Pythonic Data Structures

Welcome back to CS 2100!

Prof. Rasika Bhalerao

Two ways to create lists

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']
```

Exercise: Let's write a function that does the opposite of `split()`

It takes a `list[str]` and `delimiter: str`, and combines it into a `str`

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
```

Built-in function that does that for us

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

```python
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

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

Exercise: Let's write a function that takes a `list` and returns the last 3 elements as another `list`

Use negative indices

Use generic types

If the list is too short, raise a `ValueError`

List slices

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

```python
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

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

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

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

Omit both: create a copy of the entire list

```
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`

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: Which of these is a one-line version of the inside of our favorite `sarcasm()` function?

1. `return ''.join([character.upper() for character in phrase if random() < 0.5])`
2. `return ''.join([character.upper() if random() < 0.5 for character in phrase])`
3. `return ''.join([character.upper() if random() else character.lower() for character in phrase])`
4. `return ''.join([character.upper() if random() < 0.5 else character.lower() for character in phrase])`

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.

Sets

A set is very similar to a list: it is a collection of items.

```
words: set[str] = {'hi', 'hi', 'hello', 'hi', 'howdy', 'hi'}
print(words)  # {'hi', 'hello', 'howdy'}
```

Differences between a set and a list:

- A set is unordered
- A set can only hold each item (at most) once -- no duplicates

Exercise

Let's write a function that takes a `str` and counts the number of unique (distinct) words in it.

Solution

```python
def count_unique_words(text: str) -> int:
    return len(set(text.split()))

print(count_unique_words('hello hi hi hello howdy hi'))  # 3
```

Some set syntax

Creating a set:

```
words: set[str] = {'hi', 'hi', 'hello', 'hi', 'howdy', 'hi'}

numbers: set[int] = set(range(5))
print(numbers)  # {0, 1, 2, 3, 4}

list_of_floats: list[float] = [3.4, 3.2, 2.9, 3.4, 3.0]
measurements: set[float] = set(list_of_floats)
print(measurements)  # {3.2, 3.0, 2.9, 3.4}
```

Some set syntax

Adding and removing items, iterating over a set, and getting its size:

```python
nums: set[float] = set()

for i in range(100):
    random_float = round(random(), 2) # random float rounded to nearest hundredth
    nums.add(random_float)

print(len(nums))

numbers: set[int] = set(range(5))
numbers.remove(3)
print(numbers)  # {0, 1, 2, 4}
```

Exercise

Let's write a function that checks if any two people in this room have the same birthday. It should have a loop that iterates (up to) 40 times.

Each iteration, it should:

- Ask the user to input their birthday via two separate `int` s: the month and the day (ask twice to get the two `int` s)
- Store their birthday as a tuple
- If that birthday is already in the set, return `True`
- If not, add it to the set

After the loop (which it should only reach if no two people have the same birthday), it should return `False` .

Solution

```python
num_students: int = 80

def any_same_birthdays() -> bool:
    birthdays: set[tuple[int, int]] = set()

    for _ in range(num_students):
        month: int = int(input('Please enter the month as a number between 1 and 12: '))
        day: int = int(input('Please enter the day as a number between 1 and 31: '))
        date: tuple[int, int] = (month, day)
        if date in birthdays:
            return True
        else:
            birthdays.add(date)

    return False
```

Some set syntax

Binary set operations:

- Union (`a | b`): a set that has all elements that are in either set `a` or set `b`
- Intersection (`a & b`): a set that has all elements that are in both set `a` and set `b`
- Subset (`a <= b`): `True` if all elements in `a` are also in `b`, and `False` otherwise
    - Strict subset (`a < b`): `True` if `a <= b` **and `a` is not equal to `b`**, and `False` otherwise
- Subtraction (`a - b`): a set that has all elements in `a` that are not in `b`

```
nums_a: set[int] = set(range(1, 5))
nums_b: set[int] = set(range(3, 9))

print(nums_a | nums_b)  # {1, 2, 3, 4, 5, 6, 7, 8}
print(nums_a & nums_b)  # {3, 4}
print(nums_a <= nums_b) # False
```

Poll: Why is there no binary "Addition" operation for sets? (There is Subtraction.)

1. Because it would be the same as the Intersection operation
2. Because it would be the same as the Union operation
3. Because it would be the same as the Subtraction operation
4. There is an Addition operation

Dictionaries

We use curly brackets (`{` and `}`) to represent sets. But we also use them to represent dictionaries:

```
print(type({'hello'}))  # <class 'set'>
print(type({}))         # <class 'dict'>
```

Curly brackets, when empty (or non-empty, but formatted a specific way), denote a dictionary.

A dictionary is also known as an "associative array".
It's like a list, but the indices are not required to be contiguous ints -- the indices can be of any type

A dictionary maps key --> value

Each key can appear at most once (the keys are a set)

Here are two examples which map each animal (`str`) to their age (`int`):

