Previous X

doubly linked lists with chains as nextlink and chains as previouslink of length n (user specified)

In [ ]:

In [ ]:

so lets just write it procedurally with a next x list that we keep track of

task:
running while loop populating list with random integers
every two seconds scrape the last 20 of the running list
do this 10 times
save that to a bar chart for animation sequence

In [1]:

```python
import matplotlib.pyplot as plt
import numpy as np
import time

random_list = []
go = True
all_last_twentys = []

start_time = time.time()
i = 0
while go:
    # add random number
    random_list.append(np.random.randint(0,100))

    # calculate how much time has passed since loop started
    elapsed_time = time.time() - start_time

    # every 2 seconds scrape the last 20 of the list
    # every 2 seconds reset 2 seconds counter
    # every time we reset 2 seconds counter, we increment i and
    if elapsed_time > 2:
        start_time = time.time()
        i += 1

        # every 2 seconds get the last 20 of the list
        all_last_twentys.append(random_list[-20:])
    if i == 10:
        go = False
```

In [2]:

```python
len(all_last_twentys)
```

Out[2]:

```
10
```

In [3]:

```python
# name the file sequentially
i = 0
n = 20
for last_twenty in all_last_twentys:
    x_axis = [i for i in range(n)]
    plt.title("last twenty")
    plt.bar(x_axis, last_twenty)
    plt.savefig(f"res/last_twenty_frames/image_{i}.png")
    plt.close()
    # update file name
    i += 1
```

Move x-axis out of loop

gemini code to convert image frames to animation

In [4]:

```
import cv2
import os
import re

def create_mp4_from_images(image_folder, output_path, fps=30, im
    """
    Creates an MP4 video from a sequence of images in a folder.

    Args:
        image_folder (str): The path to the folder containing th
        output_path (str): The path to the output MP4 video file
        fps (int, optional): The frames per second of the output
        image_pattern (str, optional): A regular expression patt
            Defaults to "image_\\d+\\.png", which matches files
            Use raw strings (r"pattern") to avoid issues with ba
    """
    images = [img for img in os.listdir(image_folder) if re.sear
    # Sort the images.  Important for correct video sequence.
    images.sort()  # Simple sort, might need more robust sorting

    if not images:
        print(f"Error: No images found in the folder matching th
        return

    # Determine the frame size from the first image.
    first_image_path = os.path.join(image_folder, images[0])
    img = cv2.imread(first_image_path)
    if img is None:
        print(f"Error: Could not read the first image: {first_im
        return
    height, width, channels = img.shape

    # Define the video writer.
    fourcc = cv2.VideoWriter_fourcc(*'mp4v')  # Codec for MP4
    out = cv2.VideoWriter(output_path, fourcc, fps, (width, heig

    # Write each image to the video.
    for image in images:
        image_path = os.path.join(image_folder, image)
        img = cv2.imread(image_path)
        if img is None:
            print(f"Warning: Could not read image {image_path}.
            continue  # Skip to the next image
        out.write(img)
```

```python
    # Release the video writer.
    out.release()
    print(f"Successfully created video: {output_path} from image

def main():
    """
    Main function to run the image-to-video conversion.
    """
    # Example usage:
    image_folder = "res/last_twenty_frames"  # Replace with the
    output_path = "output.mp4"  # Replace with the desired outpu
    fps = 0.5  # You can change the frames per second
    image_pattern = r"image_\d+\.png" # Example pattern, change

    # Create a dummy image folder and images if it doesn't exist
    if not os.path.exists(image_folder):
        os.makedirs(image_folder)
        for i in range(10):  # Create 10 dummy images
            import numpy as np
            dummy_image = np.random.randint(0, 256, (100, 100, 3
            cv2.imwrite(os.path.join(image_folder, f"image_{i:03

    create_mp4_from_images(image_folder, output_path, fps, image



if __name__ == "__main__":
    main()
```

```
----------------------------------------------------------------
-----------
ModuleNotFoundError                       Traceback (most recent
call last)
Cell In[4], line 1
----> 1 import cv2
      2 import os
      3 import re

ModuleNotFoundError: No module named 'cv2'
```

display the time based animation of the distribution of random numbers being populated by grabbing the last twenty every 2 seconds

please see output.mp4

output

Okay

So how do we create this in a class centric way?

well i think we can take our doublylinked class and modify it a little

if we convert the nextlink and previouslink connectors from the link class of our doublylinked list into chains themselves of length n we can easily pull out the data of the last x

so multiple ways to do this including the procedural way i just did above but we want a class oriented way to do it

so lets work on adding previousn and nextn links to our doubly linked list class

In [ ]:

```

```

Here is our doublylinkedlist class and usage using the numbers 1 through 5

In [6]:

```
class list_index:
    def __init__(self, list_item):
        self.list_item = list_item
        self.next_list_item = None
        self.previous_list_item = None

class llist:
    def __init__(self):
        self.firstitem = None
        self.enditem = None
        self.length = 0

    def add_item(self, item):
        new_list_entry = list_index(item)
        self.length += 1
        if self.firstitem == None:
            self.firstitem = new_list_entry
            self.enditem = new_list_entry
        else:
            new_list_entry.previous_list_item = self.enditem
            self.enditem.next_list_item = new_list_entry
            self.enditem = new_list_entry

    def get_item(self, index):
        if index >= 0:
            current_index = self.firstitem
            for i in range(index):
                current_index = current_index.next_list_item
            return current_index.list_item
        elif index == -1:
            return self.enditem.list_item
        else:
            current_step = self.enditem
            for i in range(1, abs(index)):
                current_step = current_step.previous_list_item
        return current_step.list_item
```

