Hashing

Welcome back to CS 2100!

Prof. Rasika Bhalerao

Poll: What does this print?

```
class Course:
    def __init__(self, department: str, course: int):
        self.department = department
        self.course = course

course_oakland = Course('CS', 2100)
course_boston = Course('CS', 2100)

print(course_oakland == course_boston)
```

1. True
2. False

Poll: How about now?

```
class Course:
    def __init__(self, department: str, course: int):
        self.department = department
        self.course = course

    def __eq__(self, other: object) -> bool:
        if not isinstance(other, Course):
            raise NotImplementedError
        else:
            return self.department == other.department and self.course == other.course


course_oakland = Course('CS', 2100)
course_boston = Course('CS', 2100)

print(course_oakland == course_boston)
```

1. True
2. False

Moving on to the "hashing" topic...

If we try to add a `Course` to a `set`, it raises a `TypeError`

```
class Course:
    def __init__(self, department: str, course: int):
        self.department = department
        self.course = course

    def __eq__(self, other: object) -> bool:
        if not isinstance(other, Course):
            raise NotImplementedError
        else:
            return self.department == other.department and self.course == other.course


course_oakland = Course('CS', 2100)

courses: set[Course] = {course_oakland}  # TypeError: unhashable type: 'Course'
```

And same thing if you try to add it as a key to a `dict`

In order to put someting in a `set` or `dict`, it needs to follow the Hashable protocol

Hashable protocol's required method: `__hash__(self) -> int`

Corresponding interface: `from collections.abc import Hashable`

Why hashing?

If we avoid the `TypeError: unhashable type: 'Course'`:

- Just use a `list` instead
- But `list`s are slower than `set`s in some situations:

```
T = TypeVar('T')

def list_contains(item: T, list: list[T]) -> bool:
    """Returns True if the item is in the list, and False otherwise"""
    for element in list:
        if element == item:
            return True
    return False
```

Can make the code smaller, but still need to check each element

And sets check containment in *constant time* (i.e., time does not depend on list length)

How hashing makes sets magically fast

Definitions:

- *hash* (verb): To map a value to an integer index
- *hash table* (noun): A list that stores elements via hashing
- *hash set* (noun): A set of elements stored using the same hash function
- *hash function* (noun): An algorithm that maps values to indexes

One possible hash function for integers: `i % length`

index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
value										

How hashing makes sets magically fast

One possible hash function for integers: `i % length`

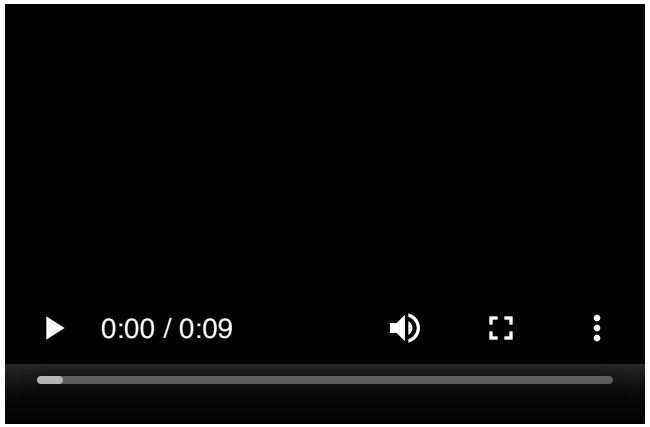
```
set.add(11)  # 11 % 10 == 1
set.add(49)  #  49 % 10 == 9
set.add(24)  #  24 % 10 == 4
set.add(7)  #  7 % 10 == 7
```

index	0	1	2	3	4	5	6	7	8	9
value		11			24			7		49

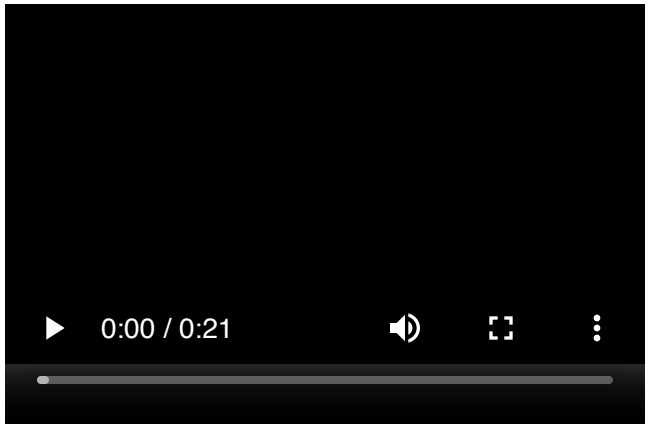
Okay, getting there, but there are some issues with that method...