```
ages: dict[str, int] = {'elephant': 12, 'cat': 10}
print(ages)  # {'elephant': 12, 'cat': 10}

also_ages: dict[str, int] = dict([('elephant', 12), ('cat', 10)])
print(also_ages)  # {'elephant': 12, 'cat': 10}  (same as before)
```

Common difficulty in HW1: testing the keys instead of the values

The tests need to work on *any* implementation, not just the one with your specific keys

```
def test_ask_additional_questions(self) -> None:
        """Test ask_additional_questions with all 'y' responses."""

        with patch('builtins.input', side_effect=['y', 'y', 'y', 'y', 'y']):

            result = self.question_asker.ask_additional_questions()

            self.assertTrue(True in result.values())
```

Exercise

Let's write a function that takes a `str` and returns a dictionary that maps from each unique word in the `str` to the number of times it appears.

Solution

```python
def word_counter(text: str) -> dict[str, int]:
    word_counts: dict[str, int] = dict()
    for word in text.split():
        word_counts[word] = word_counts.get(word, 0) + 1
    return word_counts

print(word_counter('hello hi hi hello howdy hi'))  # {'hello': 2, 'hi': 3, 'howdy': 1}
```

Some dictionary syntax

Access a value given a key:

- brackets (`[key]`)
- `get(key)` method
    - handles the case if the `key` is not in the `dict`

```python
ages: dict[str, int] = {'elephant': 12, 'cat': 10}

print(ages['cat'])  # 10
print(ages.get('cat'))  # 10
print(ages.get('dog'))  # None
print(ages.get('dog'), 3)  # 3
print(ages['dog'])  # raises KeyError
```

Some dictionary syntax

If we add the same `key` twice, it overwrites the original `value` with the second `value`.

```
ages: dict[str, int] = {'cat': 10}

ages['elephant'] = 12
print(ages)  # {'cat': 10, 'elephant': 12}

ages.update([('elephant', 13)])
print(ages)  # {'cat': 10, 'elephant': 13}

ages['elephant'] = 14
print(ages)  # {'cat': 10, 'elephant': 14}

ages.update([('dog', 3)])
print(ages)  # {'cat': 10, 'elephant': 14, 'dog': 3}
```

Exercise

Let's write a function that helps us with Scrabble.

- A very common situation: We are playing Scrabble. We see we have 3 'O's. What can we do?
- The plan: get a map that gives us options based on a letter
- Let's write a function that takes a letter as a parameter and returns a dictionary where:
  - The keys are all possible frequencies of that letter (except zero)
  - The values are the sets of words in the dictionary with that many of that letter
- Here's a list of english words if you need one (the official Scrabble list is harder to get as a text file)

Solution

```python
def scrabble_helper(letter: str) -> dict[int, set[str]]:
    result: dict[int, set[str]] = dict()
    with open('/path/to/dictionary.txt', 'r', encoding='utf-8') as english_dict:
        for word in english_dict.readlines():
            if letter in word:
                word = word.strip()
                letter_count = word.count(letter)
                if letter_count in result:
                    result[letter_count].add(word)
                else:
                    result[letter_count] = {word}
    return result

result: dict[int, set[str]] = scrabble_helper('r')

for key, value in result.items():
    if key > 2:
        print(f'{key}: {value}')
```

Some dictionary syntax

Iterate over a `dict`:

```
ages: dict[str, int] = {'cat': 10, 'elephant': 14, 'dog': 3}
```

- over its `key` s

```
for key in ages:
    print(f"{key}'s age is {ages.get(key)}")
```

- over its `key-value` pairs

```
for key, value in ages.items():
    print(f"{key}'s age is {value}")
```

JSON (JavaScript Object Notation)

- Popular format for storing data
- Very common for APIs to send us data in JSON format
  - E.g., https://openweathermap.org/api/one-call-3
- Read as a dictionary

```python
import json, pprint

with open('example_json_data.json', 'r', encoding='utf-8') as f:
    data = json.load(f)
    pprint.pp(data)
```

Notes to run the previous example on your own later

We took the example API response from the Weather API and stored it in a file called `example_json_data.json`.

- Removed the lines with ellipses (`...`)
- Removed the commas on the lines before them
- Added an ending bracket (`}`).

`pprint` (https://docs.python.org/3/library/pprint.html) is a library for printing data in a readable format.

Poll: Which data structure is best suited for this task?

Creating a product that works differently on different operating systems, and we want to know which operating systems we need to support

1. List
2. Tuple
3. Set
4. Dictionary

Poll: Which data structure is best suited for this task?

Storing the order in which young children should stand in line

1. List
2. Tuple
3. Set
4. Dictionary

Poll: Which data structure is best suited for this task?

Storing the 7 days of the week (Sunday, Monday, Tuesday, ..., Saturday)

1. List
2. Tuple
3. Set
4. Dictionary

Poll: Which data structure is best suited for this task?

Keeping track of each student's favorite color

1. List
2. Tuple
3. Set
4. Dictionary

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?