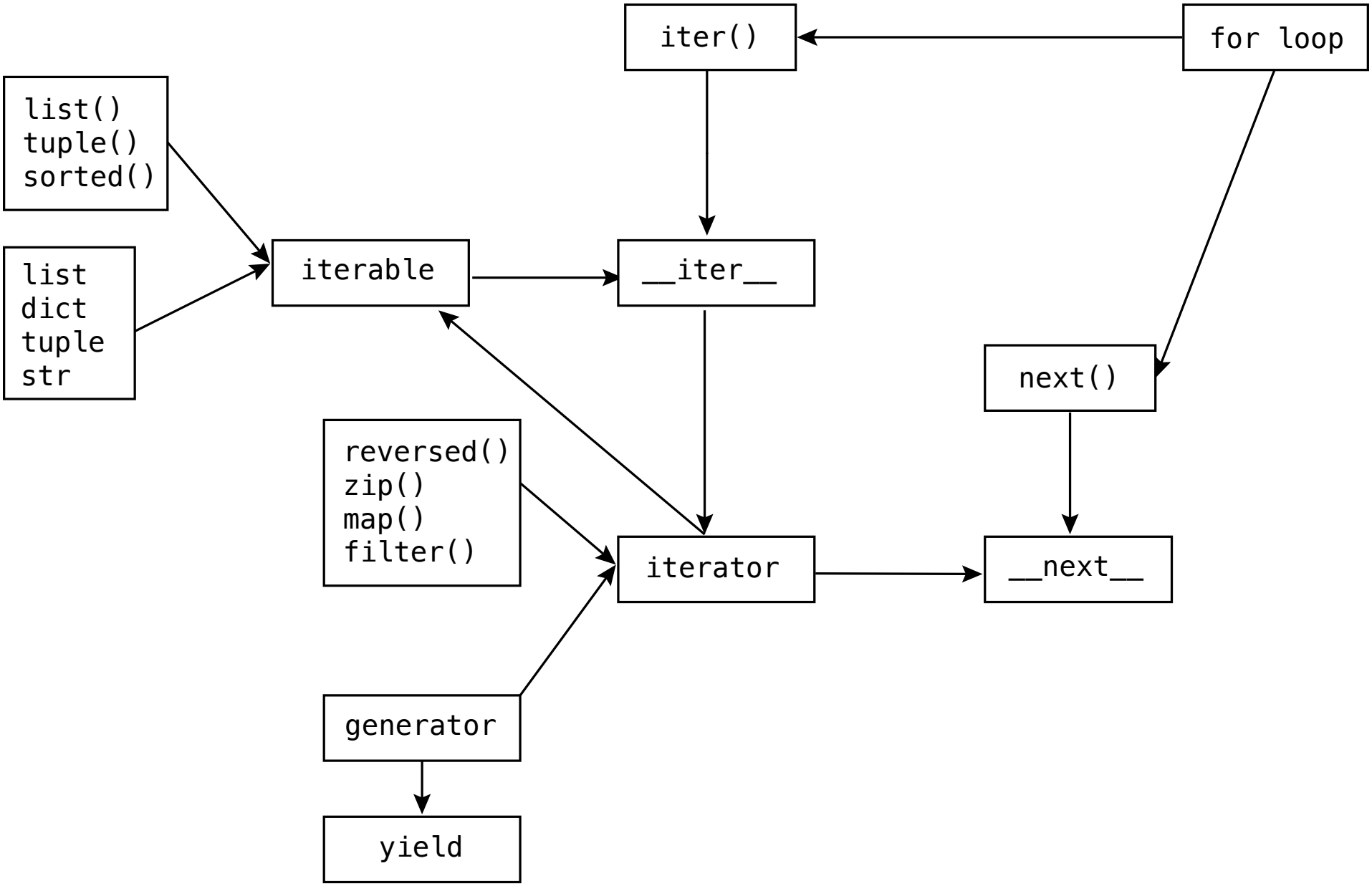