In [7]:

```
dl_list = llist()

dl_list.add_item(1)
dl_list.add_item(2)
dl_list.add_item(3)
dl_list.add_item(4)
dl_list.add_item(5)

negative_list = [-i for i in range(1, 6)]
negative_list = [-1,-2,-3,-4,-5]

# 1 based negative indexing
for item in negative_list:
    print(dl_list.get_item(item))
```

```
5
4
3
2
1
```

In [8]:

```
dl_list = llist()

dl_list.add_item(1)
dl_list.add_item(2)
dl_list.add_item(3)
dl_list.add_item(4)
dl_list.add_item(5)

# 0 based positive indexing
for i in range(dl_list.length):
    print(dl_list.get_item(i))
```

```
1
2
3
4
5
```

In [9]:

```
# please see the demo animation in readme,
# doubly linked lists just wire up the previous link before rese
# here ^
```

convert this implementation to use chains for its next and previous connections of length n (n specified by user)

well maybe a static previous n and next n is bad and we should only care about in the moment previous, in that case we dont have to change our doubly linked class at all as you can see in our for loops we are using dl_list to iterate through the whole list

if we just use the already implemented negative indexing and a constrained for loop we can already do any node of slicing starting from the n

here is an example of the outputs above

lets get the last 3

In [10]:

```python
# print the whole list
for i in range(dl_list.length):
    print(dl_list.get_item(i))
```

```
1
2
3
4
5
```

okay use negative indexing to only get the last 3

In [11]:

```python
last_n = 3
for i in range(1, last_n+1):
    print(dl_list.get_item(-i))
```

```
5
4
3
```

right?

so whats the problem with this?

well this is an iterative approach the previous x are not known

n has to be counted to as you can see in the for loop above

as n grows large, the number of counts grows large, and your program slows down

so we need to keep going

In [ ]:

```

```

In [ ]:

```

```

In [ ]:

```

```

So whats the current TODO:

In [12]:

```
# TODO: implement python slicing
# mostly care about negative slicing
# negative slicing already works in a user defined for loop usin
# built into get_index and a for loop to iterate through last n
```

In [13]:

```
last_n = 3
for i in range(1, last_n+1):
    print(dl_list.get_item(-i))
```

```
5
4
3
```

okay so rewrite todo for when i come back for further touches

TODO: mf just build python list slicing but dont make it count up to n every god damn time and thats it (into doubly linked lists)

In [14]:

```
# TODO: mf just build python list slicing but dont make it count
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

Okay lets add some notes

?

?

?

?

?

?

So whats this mean??

we ALREADY HAVE SLICING BUILT IN by having the get_index function of our chain classes able to use negative indexing

what we really need to do is keep a running track of slices that have already been requested

- getting the last 3 above is an indication of slicing
- if we keep a running track through lists of lists we can keep ourselves from recounting if the count can be lower than the list length

what i mean is basically we need to add a new member variable to our

chain class
we have self.firstlink
self.nextlink
self.previouslink
self.endlink

LEAVE THESE THE SAME

add a new self.slice_receipts

basically slice receipts will be a list of lists of previous subsets of our main chain

so whenever a slice is requested that we havent counted yet, we count it, and save it to the slice_receipt keyring
(this is a list of list or a chain of chains)

at each slice we check our saved slices

if we already counted that slice we just return it

if we havent we can count from the beginning or use a modification of another subset already counted to perform the slice faster

so whats a good way to preserve this:

Here is mine:

at each slice we ask:

- Where did you start?
- What direction did yougo?
- how long did you go in that direction?

example we already counted and saved a slice of the last 8,000,000 in the list
okay now user requesting a slice of the last 8,001,000

okay its actually quicker to count to 1000 than 8,001,000

so we can create a duplicate of the 8,000,000 slice in our already counted keyring

(the first elements of this list are)
index 0: where did you start? index 1: which direction did you go? negative or positive index 2: how long did you go in that direction? index 3 to end is the actual subset

This gives an example of how we can count once never count again and even use previous counts to speed up things we actually have not counted

all we have to do is add a list of lists called slice somethings that checks three things before counting again:

what is our slice list? do we have any? if no then have to count first time if yes we check to see if our preserved slices are the same or close to what the user is asking

we then use logical stepage to count less than the full amount if its quicker to modify a previous slice than to build a new one entirely

again we then save the newely created subset as a list into slice receipts or something (name up to you)

again slicereceipt is a list of lists where each list represents a previous slice

the first index -> where did the slice start
the next index -> which direction did you go +, or -?
the next index -> how far did you go in that direction? the next index to end of list is actually the subset

right?

so basically what do we need to do:

TODO: take doublylinkedlist class and add explicit ability to slice (we already see it has it using a for loop) merge

for loop into actual class function so we can just call slice

TODO: keep running track of slices so if we can count less or not count again we do!

That's it i think

okay so start with adding slice to doubly linked lists below

In [15]:

```
class list_index:
    def __init__(self, list_item):
        self.list_item = list_item
        self.next_list_item = None
        self.previous_list_item = None

class llist:
    def __init__(self):
        self.firstitem = None
        self.enditem = None
        self.length = 0

    def add_item(self, item):
        new_list_entry = list_index(item)
        self.length += 1
        if self.firstitem == None:
            self.firstitem = new_list_entry
            self.enditem = new_list_entry
        else:
            new_list_entry.previous_list_item = self.enditem
            self.enditem.next_list_item = new_list_entry
            self.enditem = new_list_entry

    def get_item(self, index):
        if index >= 0:
            current_index = self.firstitem
            for i in range(index):
                current_index = current_index.next_list_item
            return current_index.list_item
        elif index == -1:
            return self.enditem.list_item
        else:
            current_step = self.enditem
            for i in range(1, abs(index)):
                current_step = current_step.previous_list_item
        return current_step.list_item
```

In [16]:

```
dl_list = llist()

dl_list.add_item(1)
dl_list.add_item(2)
dl_list.add_item(3)
dl_list.add_item(4)
dl_list.add_item(5)

negative_list = [-i for i in range(1, 6)]
negative_list = [-1,-2,-3,-4,-5]

# 1 based negative indexing
for item in negative_list:
    print(dl_list.get_item(item))
```

```
5
4
3
2
1
```

In [17]:

```
# show slicing already works, but we are recounting everytime
start = -1
end = -3
number_steps = 3
# this will grab the last 3 elements if we loop it
for i in range(1, number_steps+1):
    print(dl_list.get_item(-i))

print('we just sliced the last 3')
```

```
5
4
3
we just sliced the last 3
```

In [18]:

```
# okay do a medium slice
# slice from -2 to -5:
for i in range(3, 6):
    print(dl_list.get_item(-i))
```

```
3
2
1
```

In [19]:

```
# ^ this returning a reversed list showing negative direction whe

#dl_list[-5:-2]
#and would actually print
dl_pythonlist = [1, 2, 3, 4, 5]
dl_pythonlist[-5:-2]
```

Out[19]:

```
[1, 2, 3]
```

In [20]:

```
# so that shows we need some modifications there too

# anyways just add this as logic to our class and we fix it late
```

In [21]:

```
class list_index:
    def __init__(self, list_item):
        self.list_item = list_item
        self.next_list_item = None
        self.previous_list_item = None

class llist:
    def __init__(self):
        self.firstitem = None
        self.enditem = None
        self.length = 0

    def add_item(self, item):
        new_list_entry = list_index(item)
        self.length += 1
        if self.firstitem == None:
            self.firstitem = new_list_entry
            self.enditem = new_list_entry
        else:
            new_list_entry.previous_list_item = self.enditem
            self.enditem.next_list_item = new_list_entry
            self.enditem = new_list_entry

    def get_item(self, index):
        if index >= 0:
            current_index = self.firstitem
            for i in range(index):
                current_index = current_index.next_list_item
            return current_index.list_item
        elif index == -1:
            return self.enditem.list_item
        else:
            current_step = self.enditem
            for i in range(1, abs(index)):
                current_step = current_step.previous_list_item
        return current_step.list_item

    def slice(self, start, end):
        chain = llist()
        for i in range(start, end):
            chain.add_item(self.get_item(i))
        return chain
```

In [22]:

```
my_list = llist()
my_list.add_item(1)
my_list.add_item(2)
my_list.add_item(3)
my_list.add_item(4)
my_list.add_item(5)

for i in range(my_list.length):
    print(my_list.get_item(i))
```

```
1
2
3
4
5
```

In [23]:

```
sliced = my_list.slice(0, 2)
sliced.length
```

Out[23]:

```
2
```

In [24]:

```
current_slice_item = sliced.firstitem
print(current_slice_item.list_item)
current_slice_item = current_slice_item.next_list_item
print(current_slice_item.list_item)
current_slice_item = current_slice_item.next_list_item
print(current_slice_item)
```

```
1
2
None
```

In [25]:

```
# ^ that is slicing, we just need to add logic to speed it up be
# every time we use slicing it is paternistic
# so again, we need to add previous slice receipts and logic of
# so slicing already done, we just need to stop ourselves from r
# this will just be a list of lists containing previous slices a
# again with:
# index 1 where did the slice start
# index 2 which direction did the slice go
# index 3 how long did the slice go in that direction
# index 4 to end: the actual subset created by the slice
```

In [ ]:

```
```