw-r-- 1 jlv34 jlv34 13 Feb 22 20:40 foo
[jlv34@peacock test1]$ cat foo
hello, world
[jlv34@peacock test1]$ cp foo bar
[jlv34@peacock test1]$ ls -l
total 0
-rw-rw-r-- 1 jlv34 jlv34 13 Feb 22 21:08 bar
-rw-rw-r-- 1 jlv34 jlv34 13 Feb 22 20:40 foo
[jlv34@peacock test1]$ cat bar
hello, world
[jlv34@peacock test1]$
```

scenario 8

```
[[jlv34@peacock test1]$ ls -l
total 0
-rw-rw-r-- 1 jlv34 jlv34 13 Feb 22 20:40 foo
[[jlv34@peacock test1]$ cat foo
hello, world
[[jlv34@peacock test1]$ [redacted]
[[jlv34@peacock test1]$ ls -l
total 0
-rw-rw-r-- 1 jlv34 jlv34 13 Feb 22 20:40 bar
[[jlv34@peacock test1]$ cat bar
hello, world
[jlv34@peacock test1]$
```

mv foo bar

mv foo bar ⇒ move or rename a file → foo is renamed to bar

scenario 8

```
[jlv34@peacock test1]$ ls -l
total 0
-rw-rw-r-- 1 jlv34 jlv34 13 Feb 22 20:40 foo
[jlv34@peacock test1]$ cat foo
hello, world
[jlv34@peacock test1]$ mv foo bar
[jlv34@peacock test1]$ ls -l
total 0
-rw-rw-r-- 1 jlv34 jlv34 13 Feb 22 20:40 bar
[jlv34@peacock test1]$ cat bar
hello, world
[jlv34@peacock test1]$
```

wc : word count

(Number of Lines) (Number of Words) (Number of Characters) (File Name)

scenario 9

```
[jlv34@peacock test4]$ ls -l
total 1200
-rw-rw-r-- 1 jlv34 jlv34 1220147 Feb 22 20:53 entire_moby_dick_book
[jlv34@peacock test4]$ wc entire_moby_dick_book
  22930  212044 1220147 entire_moby_dick_book
[jlv34@peacock test4]$ [redacted]
MOBY DICK; OR THE WHALE

by Herman Melville

ETYMOLOGY.

(Supplied by a Late Consumptive Usher to a Grammar School)

The pale Usher--threadbare in coat, heart, body, and brain; I see him
now.  He was ever dusting his old lexicons and grammars, with a queer
[jlv34@peacock test4]$
```

head entire_moby_dick_book : Display first 10 lines of the file

scenario 9

```
[jlv34@peacock test4]$ ls -l
total 1200
-rw-rw-r-- 1 jlv34 jlv34 1220147 Feb 22 20:53 entire_moby_dick_book
[jlv34@peacock test4]$ wc entire_moby_dick_book
  22930  212044 1220147 entire_moby_dick_book
[jlv34@peacock test4]$ head entire_moby_dick_book
MOBY DICK; OR THE WHALE

by Herman Melville

ETYMOLOGY.

(Supplied by a Late Consumptive Usher to a Grammar School)

The pale Usher--threadbare in coat, heart, body, and brain; I see him
now.  He was ever dusting his old lexicons and grammars, with a queer
[jlv34@peacock test4]$
```

scenario 10

```
[jlv34@peacock test4]$ pwd
/home/accts/jlv34/midterm1_prep/test4
[jlv34@peacock test4]$ ls -l
total 1200
-rw-rw-r-- 1 jlv34 jlv34 1220147 Feb 22 20:53 entire_moby_dick_book
[jlv34@peacock test4]$ [redacted] mkdir test5
[jlv34@peacock test4]$ [redacted] cd test5
[jlv34@peacock test5]$ pwd
/home/accts/jlv34/midterm1_prep/test4/test5
[jlv34@peacock test5]$
```

scenario 10

```
[jlv34@peacock test4]$ pwd
/home/accts/jlv34/midterm1_prep/test4
[jlv34@peacock test4]$ ls -l
total 1200
-rw-rw-r-- 1 jlv34 jlv34 1220147 Feb 22 20:53 entire_moby_dick_book
[jlv34@peacock test4]$ mkdir test5
[jlv34@peacock test4]$ cd test5
[jlv34@peacock test5]$ pwd
/home/accts/jlv34/midterm1_prep/test4/test5
[jlv34@peacock test5]$
```

scenario 11