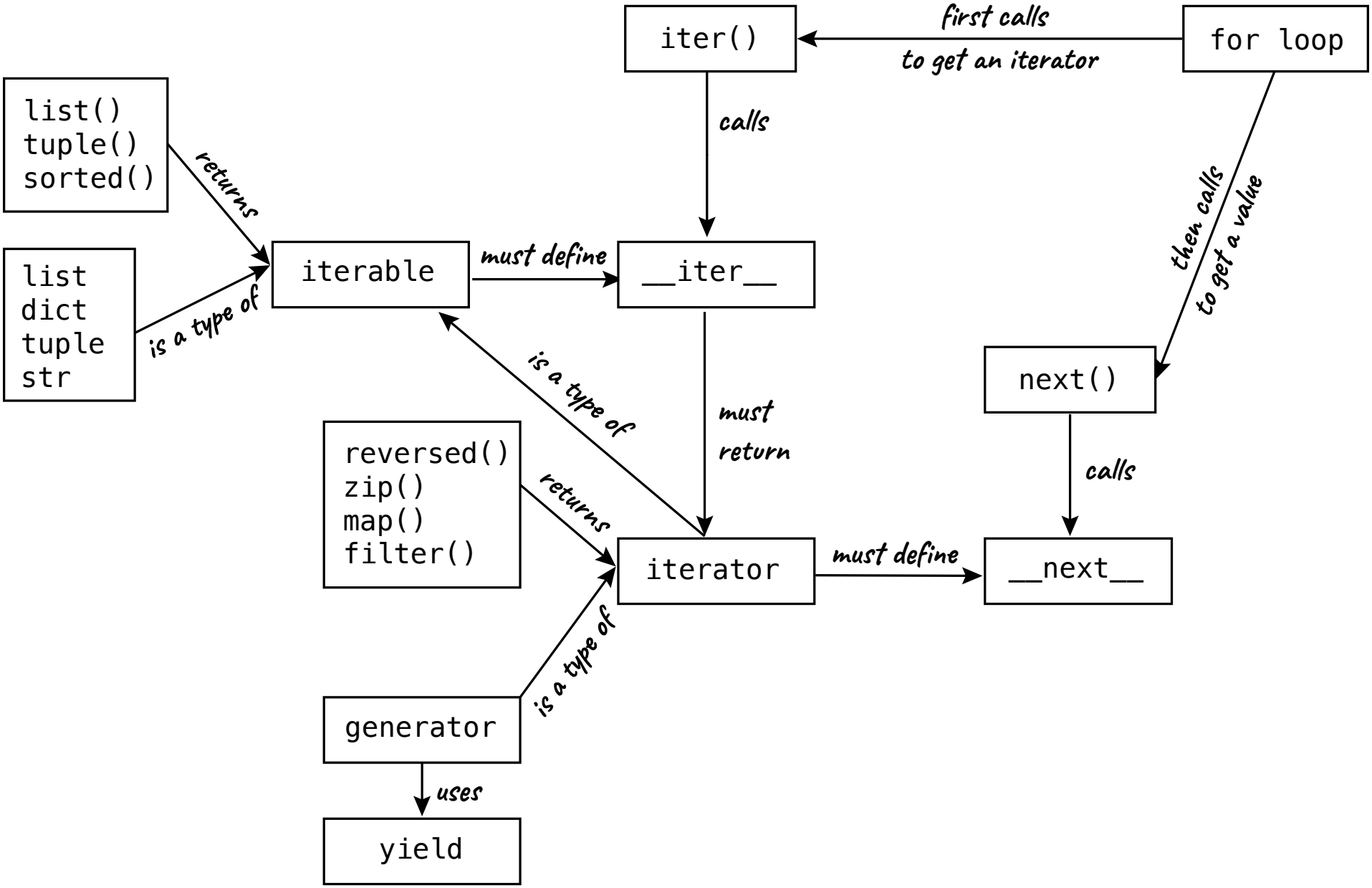
Fun with Iterables

