Iterating and Comparing

Welcome back to CS 2100!

Prof. Rasika Bhalerao

Recall:

Python's beautiful alternative to interfaces: Contracts

- **Interface** using `ABC` is an explicit contract: classes must follow the rules (enforced)
    - Early error detection, readability, easier to follow, requires other implementors to follow our rules, teaches fundamental concepts
- Python's built-in **contracts** are followed by convention but not enforced
    - Includes things that interfaces cannot include (like specifying *what the methods should do*, rather than simply listing the methods that need to be implemented)

Length / size protocol and `Sized` interface

Protocol:

- `def __len__(self) -> int` which returns a non-negative `int`
- this is what is returned by the `len()` function

`ABC` interface `Sized`:

- Enforces that we implement `__len__()`

```python
from collections.abc import Sized

class Cat(Sized):
    def __len__(self) -> int:
        return 900

print(len(Cat()))  # 900
```

Neglecting to implement `__len__()` (or having it return a negative number) will cause an error.

Membership test protocol and `Container` interface

"Membership test protocol" / "containment protocol":

- When you use `in`, Python calls `__contains__()`

`ABC` interface `Container`

```python
from collections.abc import Container

class Document(Container[str]):
    def __init__(self, text: str):
        self.words = text.split()

    def __contains__(self, word: object) -> bool:
        if not isinstance(word, str):
            raise TypeError
        return word in self.words

print('hi' in Document('hi this is mini'))  # True
print('cat' in Document('hi this is mini'))  # False
```

New: Iterable protocol

We can iterate over collections and `str`s using `for` loops, but we can't iterate over `int`s:

```python
for letter in 'hello':
    print(letter)

for digit in 12345:  # TypeError: 'int' object is not iterable
    print(digit)
```

That's because `str` and collections like `List` follow the Iterable protocol, and `int`s don't.

Iterable protocol (and `ABC` interface)

- Any object that follows the `Iterable protocol` can be iterated over.
- The `Iterable protocol` requires one method: `__iter__()`, which returns an an object that follows the `Iterator protocol`.

```python
from collections.abc import Iterable

class Calendar(Iterable[str]):
    def __init__(self, days: List[str]):
        self.days = days

    def __iter__(self) -> Iterator[str]:
        """Returns an iterator over the lecture days this week"""
        return iter(self.days)

for lecture in Calendar(['Monday', 'Wednesday', 'Thursday']):
    print(lecture)
```

Iterator protocol (and `ABC` interface)

```python
from collections.abc import Iterable, Iterator

class Calendar(Iterable[str]):
    def __init__(self, days: List[str]):
        self.days = days

    def __iter__(self) -> Iterator[str]:
        """Returns an iterator that returns
        every other day this week"""
        return AlternatingDayIterator(
            self.days)

class AlternatingDayIterator(Iterator[str]):
    def __init__(self, days: List[str]):
        self.days = days
        self.index: int = 0

    def __next__(self) -> str:
        if self.index >= len(self.days):
            raise StopIteration

        value = self.days[self.index]
        self.index += 2
        return value
```

```python
days = Calendar(
    ['Monday', 'Tuesday',
    'Wednesday', 'Thursday',
    'Friday'])

for alternating_day in days:
    print(alternating_day)
```

But, please use an existing Iterator when possible (like `list`'s, `set`'s, etc.)

Iterator protocol (and `ABC` interface)

The Iterator protocol and interface require two methods:

- `__next__(self) -> T`: returns the next value in the sequence, or raises `StopIteration` if there are no more values left to iterate through
- `__iter__(self) -> Iterator[T]`: returns the iterator object itself
    - Inherited version usually works, no need to rewrite

Exercise:

If we pass a `min` which is bigger than a `max`, there is no output:

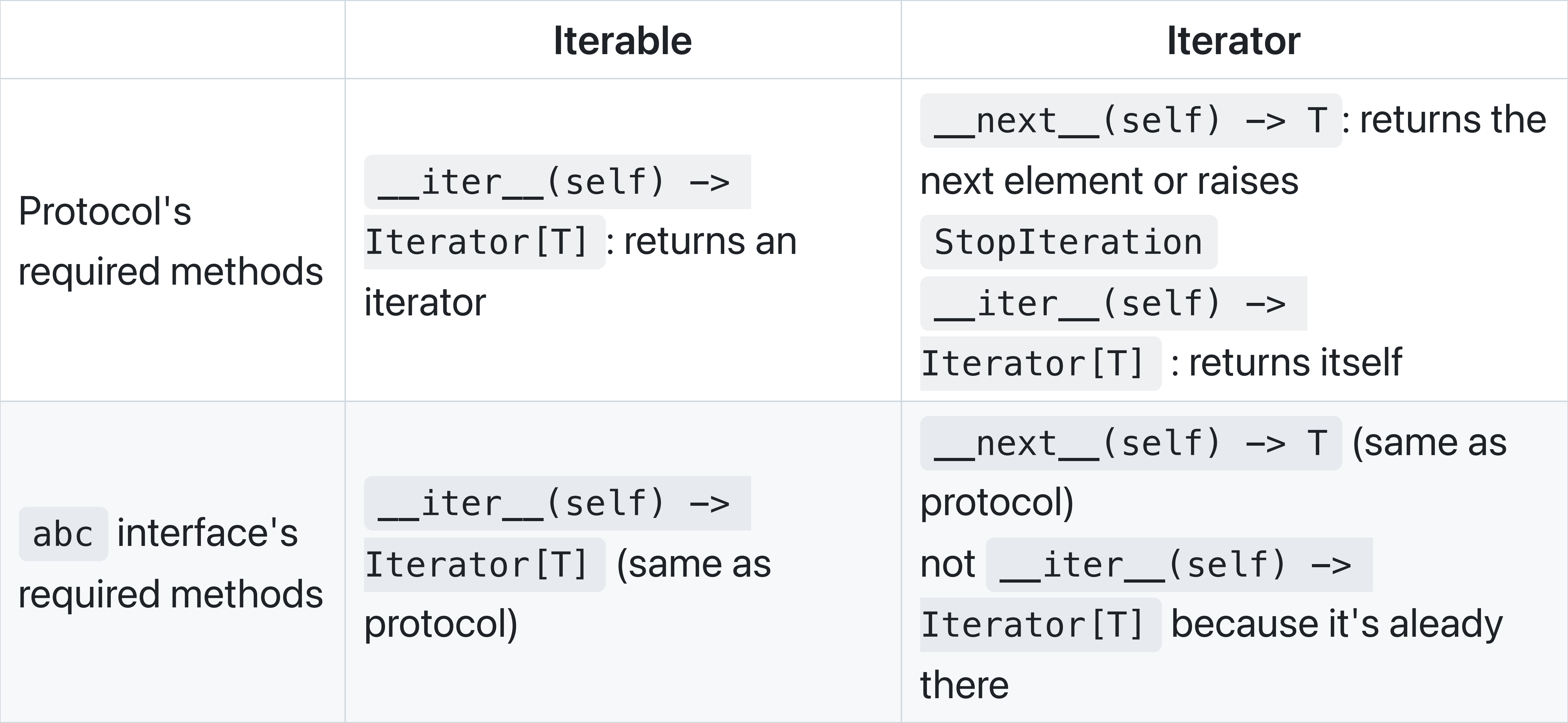
```
for i in range(5, 2):
    print(i)
```

Let's write our own version called `Range` (with a capital R) that works forwards or backwards. Our version will require a `start` and a `stop`, with an optional `step`.

```python
class Range(Iterable[int]):
    def __init__(self, start: int, stop: int, step: int = 1):
        self.start = start
        self.stop = stop
        self.step = step

    def __iter__(self) -> Iterator[int]:
        if self.start < self.stop:
            return iter(range(self.start, self.stop, self.step))
        else:
            return BackwardsIter(self.start, self.stop, self.step)

class BackwardsIter(Iterator[int]):
    def __init__(self, start: int, stop: int, step: int = 1):
        self.start = start
        self.stop = stop
        self.step = step
        self.current = start

    def __next__(self) -> int:
        if self.current <= self.stop:
            raise StopIteration
        value = self.current
        self.current -= self.step
        return value
```

Poll: In a `for` loop using the built-in `range()`, what happens if a client passes in a `min` which is bigger than the `max`?