```
[jlv34@zebra test1]$ pwd
/home/accts/jlv34/midterm1_prep/test1
[jlv34@zebra test1]$ ls -l
total 4
drwxrwxr-x 2 jlv34 jlv34 4096 Feb 23 17:16 test2
[jlv34@zebra test1]$ cd test2
[jlv34@zebra test2]$ echo "hello world" > file
[jlv34@zebra test2]$ cd ..
[jlv34@zebra test1]$ [redacted]
[jlv34@zebra test1]$ ls -l
total 4
-rw-rw-r-- 1 jlv34 jlv34   12 Feb 23 17:17 file
drwxrwxr-x 2 jlv34 jlv34 4096 Feb 23 17:17 test2
[jlv34@zebra test1]$ cat file
hello world
[jlv34@zebra test1]$
```

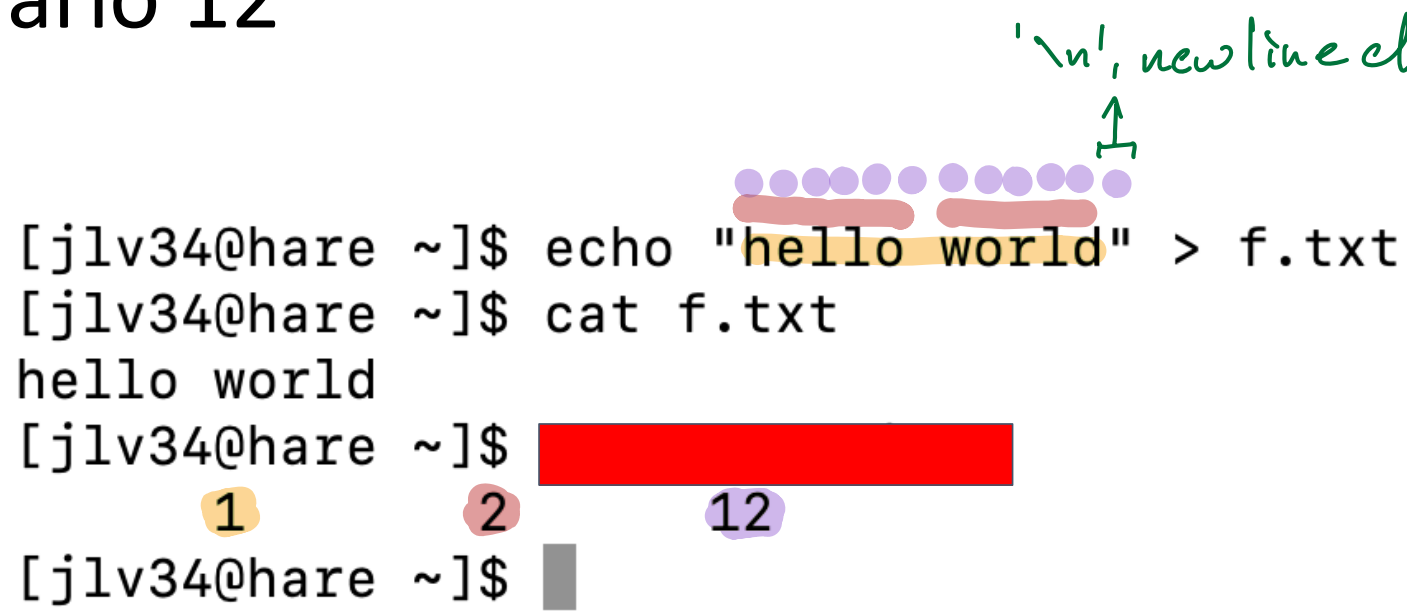
cp test2/file . 3 Copy file from one location to another

scenario 11

```
[jlv34@zebra test1]$ pwd
/home/accts/jlv34/midterm1_prep/test1
[jlv34@zebra test1]$ ls -l
total 4
drwxrwxr-x 2 jlv34 jlv34 4096 Feb 23 17:16 test2
[jlv34@zebra test1]$ cd test2
[jlv34@zebra test2]$ echo "hello world" > file
[jlv34@zebra test2]$ cd ..
[jlv34@zebra test1]$ cp test2/file .
[jlv34@zebra test1]$ ls -l
total 4
-rw-rw-r-- 1 jlv34 jlv34   12 Feb 23 17:17 file
drwxrwxr-x 2 jlv34 jlv34 4096 Feb 23 17:17 test2
[jlv34@zebra test1]$ cat file
hello world
[jlv34@zebra test1]$
```

Copies the file named file from the test2 directory to the current directory (test1). the '.' represents the current directory

scenario 12



cat f.txt | wc 3 Word count → the result is piped

The file has 1 line, 2 words, and 12 characters

scenario 12

```
[jlv34@hare ~]$ echo "hello world" > f.txt
[jlv34@hare ~]$ cat f.txt
hello world
[jlv34@hare ~]$ cat f.txt | wc
      1       2      12
[jlv34@hare ~]$
```

Agenda

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

Agenda

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

Python Background

- **Python Overview:**
  - High-level, general-purpose programming language
  - Known for simplicity, readability, and ease of learning
  - Versatile, with a vast ecosystem of libraries and frameworks
- **Key Uses:**
  - Web development
  - Data science and machine learning
  - Automation and scripting
  - Scientific computing
- **Programming Paradigms:**
  - Supports object-oriented, procedural, and functional programming
- **Why Python is Important:**
  - Efficient for solving real-world problems
  - Extensive community and support networks

Agenda

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

Procedures

In Python, a **procedure** (also referred to as a **function** that does not return a value) is a reusable block of code that performs a specific task. A procedure may take input in the form of arguments and typically modifies data or performs an action, but it does not necessarily return a result.

Key characteristics:
Definition: A procedure is defined using the def keyword followed by the procedure's name and any parameters it might take.
Task-Oriented: A procedure is designed to carry out a specific task, such as modifying a list, printing data, or validating input. It doesn't always need to return a value.
Parameters (optional): Procedures can take one or more input parameters, which allow them to work with different data without rewriting the code.
No Return Value: Unlike a function, a procedure might not return any value. If it does return a value, it's more of a side effect, not its main purpose.

Example 1

replace_even(lst)

Define a Python procedure **replace_even(lst)** that changes every even number in the list **lst** to half of its value. Assume that the list contains only integers.

Examples:

>>> replace_even([2, 3, 4, 5, 6])

=> [1, 3, 2, 5, 3]

>>> replace_even([8, 7, 6, 5, 4])

=> [4, 7, 3, 5, 2]

>>> replace_even([10, 20, 30, 40])

=> [5, 10, 15, 20]

replace_even(lst)

