Lists

Welcome back to CS 2100!

Prof. Rasika Bhalerao

We've seen how to create lists by listing their elements:

```
my_nums: List[int] = [6, 7, 8, 9]
words: List[str] = ['never', 'gonna', 'give', 'you', 'up']
```

`split()`: split a `str` into separate words

```
words: List[str] = 'never gonna give you up'.split()
```

```
['never', 'gonna', 'give', 'you', 'up']
```

Optional parameter `sep` to use something other than whitespaces:

```
lyric = 'never gonna give you up'
words: List[str] = lyric.split(sep = 'e')
```

```
['n', 'v', 'r gonna giv', ' you up']
```

`join()`: combine a list of `str` into a single `str`

```
phrase: str = ' '.join(
    ['never', 'gonna', 'give',
    'you', 'up'])

also_phrase: str = 'e'.join(
    ['n', 'v', 'r gonna giv',
    ' you up'])
```

`str` to the left of the `.join()` is used as "glue" or "fenceposts" between the combined `str`s

List indices

In Python (and most other programming languages), lists are indexed starting with 0 on the very left, and increasing as it goes to the right.

```python
words: List[str] = 'never gonna give you up'.split()
second_word: str = words[1]
first_word: str = words[0]


first_three_words: List[str] = [first_word, second_word, words[2]]
print(first_three_words)   # ['never', 'gonna', 'give']
```

Remember: this means that the last index in the list is its length minus one

What happens if we try to access an index that is larger than that:

```
IndexError: list index out of range
```

Python also has a second set of indices starting with -1 on the very right, and counting down (more negative) as it steps leftward.

```python
last_word: str = words[-1]
penultimate_word: str = words[-2]
print(f'{penultimate_word} {last_word}')
```

```
you up
```

Poll: What does this evaluate to?

```
'never gonna give you up'.split()[-4]
```

1. never
2. gonna
3. give
4. you
5. up

List slices

Use "slicing" to get a sub-list (a contiguous part of the list)

```
letters: List[str] = list('abcdefghijklmnopqrstuvwxyz')
print(letters)  # ['a', 'b', 'c', 'd', ..., 'x', 'y', 'z']

second_third_fourth_letters: List[str] = letters[2:5]
print(second_third_fourth_letters)  # ['c', 'd', 'e']
```

In brackets: starting index (inclusive) and stopping index (exclusive)

- The start must be to the left of the stop.
- Both must be valid indices.
- It returns a new list that is a copy of that part of the original list, without modifying the original list.

Omit starting index: start at the very beginning of the list

```python
letters: List[str] = list('abcdefghijklmnopqrstuvwxyz')
print(letters[:4])  # ['a', 'b', 'c', 'd']
```

Omit stopping index: end at the very end of the list

```python
letters: List[str] = list('abcdefghijklmnopqrstuvwxyz')
print(letters[20:])  # ['u', 'v', 'w', 'x', 'y', 'z']
```

Omit both: create a copy of the entire list

```python
letters: List[str] = list('abcdefghijklmnopqrstuvwxyz')
print(''.join(letters[:]))  # abcdefghijklmnopqrstuvwxyz
```

Poll: What is printed?

```python
letters: List[str] = list('abcdefghijklmnopqrstuvwxyz')

print(f'{letters[-len(letters)]} {''.join(letters[23:])} {letters[-1]}')
```

1. `a wxyz z`
2. `a xyz z`
3. `z wxyz z`
4. `z xyz z`

Modifying a list

Replace elements:

```
my_nums[-1] = 900
```

Insert or append elements (which makes the list longer):

```
my_nums: List[int] = [5, 6, 7, 8, 9]

my_nums.insert(3, 600)
print(my_nums)  # [5, 6, 7, 600, 8, 9]

my_nums.append(10)
print(my_nums)  # [5, 6, 7, 600, 8, 9, 10]
```

Modifying a list

Replace elements:

```
my_nums[-1] = 900
```

Insert or append elements (which makes the list longer):

```
my_nums: List[int] = [5, 6, 7, 8, 9]
my_nums.insert(3, 600)
print(my_nums)  # [5, 6, 7, 600, 8, 9]
my_nums.append(10)
print(my_nums)  # [5, 6, 7, 600, 8, 9, 10]
```