1. Python doesn't allow it because it will iterate forever
2. Python doesn't allow it because it would raise a `StopIeration` before iterating even once
3. There is no output because it raises a `StopIteration`, which is caught by the `for` loop's internal workings
4. It will work as expected, and the loop will count down from the `min` to the `max`



	Iterable	Iterator
Protocol's required methods	`__iter__(self) -> Iterator[T]` : returns an iterator	`__next__(self) -> T` : returns the next element or raises `StopIteration` `__iter__(self) -> Iterator[T]` : returns itself
`abc` interface's required methods	`__iter__(self) -> Iterator[T]` (same as protocol)	`__next__(self) -> T` (same as protocol) not `__iter__(self) -> Iterator[T]` because it's aleady there

Modifying things while iterating over them

Lists

It is possible, though confusing / discouraged, to modify a list while iterating over it:

```
nums: List[int] = [1, 2, 3]
for i in nums:
    print(f'Element: {i}')
    print(f'Removing {nums.pop(0)}')
    print(f'List: {nums}\n')
```

```
Element: 1
Removing 1
List: [2, 3]

Element: 3
Removing 2
List: [3]
```

Instead, iterate over a copy of the list or use list comprehension:

```python
nums: List[int] = [1, 2, 3]
for i in nums.copy():  # iterating over a copy
    print(f'Element: {i}')
    print(f'Removing {nums.pop(0)}')
    print(f'List: {nums}\n')

more_nums: List[int] = [1, 2, 3, 4, 5, 6]
even = [i for i in more_nums if i % 2 == 0]  # list comprehension
print(even)
```

Modifying things while iterating over them

Sets and dictionaries

Modifying a set or a dictionary while iterating over it will result in a `RuntimeError`.

```python
nums = {1, 2, 3, 4, 5}
for item in nums:
    nums.add(600)  # RuntimeError
```

```python
my_dict = {'a': 1, 'b': 2, 'c': 3}
for key in my_dict:
    my_dict['d'] = 4  # RuntimeError
```

Though it's discouraged, and the linter complains about it, we can directly call `__iter__()` in any iterable object, and `__next__()` in any iterator object.

Poll: What happens if we iterate through all items in the collection, and then call `__next__()` after that?

```python
iterator: Iterator[int] = range(4).__iter__()
print(iterator.__next__())
print(iterator.__next__())
print(iterator.__next__())
print(iterator.__next__())
print(iterator.__next__())
```

1. `__next__()` returns `None`
2. It goes back to the beginning and iterates through the collection again
3. It raises a `RuntimeError`
4. It raises a `StopIteration`

Comparable protocol (and no interface)

- `__eq__(self, other: object) -> bool`: equals `==`
- `__ne__(self, other: object) -> bool`: not equals `!=`
- `__lt__(self, other: object) -> bool`: less than `<`
- `__le__(self, other: object) -> bool`: less than or equal to `<=`
- `__gt__(self, other: object) -> bool`: greater than `>`
- `__ge__(self, other: object) -> bool`: greater than or equal to `>=`

Don't need all six

Common: Implement `__eq__()` and one ordering method like `__lt__()`

Exercise: Let's write a class for a Plant. Plant are bigger if they get more sunlight.

```python
class Plant:
    def __init__(self) -> None:
        self.sunlight_hours = 0

    def get_sunlight(self) -> None:
        self.sunlight_hours += 1

    def __eq__(self, other: object) -> bool:
        if not isinstance(other, Plant):
            raise NotImplementedError
        return self.sunlight_hours == other.sunlight_hours

    def __lt__(self, other: object) -> bool:
        if not isinstance(other, Plant):
            raise NotImplementedError
        return self.sunlight_hours < other.sunlight_hours

plant1 = Plant()
plant2 = Plant()

plant1.get_sunlight()
```

Poll: What goes in the ??? ?

```python
class Bouquet:
    """Bouquets are compared by the number of flowers in them"""
    def __init__(self, flowers: List[Flower]):
        self.flowers = flowers

    def __eq__(self, other: object) -> bool:
        if not isinstance(other, Bouquet):
            raise NotImplementedError
        return ???

    def __gt__(self, other: object) -> bool:
        if not isinstance(other, Bouquet):
            raise NotImplementedError
        return ???
```

1. `len(self.flowers) == len(other.flowers)` and `len(self.flowers) < len(other.flowers)`
2. `len(self.flowers) == len(other.flowers)` and `len(self.flowers) > len(other.flowers)`
3. `len(self.flowers) != len(other.flowers)` and `len(self.flowers) < len(other.flowers)`
4. `len(self.flowers) != len(other.flowers)` and `len(self.flowers) > len(other.flowers)`

Poll:

1. What is your main takeaway from today?

2. What would you like to revisit next time?