```
def replace_even(lst):
    for i in range(len(lst)):
        if lst[ i ] % 2 == 0:  # Check if the number is even
            lst[ i ] = lst[ i ] // 2  # Replace with half the value
    return lst
```

Explanation:

- **Understand the problem:** You need to identify even numbers in a list and replace them with half of their value
- **Identify even numbers**: Use num % 2 == 0
- **Modify the list**: Loop through the list and replace even numbers with half their value
- **Use a for loop**: Iterate with range(len(lst)) to modify the list in place
- **Handle edge cases**: Consider empty lists, lists without even numbers, and negative numbers

REMEMBER: In Python, indentation matters

Example 2

square_odd(lst)

Define a Python procedure **square_odd(lst)** that changes every odd number in the list **lst** to its square. Assume that the list contains only integers.

Examples:

>>> square_odd([1, 2, 3, 4, 5])

=> [1, 2, 9, 4, 25]

>>> square_odd([9, 8, 7, 6])

=> [81, 8, 49, 6]

>>> square_odd([11, 15, 13])

=> [121, 225, 169]

square_odd(lst)

```
def square_odd(lst):
    for i in range(len(lst)):
        if lst[i] % 2 != 0:  # Check if the number is odd
            lst[i] = lst[i] ** 2  # Replace with the square
    return lst
```

Explanation:

- **Understand the problem**
- **Identify odd numbers**: Use num % 2 != 0
- **Modify the list**: Loop through and square odd numbers.
- **Use a for loop**: Iterate with range(len(lst)) to change the values in place
- **Handle edge cases**: Consider empty lists, no odd numbers, and negative numbers

Example 3

divisible_by(base, lst)

Define a Python procedure **divisible_by(base, lst)** that changes every element in **lst** that is divisible by base to True, and every other element to False. Assume that the list contains only integers.

Examples:

>>> divisible_by(3, [3, 6, 9, 12])

=> [True, True, True, True]

>>> divisible_by(2, [1, 2, 3, 4, 5])

=> [False, True, False, True, False]

>>> divisible_by(5, [5, 10, 15, 20])

=> [True, True, True, True]

divisible_by(base, lst)

```
def divisible_by(base, lst):
    for i in range(len(lst)):
        lst[i] = (lst[i] % base == 0)  # True if divisible by base, False otherwise
    return lst
```

Explanation:

- **Understand the problem**
- **Identify odd numbers**: Use num % base == 0
- **Modify the list**: Replace each element with True or False
- If the remainder is 0 (meaning num is divisible by base), the comparison num % base == 0 returns True.
- If the remainder is not 0 (meaning num is **not** divisible by base), the comparison returns False.
- **Use a for loop**
- **Handle edge cases**: Consider empty lists, zero as a base, and negative numbers

Example 4

replace_and_square(lst)

Define a Python procedure **replace_and_square(lst)** that replaces every negative number in the list with zero and squares every positive number.

Examples:

>>> replace_and_square([1, -2, 3, -4, 5])

=> [1, 0, 9, 0, 25]

>>> replace_and_square([-10, 0, 25, -300, 42])

=> [0, 0, 625, 0, 1764]

>>> replace_and_square([0, -1, 1, -1, 2])

=> [0, 0, 1, 0, 4]

replace_and_square(lst)

```
def replace_and_square(lst):
    return [x**2 if x > 0 else 0 for x in lst]
```

Explanation:

• **List comprehension**: This is a compact way to create a new list based on an existing list (**lst**), by iterating over its elements and applying some conditions or transformations

The syntax is: **[expression** for **item** in **iterable** if **condition]**

expression: What to include in the new list
item: Each element in the list you're iterating over
iterable: The list (or another iterable) you are looping through
condition: An optional condition to filter element

- **if x > 0:**
- If **x** is positive (**x > 0**), it squares the number (**x**2**)
- If **x** is zero or negative (**else**), it replaces the number with **0**

• **Return the modified list**: The result is a new list where negative numbers are replaced with zero and positive numbers are squared

Example 5

keep_long_words(lst, n)

Define a Python procedure **keep_long_words(lst, n)** that takes a list of strings **lst** and an integer **n**, and returns a new list containing only the words from **lst** that are longer than **n** characters.

Examples:

>>> keep_long_words(["apple", "banana", "cherry", "date"], 5)

=> ['banana', 'cherry']

>>> keep_long_words(["cat", "elephant", "lion", "tiger"], 3)

=> ['elephant', 'lion', 'tiger']

>>> keep_long_words(["dog", "hippopotamus", "whale", "bat"], 4)

=> ['hippopotamus', 'whale']

keep_long_words(lst, n)

```
def replace_and_square(lst):
    return [x**2 if x > 0 else 0 for x in lst]
```

Explanation:

•**List comprehension**: We use a list comprehension to iterate through **lst** and apply transformations to each element

if x > 0:

If **x** is positive (**x > 0**), it squares the number (**x**2**)
If **x** is zero or negative (**else**), it replaces the number with **0**

Return the modified list: The result is a new list where negative numbers are replaced with zero and positive numbers are squared

Example 6

double_integers(tree)

Define a Python procedure **double_integers(tree)** that doubles every integer in a nested list structure **tree**. The procedure should traverse the nested list and, whenever an integer is encountered, it should replace it with twice its value. Non-integer elements (like strings, floats, or other data types) should remain unchanged. You may assume that the nested list can contain any combination of integers, lists, and other data types.

- Do not use any auxiliary procedures for this problem.
- You may use iteration or recursion.

Examples:

```
>>> double_integers([1, 2, [3, 'hello', [4, 5], 'world'], 6])

=> [2, 4, [6, 'hello', [8, 10], 'world'], 12]