Append multiple elements at once using `extend()`:

```
my_nums.extend([700, 800, 900])
print(my_nums)   # [5, 6, 7, 600, 8, 9, 10, 700, 800, 900]
```

Poll: Which function takes a list of integers, and returns a copy of it, but without the negative numbers?

a)

```python
def positive_copy(nums: List[int]) -> List[int]:
    result: List[int] = list()
    for i in nums:
        if i >= 0:
            result.append(i)
    return result
```

b)

```python
def positive_copy(nums: List[int]) -> List[int]:
    result: List[int] = list()
    for i in range(len(nums)):
        if i >= 0:
            result.append(i)
    return result
```

c)

```python
def positive_copy(nums: List[int]) -> List[int]:
    result: List[int] = list()
    for i in range(len(nums)):
        if nums[i] >= 0:
            result.append(nums[i])
    return result
```

d)

```python
def positive_copy(nums: List[int]) -> List[int]:
    result: List[int] = list()
    for i in range(0, -len(nums), -1):
        if nums[i - 1] >= 0:
            result.insert(0, nums[i - 1])
    return result
```

List comprehension

- Creating an empty list and adding elements one by one is not efficient.
- It's also hard to read.
- Instead, we can use **list comprehension** to let Python optimize it.

Use list comprehension to make a copy of the list, but with each element increased by one:

```python
my_nums: List[int] = [6, 7, 8, 9]

increased_nums: List[int] = [i + 1 for i in my_nums] # list comprehension

print(increased_nums)  # [7, 8, 9, 10]
```

One way to look at this format: that we moved the body of a `for loop` to right before the `for` (after the opening bracket `[`).

If we want the resulting list to **filter elements**, we add the `if` clause after the `for` clause.

`positive_copy()` using list comprehension:

```
def positive_copy(nums: List[int]) -> List[int]:
    return [i for i in nums if i >= 0]
```

Poll: What does this function do?

```python
def thing(n: int, m: int) -> float:
    total: int = sum([int(random() * n) for i in range(m)])
    return total / n
```

1. It returns a list of `n` random numbers between 0 and `m`
2. It returns a list of `m` random numbers between 0 and `n`
3. It returns the average of `n` random numbers between 0 and `m`
4. It returns the average of `m` random numbers between 0 and `n`

List comprehension can be used for things that aren't lists

Iterate over a string and create a set:

```python
phrase: str = 'never gonna give you up'

letters: Set[str] = {letter.lower() for letter in phrase}

print(letters)  # {'v', 'g', 'i', 'o', 'n', 'a', 'y', 'p', 'u', 'e', 'r', ' '}
```

It is always up to you to decide which version is easiest to read for your code. Sometimes, a basic `for` loop is more readable.

The Accumulator Pattern

A large part of this course will involve *design patterns*: a structure or template that software engineers have agreed solves a common software problem.

The Accumulator Pattern is used when we want to add up, or *accumulate*, a sequence of items.

Exercise:

Let's write a function that:

1. Asks the user how many numbers they would like to input
2. Asks the user for that many numbers (`float`s)
3. Prints the minimum, maximum, and average of those numbers

Let's do it without creating any lists.

Solution:

```python
count = int(input('How many numbers? '))
sum: float = 0.0
min: float = float('inf')
max: float = float('-inf')
for _ in range(count):
    num = float(input('Please enter a number: '))
    sum += num
    if num < min:
        min = num
    if num > max:
        max = num
print(f'min: {min}\nmax: {max}\navg: {sum / count}')
```

Exercise for you as a future reader: how can we use the Accumulator Pattern to also print the median of the numbers?

Poll: Which of these describes the Accumulator Pattern?

1. Initialize the loop variable before a loop over the sequence, and update the accumulator inside the loop
2. Initialize the loop variable before a loop over the sequence, and add (`+`) to it inside the loop
3. Initialize the accumulator variable before a loop over the sequence, and update it inside the loop
4. Initialize the accumulator variable to `0` before a loop over the sequence, and update it inside the loop

Poll:

1. What is your main takeaway from today?
2. What would you like to revisit next time?