dog hash collision gif

Source: https://m.vayagif.com/busqueda/0/el perro no nace agresivo/p/893



cat collision gif



no collision

Source: Tyler Yeats

How hashing makes sets magically fast

- *collision*: When hash function maps 2 values to same index

```
set.add(11)
set.add(49)
set.add(24)
set.add(7)

set.add(54)  # collides with 24
```

index	0	1	2	3	4	5	6	7	8	9
value		11			24			7		49

- *collision resolution* (noun): An algorithm for fixing collisions

How hashing makes sets magically fast

- *probing* (verb): Resolving a collision by moving to another index

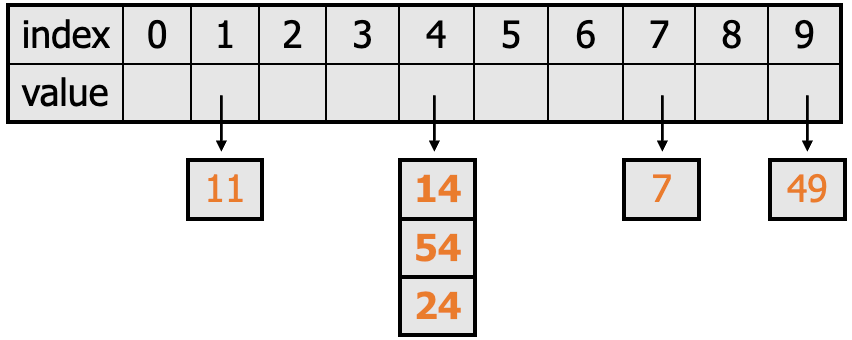
```
set.add(11)
set.add(49)
set.add(24)
set.add(7)

set.add(54)  # spot is taken -- probe (move to the next available spot)
```

index	0	1	2	3	4	5	6	7	8	9
value		11			24	54		7		49

How hashing makes sets magically fast

- *chaining* (verb): Resolving collisions by storing a list at each index
  - add / search / remove must traverse lists, but the lists are short
  - impossible to "run out" of indexes



Poll: Which of these can be done in constant time?

1. Checking if an element is in a set
2. Adding an element to a set
3. Removing an element from a set
4. Checking if a key is present in a map
5. Getting the value associated with a key in a map
6. Changing the value associated with a key in a map
7. Checking if a value appears in a map

What makes a good hash code?

`__hash__()` should return an `int` which is relatively unique to that object (so it can be used as an index in the hash table)

Rules for `__hash__()`:

- `__hash__()` must always return the same value for a given object
- If two objects are equal, then `__hash__()` must return the same value for them

Poll: Is this a legal hash function?

```
    def __hash__(self) -> int:
        return 42
```

1. Yes
2. No

In addition to the rules for hash functions...

Desired characteristics for `__hash__()`:

- We would like different objects to have different values
- The hash function should be quick to compute (ideally constant time)

Poll: Strings, numbers, and tuples are hashable by default in Python. Lists, sets, and dictionaries are not hashable by default in Python. Why might that be?

1. Because we rarely need to add lists, sets, or dictionaries to a set, or anything that requires hashing
2. Because lists, sets, and dictionaries are mutable, which could result in a changing hash code
3. Because lists, sets, and dictionaries are rarely equal to each other, so they don't need a hash code
4. Because lists, sets, and dictionaries can hold `None` in them, which shouldn't get a hash code

```python
from collections.abc import Hashable

class Course(Hashable):
    def __init__(self, department: str, course: int):
        self.department = department
        self.course = course

    def __str__(self) -> str:
        return f'{self.department}{self.course}'

    def __repr__(self) -> str:
        return self.__str__()

    def __eq__(self, other: object) -> bool:
        if not isinstance(other, Course):
            raise NotImplementedError
        else:
            return self.department == other.department and self.course == other.course

    def __hash__(self) -> int:
        return hash(str(self))


course_oakland = Course('CS', 2100)
course_boston = Course('CS', 2100)

courses: set[Course] = {course_oakland}
courses.add(course_boston)
```

Poll:

1. What is your main takeaway from today?

2. What would you like to revisit next time?