>>> double_integers([['cat', 7], 9, [11, 'dog'], [[13]]])

=> [['cat', 14], 18, [22, 'dog'], [[26]]]
```

double_integers(tree)

```
def double_integers(tree):
    if not tree:
        return tree
    if type(tree) == int:  # If it's an integer, double it
        return tree * 2
    if type(tree) != list:  # If it's not a list, leave it unchanged
        return tree
    result = [ ]
    for i in tree:
        result.append(double_integers(i))  # Recursively call on sub-elements
    return result
```

Explanation:

- **Handle empty input**: If the input **tree** is empty or **None**, return it as is
- **Double integers**: If an element is an integer, return **tree * 2** (double its value)
- **Leave non-lists unchanged**: If the element is not a list and not an integer (e.g., string), return it unchanged
- **Recursively process lists**: If the element is a list, recursively apply the function to each element within the list
- **Return modified list**: Build and return a new list with doubled integers and unchanged non-integer elements

Example 7

replace_integers_in_range(new_val, lower_bound, upper_bound, tree)

Define a Python procedure **replace_integers_in_range(new_val, lower_bound, upper_bound, tree)** that replaces every integer in the nested list structure **tree** that falls within the inclusive range **[lower_bound, upper_bound]** with **new_val**. Non-integer elements and integers outside the specified range should remain unchanged. The list may contain a mixture of integers, lists, and other data types (e.g., strings, floats, etc.).

- You may assume the nested list can have any depth
- Do not use any auxiliary procedures for this problem
- You may use iteration or recursion

Examples:

```
>>> replace_integers_in_range(0, 3, 6, [1, 2, [3, 4, 'apple', [5, 6]], 7, 'banana'])

=> [1, 2, [0, 0, 'apple', [0, 0]], 7, 'banana']

>>> replace_integers_in_range(-1, 10, 20, [10, [15, 5, 'cat'], [25, [18, 12]]])

=> [-1, [-1, 5, 'cat'], [25, [-1, -1]]]
```

Guidelines:
Base case: If the element is an integer within the specified range, replace it with **new_val**
Recursive case: If the element is a list, recursively apply the function to each element within the list
Non-integer case: If the element is not an integer and not a list, leave it unchanged

replace_integers_in_range(new_val, lower_bound, upper_bound, tree)

```
def replace_integers_in_range(new_val, lower_bound, upper_bound, tree):
    if not tree: # Base case: If tree is None or empty, return it as is
        return tree
    if type(tree) == int: # Check if the current element is an integer
        if lower_bound <= tree <= upper_bound:  # Check if the integer is within the specified range
            return new_val  # Replace with new_val if it's in range
        else:
            return tree  # Leave unchanged if it's outside the range
    if type(tree) != list: # If the current element is not a list and not an integer, return it as is
        return tree  # Leave it unchanged
    result = [ ] # Recursive case: If the current element is a list, process its elements
    for i in tree:
      result.append(replace_integers_in_range(new_val, lower_bound, upper_bound, i))  # Recursively process
    return result
```

Explanation:

- **Handle empty or None input**
- **Check for integers** and replace them if they fall within the range
- **Recursively process lists** and handle nested elements
- **Return the result** after recursively modifying the list

Agenda

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

Regular Expression (regex) in Python

Regular expressions (regex) are a **sequence of characters** that form a search pattern, used for string matching and manipulation. In Python, they are used with the re module to search for patterns in strings, match specific parts, or replace text.
Some common regex functions in Python include:

re.search(): Search for a match within a string
re.match(): Check if the string starts with a match
re.findall(): Find all matches in a string
re.sub(): Substitute matches in a string with a new value

Regular expressions are a powerful tool for working with text data, offering flexibility and efficiency in tasks that involve pattern matching, validation, or transformation. Their ability to handle complex string operations makes them invaluable in many fields, from data processing to web development

Regular Expression (regex) in Python

When to Use Regex:

- When you need to process or clean large text data (logs, documents, datasets)
- For input validation, such as checking forms for correct data entry (email, passwords)
- To extract specific data from unstructured text, such as parsing HTML/XML files or logs
- To perform batch search and replace across files or strings, e.g., reformatting date or time formats across a document

When Not to Use Regex:

- When the problem is simple: For basic string operations (e.g., checking if a string contains a specific word), regex might be overkill
- When performance is critical: For very large datasets or real-time systems, regex can be computationally expensive, so consider alternatives if speed is a concern

Regular Expressions: For each pattern, list the strings that will match from the following strings list.
Instructions: For each pattern, identify which strings (from 1 to 12) match the regular expression.

Patterns:

- **A.** `^\d{3}$`
- **B.** `^[a-zA-Z]{3}$`
- **C.** `\w+\d{2}$`
- **D.** `^[A-Z][a-z]*$`
- **E.** `^\d{2,3}-\d{2,3}-\d{4}$`
- **F.** `^[^aeiouAEIOU]*$`
- **G.** `^\d+\.\d{2}$`
- **H.** `\d{2,3}-[A-Z]+$`
- **I.** `^\w+@\w+\.\w{2,3}$`
- **J.** `^\([0-9]{3}\) \d{3}-\d{4}$`
- **K.** `^[A-Z]+[a-z]+$`
- **L.** `^\d{4}\s+\d{4}$`
- **M.** `^.{6,}$`

Strings:

1. `123`
2. `abc`
3. `password123`
4. `HelloWorld`
5. `john@example.com`
6. `456-789-1234`
7. `(123) 456-7890`
8. `12-34-5678`
9. `25.99`
10. `PASSWORD`
11. `6789 1234`
12. `Word`

Pattern A: `^\d{3}$`

- **Explanation**: Matches exactly three digits.
- **Matches**:
  - **1:** `'123'`

Pattern B: `^[a-zA-Z]{3}$`

- **Explanation**: Matches exactly three alphabetic characters (uppercase or lowercase).
- **Matches**:
  - **2:** `'abc'`

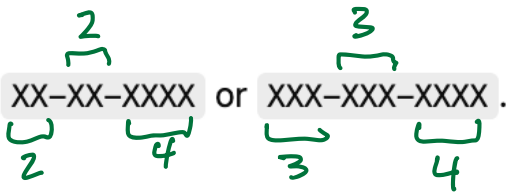
Pattern C: `\w+\d{2}$`

- **Explanation**: Matches one or more word characters followed by exactly two digits at the end.
- **Matches**:
  - 3: `'password123'`

Pattern D: `^[A-Z][a-z]*$`

- **Explanation**: Matches a string that starts with an uppercase letter and is followed by zero or more lowercase letters.
- **Matches**:
  - 12: `'Word'`

Pattern E: `^\d{2,3}-\d{2,3}-\d{4}$`



- **Explanation**: Matches numbers in the format `XX-XX-XXXX` or `XXX-XXX-XXXX`.
- **Matches**:
  - **6:** `'456-789-1234'`
  - **8:** `'12-34-5678'`

Pattern F: `^[^aeiouAEIOU]*$`

- **Explanation**: Matches strings that contain no vowels (case-insensitive).
- **Matches**:
  - **10:** `'PASSWORD'`

Pattern G: `^\d+\.\d{2}$`

- **Explanation**: Matches numbers with a decimal point followed by exactly two digits.
- **Matches**:
  - **9:** `'25.99'`

Pattern H: `\d{2,3}-[A-Z]+$`

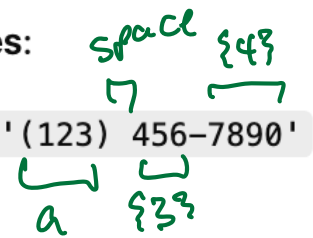
- **Explanation**: Matches two or three digits followed by a hyphen and one or more uppercase letters.
- **Matches**: None.

Pattern I: `^\w+@\w+\.\w{2,3}$`

- **Explanation**: Matches a basic email format (e.g., `user@domain.com`).
- **Matches**:
  - 5: `'john@example.com'`

Pattern J: `^\([0-9]{3}\) \d{3}-\d{4}$`

- **Explanation**: Matches a phone number in the format `(XXX) XXX-XXXX`.
- **Matches**:
  - 7: `'(123) 456-7890'`



Pattern K: `^[A-Z]+[a-z]+$`

- **Explanation**: Matches strings that start with one or more uppercase letters followed by one or more lowercase letters.
- **Matches**:
  - **12:** `'Word'`

Pattern L: `^\d{4}\s+\d{4}$`

- **Explanation**: Matches two groups of four digits separated by spaces.
- **Matches**:
  - **11:** `'6789 1234'`

Pattern M: `^.{6,}$`

- **Explanation**: Matches any string that has six or more characters.
- **Matches**:
  - **3:** `'password123'`
  - **4:** `'HelloWorld'`
  - **6:** `'456-789-1234'`
  - **7:** `'a.b.c.d'`
  - **9:** `'25.99'`
  - **10:** `'PASSWORD'`

Here are a few **tactics** to approach and solve regular expression problems like the one we've worked on:

1. Understand the Regular Expression Syntax:

- Break down each part of the regular expression.
- Look for common regex symbols:
  - `\d` for digits, `\w` for word characters, `.` for any character, etc.
  - Anchors like `^` (start of string) and `$` (end of string).
  - Quantifiers: `{n}` (exactly `n` times), `*` (zero or more), `+` (one or more).
- Identify what the pattern is specifically looking for, such as digits, word boundaries, or specific formats (e.g., phone numbers or emails).

2. Match Each Part to String Patterns:

- Look at the structure of each string (e.g., whether it has numbers, letters, punctuation, or a specific length).
- Identify strings that have similar patterns to the regex.
- Test if the string starts or ends with the specified patterns if `^` and `$` are used.

3. Apply Step-by-Step Matching:

- Start by isolating sections of the pattern.
  - For example, if the pattern is `^\d{3}$`, you know it needs **exactly** 3 digits.
  - If the pattern is `\w+@\w+\.\w{2,3}$`, focus on checking whether the string contains word characters followed by an `@` and then another domain-like pattern.

4. Identify Special Characters:

- Be aware of escape characters like `\.` which represents a **literal dot** (.) instead of any character.
- Recognize character classes like `[a-z]` or negations like `[^aeiou]` (matches anything except vowels).

5. Test Edge Cases:

- Consider potential edge cases (e.g., empty strings, strings with special characters, or strings that are too long or short).
- If a pattern contains `{6,}`, ensure you look for strings that are at least 6 characters long.

6. Use a Process of Elimination:

- For each regex, eliminate strings that clearly do not fit the pattern.
- Narrow down the possibilities as you go through each string.

7. Experiment and Iterate:

- In real-world scenarios or tests, use Python's `re` module to experiment with matching patterns.
- For practice, mentally break down how each component of the regex works and apply it to the strings provided.

Agenda

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

Python Expressions

In Python, an **expression** is any valid piece of code that produces a value. Expressions can be as simple as a single value, or they can combine multiple operations and function calls to produce more complex results. The purpose of an expression is to evaluate to a value.

Common Types of Python Expressions:

1. **Literals**:
   - Examples: Numbers (`3`, `7.5`), strings (`"hello"`, `'world'`), boolean (`True`, `False`).
   - **Usage**: These are simple values used directly in code.
   - **Example**:

```python
x = 5  # 5 is an expression that evaluates to the integer value 5
```

2. **Operators**:
    - Arithmetic (`+`, `-`, `*`, `/`, `//`, `%`, `**`), comparison (`==`, `!=`, `<`, `>`), logical (`and`, `or`, `not`).
    - **Usage**: Operators are used to manipulate data and create relationships between values.
    - **Example**:

```python
x = 5 + 3  # Expression evaluates to 8
is_even = (x % 2 == 0)  # Expression evaluates to False
```

3. **Function Calls**:
    - A call to a function returns a result, making it an expression.
    - **Usage**: Used to execute a function that returns a value.
    - **Example**:

```python
result = max(10, 20)  # Expression evaluates to 20
```

4. **Lambda Expressions**:
    - Anonymous function defined in one line.
    - **Usage**: Quick, inline functions for simple operations.
    - **Example**:

```python
square = lambda x: x * x  # Expression evaluates to a function that squares
print(square(5))  # Output: 25
```

5. **Comprehensions**:
    - A way to construct lists, sets, or dictionaries in a concise way.
    - **Usage**: Efficiently build collections from iterables.
    - **Example**:

```python
squares = [x ** 2 for x in range(5)]  # Expression evaluates to [0, 1, 4, 9
```

6. **Conditional Expressions**:
    - The `if-else` shorthand for returning a value based on a condition.
    - **Usage**: Simplifies conditional logic in a single line.
    - **Example**:

```python
result = 'even' if x % 2 == 0 else 'odd'  # Evaluates to 'odd' if x is odd,
```

How Python Expressions Are Used:

Python expressions are used in any context where a value is needed. They are the building blocks for statements like variable assignments, function arguments, control flow (e.g., `if`, `while`), and many other operations.

- **Variable Assignment**:

```python
x = 2 + 3  # The expression 2 + 3 evaluates to 5, which is assigned to x
```

- **Conditionals:**

```python
if x > 10:  # x > 10 is an expression that evaluates to True or False
    print("x is large")
```

- **Looping:**

```python
for i in range(10):  # range(10) is an expression that evaluates to an iterable
    print(i)
```

Evaluate the following Python expressions

(a) `'@'.join(list('hello'))`

=>

(b) `list(map(lambda x: x % 2 == 0, [1, 4, 7, 8, 10]))`

=>

(c) `sorted(['apple', 'banana', 'cherry', 'kiwi'], key=len)`

=>

(d) `{i: i**3 for i in range(6)}`

=>

(e) `(lambda x: (x // 2) if x % 3 == 0 else (x * 2))(9)`

=>

1. (a) `'@'.join(list('hello'))`

- **Explanation**:
  - `list('hello')` converts the string `'hello'` into a list of characters: `['h', 'e', 'l', 'l', 'o']`.
  - `'@'.join(...)` joins the elements of the list with `'@'` between each character.
- **Answer**: `'h@e@l@l@o'`

1. (b) `list(map(lambda x: x % 2 == 0, [1, 4, 7, 8, 10]))`

- **Explanation**:
  - `map(lambda x: x % 2 == 0, [1, 4, 7, 8, 10])`: The lambda function checks if each element in the list is even (`x % 2 == 0`).
  - For each element:
    - `1 % 2 == 0` → `False`
    - `4 % 2 == 0` → `True`
    - `7 % 2 == 0` → `False`
    - `8 % 2 == 0` → `True`
    - `10 % 2 == 0` → `True`
  - `list(...)` converts the result of `map` into a list.
- **Answer**: `[False, True, False, True, True]`

1. (c) `sorted(['apple', 'banana', 'cherry', 'kiwi'], key=len)`

- **Explanation**:
  - The `sorted` function sorts the list of words based on the length of each string (`key=len`).
  - The lengths of the words are:
    - `'kiwi'` → 4
    - `'apple'` → 5
    - `'banana'` → 6
    - `'cherry'` → 6
  - `'kiwi'` comes first because it has the shortest length, followed by `'apple'`. `'banana'` and `'cherry'` are both length 6, so their order depends on lexicographical order (alphabetical order).
- **Answer**: `['kiwi', 'apple', 'banana', 'cherry']`

1. (d) `{i: i**3 for i in range(6)}`

- **Explanation**:
  - This is a dictionary comprehension that iterates over the range `0` to `5` (`range(6)`).
  - For each value of `i`, the key is `i` and the value is `i**3` (the cube of `i`).
  - The result is a dictionary where the keys are numbers from 0 to 5, and the values are their cubes.
    - `0**3 = 0`
    - `1**3 = 1`
    - `2**3 = 8`
    - `3**3 = 27`
    - `4**3 = 64`
    - `5**3 = 125`
- **Answer**: `{0: 0, 1: 1, 2: 8, 3: 27, 4: 64, 5: 125}`

1. (e) `(lambda x: (x // 2) if x % 3 == 0 else (x * 2))(9)`

- **Explanation**:
  - This is a lambda function that checks if `x` is divisible by `3` (`x % 3 == 0`).
  - If `x` is divisible by `3`, it returns `x // 2` (integer division by 2).
  - Otherwise, it returns `x * 2`.
  - When `x = 9`:
    - `9 % 3 == 0` is `True`, so the result is `9 // 2 = 4` (integer division).
- **Answer**: `4`

Agenda

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

Recursion

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

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

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

flatten(lst)

Define a recursive procedure **flatten(lst)** that takes a list, which may contain nested lists, and returns a flat list with all the elements from the original list and sublists. The procedure should work for any level of nesting.

Examples:

```
>>> flatten([1, 2, [3, 4], [5, [6, 7]], 8])
[1, 2, 3, 4, 5, 6, 7, 8]
>>> flatten([['a', ['b', 'c']], 'd'])
['a', 'b', 'c', 'd']
>>> flatten([1, [2], [[3], [4]], [[[5]]]])
[1, 2, 3, 4, 5]
>>> flatten([[]])
[]
>>> flatten([1, 2, 3])
[1, 2, 3]
```

flatten(lst)

- **Base Case**:
  - If the list is empty (**[]**), return an empty list, as there are no elements to flatten
- **Recursive Case 1**:
  - If the first element is a list, recursively call **flatten** on both the first element (**lst[0],** which is a sublist) and the rest of the list (**lst[1:]**)
  - Concatenate the results of flattening both the sublist and the rest of the list
- **Recursive Case 2**:
  - If the first element is not a list, append it to the flattened version of the rest of the list

flatten(lst)

```
def flatten(lst):
    # Base case: if the list is empty, return an empty list
    if not lst:
        return []
    # Recursive case 1: if the first element is a list, flatten it recursively
    elif isinstance(lst[0], list):
        return flatten(lst[0]) + flatten(lst[1:])
    # Recursive case 2: if the first element is not a list, include it and flatten the rest
    else:
        return [lst[0]] + flatten(lst[1:])
```

flatten(lst)

Walkthrough Example: Let's say we call **flatten([1, 2, [3, 4], [5, [6, 7]], 8])**:

- The function proceeds as follows:
  - The first element is **1**, so we append it and call **flatten([2, [3, 4], [5, [6, 7]], 8])**
  - The next element is **2**, so we append it and call **flatten([[3, 4], [5, [6, 7]], 8])**
  - The next element is a list **[3, 4]**, so we recursively flatten it:
    - Flattening **[3, 4]** returns **[3, 4]**
  - We concatenate **[3, 4]** and continue with **flatten([[5, [6, 7]], 8])**
  - The next element is a list **[5, [6, 7]]**, so we recursively flatten it:
    - Flattening **[5, [6, 7]]** requires flattening **[6, 7]** first, which returns **[6, 7]**, and concatenating it with **5**
    - This results in **[5, 6, 7]**
  - We concatenate **[5, 6, 7]** and continue with **flatten([8])**
  - Finally, we append 8 to the flattened result, producing **[1, 2, 3, 4, 5, 6, 7, 8]**

flatten(lst)

Noticeable challenges:

Handling Nested Structures: Understanding how to recursively traverse a list with sublists can be tricky at first. You need need to conceptualize how to treat each element and sublist separately.

Recursive Mindset: You must develop a recursive mindset, where you trust the function will work for smaller subproblems and then combine results.

Agenda

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

set and dict Comprehensions

Comprehensions in Python are a concise way to create collections like lists, sets, and dictionaries by using a single line of code

They allow you to build these collections from existing iterables (such as lists, sets, or dictionaries) in a more readable and efficient way than using traditional loops

set Comprehensions

Set comprehension creates a set from an existing iterable, applying an optional condition or transformation. The syntax is similar to list comprehensions, but it uses curly braces **{}** instead of square brackets **[]**.

Syntax: {**expression** for **item** in **iterable** if **condition**}

Python Example:

```
nums = [1, 2, 3, 4, 1, 2, 3, 5]

unique_squares = {x**2 for x in nums}

print(unique_squares)
```

Output: {1, 4, 9, 16, 25}

Explanation: The comprehension iterates over **nums**, squares each value (**x**2**), and adds it to the set, automatically removing duplicates since sets don't allow duplicates

Key Points:

- The result is a **set**, which is an unordered collection of unique elements
- Set comprehensions automatically remove duplicate values from the result

dict Comprehensions

Dictionary comprehension creates a dictionary from an existing iterable by specifying both a key and a value for each element. The syntax is similar to list comprehensions but uses key-value pairs and curly braces **{}**

Syntax: {**key_expression: value_expression** for **item** in **iterable** if **condition**}

Python Example:

```
words = ['apple', 'banana', 'cherry']

word_lengths = {word: len(word) for word in words}

print(word_lengths)
```

Output: {'apple': 5, 'banana': 6, 'cherry': 6}

Explanation: The comprehension iterates over the list **words** and creates key-value pairs where the key is the word and the value is its length

Key Points:

- The result is a **dictionary**, which is a collection of key-value pairs
- dict comprehensions allow you to transform or filter data to build dictionaries efficiently

Key Differences Between set and dict Comprehensions

Set Comprehension: Creates a set with unique elements.

Example: {x**2 for x in range(5)}

Creates a set of squares for each number in the range 0 to 4, produces {0, 1, 4, 9, 16}

Dict Comprehension: Creates a dictionary with key-value pairs.

Example: {x: x**2 for x in range(5)}

Creates a dictionary where each number from 0 to 4 is a key, and its square is the corresponding value, resulting in {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

Why Use Comprehensions?

Conciseness: Comprehensions provide a more readable and compact way of writing loops and conditionals when creating collections

Performance: They are often faster than traditional loops for creating lists, sets, or dictionaries

Expressiveness: They express the intent of the code more clearly by focusing on **what** should be produced, not **how** to produce it

Agenda

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

Object-Oriented Programming

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects," which are instances of **classes**. OOP allows you to structure programs so that they are modular, reusable, and easier to manage, especially for large codebases

Benefits of OOP:

Modularity: Each class is self-contained and can be developed, tested, and maintained separately

Reusability: Once a class is defined, it can be reused across different parts of the program or even in other programs

Scalability: OOP allows programs to scale more easily since objects are independent and can interact in a modular way

Flexibility: With inheritance and polymorphism, you can introduce new functionality without changing existing code, which is especially useful for large applications

Midterm Example

Define a class employee and associated methods that has the following behavior.

```
e1 = employee("John", 30000)
e2 = employee("Mary", 40000)
e3 = employee("Jane", 50000)
e4 = employee("Hannah", 60000)

e1.add_supervisor(e3)
e2.add_supervisor(e3)
e3.add_supervisor(e4)

employee.members => [employee('John', 30000), employee('Mary',
40000), employee('Jane', 50000), employee('Hannah', 60000)]

employee.highest_paid() => ('Hannah', 60000)

e4.all_reports() =>
employee('Jane', 50000)
employee('John', 30000)
employee('Mary', 40000)
```

Class Employee Problem Explanation → This problem is about creating a class employee that models employees and their relationships, like supervisors and reports. The class must store the details of each employee, track who their supervisor is, and list the reports for each supervisor. Additionally, you should be able to find the employee with the highest salary using a class method.

Walkthrough of the Solution

1. Class Definition and Initialization:

- The **employee** class is defined with class-level storage **members**, which is a list that stores every employee object created.
- The **__init__** method takes the employee's name and salary and stores them in the object. Each time an employee is created, they are added to the **employee.members** list.
- When an **employee** object is created, its name, salary, and the supervisor and report information are initialized.

```
class employee:
    members = []

    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
        employee.members.append(self)
        self.supervisor = ''  # No supervisor initially
        self.reports = []     # List of employees reporting to this one
```

2. __repr__ Method:

- The **__repr__** method is used to provide a readable string representation of the employee object
- When printing an **employee** object, this method will return a string in the format: **employee('name', salary)**

```
def __repr__(self):
    return "employee({}, {})".format(repr(self.name), repr(self.salary))
```

3. add_supervisor Method:

- This method is used to assign a **supervisor** to an employee. The method adds the supervisor's name to the supervisor attribute of the employee
- It also adds the current employee to the supervisor's list of **reports**

```
def add_supervisor(self, item):
    self.supervisor = item
    item.reports.append(self)
```

Example:

e1.add_supervisor(e3) # John reports to Jane

4. all_reports Method:

- This method prints all employees that report to the current employee.
- It goes through the **reports** list of the employee and prints each one.

```
def all_reports(self):
    for r in self.reports:
        print(r)
```

Example: e4.all_reports() # This prints Jane, John, and Mary who report to Hannah.

5. highest_paid Static Method:

- This is a **static method**, meaning it doesn't require an instance of the class to be called
- It iterates over the em**ployee.members** list to find the employee with the highest salary and returns their name and salary

```
@staticmethod
def highest_paid():
    max = 0
    maxname = ''
    for e in employee.members:
        if e.salary > max:
            max = e.salary
            maxname = e.name
    return (maxname, max)
```

Example: employee.highest_paid() # This returns ('Hannah', 60000) because Hannah has the highest salary

Summary of key OOP concepts

Classes allow you to model real-world objects, such as employees

Attributes store data related to each object, such as **name**, **salary**, **supervisor**, and **reports**

Methods like **add_supervisor** and **all_reports** define behaviors associated with the object

Static Methods are used when the method's behavior is associated with the class itself (like finding the highest paid employee) rather than with any particular instance

This example demonstrates how OOP principles can be applied to manage real-world scenarios (like employee relationships, salaries, and reporting structures) in a clean, reusable, and maintainable way.

Agenda

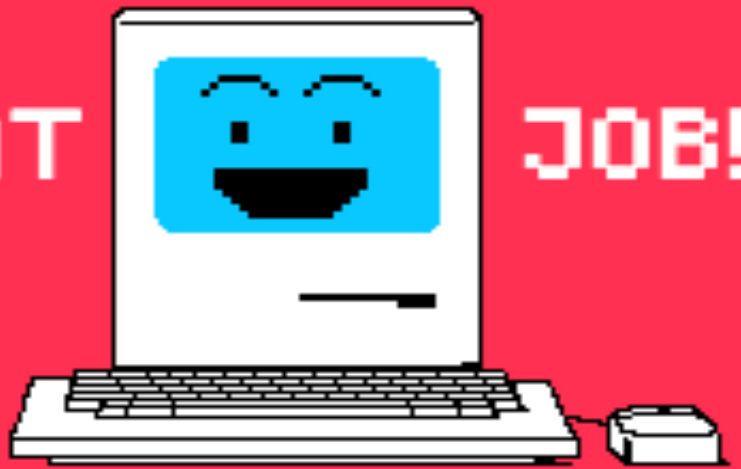
- Midterm Logistics
- Reviewing resources
- Unix
- Python review
  - Python Background
  - Procedure Examples
  - Regular Expressions
  - Python Expressions
  - Recursion
  - Comprehensions
  - Object-oriented Programming
- Final Tips

Final Tips

Practice, Practice, Practice: The more you practice coding in Python and using Unix commands, the more familiar you'll become

Understand Concepts, Don't Just Memorize: Focus on understanding how Python and Unix work. Knowing the "why" behind each concept will help during the exam

Time Management:
Practice solving problems within a time limit to simulate the exam environment
Prioritize questions based on difficulty—solve the ones you're most confident about first

YOU'RE DOING A GREAT  JOB!