A concept map



A concept map (labeled)



(Re-visit) Lecture quiz questions

1. If we want to be able to iterate through an instance of a custom class in Python, what method name should we define?

`__iter__`

2. What could the definition of that method return?

- A single value (like str or int)
- A higher order function
- An iterator
- A list
- An iterable
- A generator

(Re-visit) Lecture quiz questions

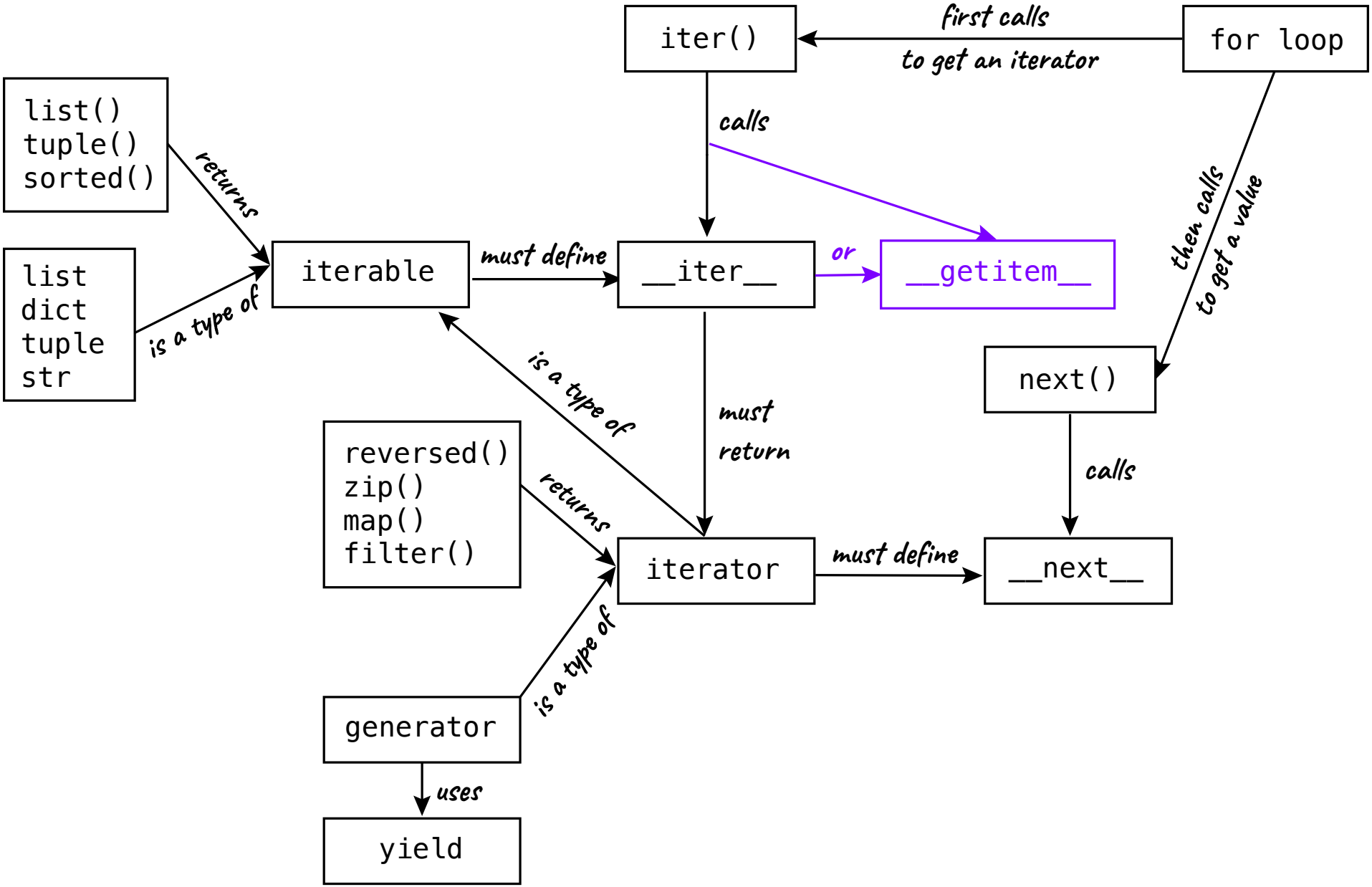
1. If we want to be able to iterate through an instance of a custom class in Python, what method name should we define?

`__iter__`

2. What could the definition of that method return?

- A single value (like str or int)
- A higher order function
- An iterator # ✅Correct
- A list
- An iterable
- A generator # ✅Correct

A concept map (complete!?)



Functions that return iterators

Function	Description
`reversed(sequence)`	Iterate over item in `sequence` in reverse order (See example in PythonTutor)
`zip(*iterables)`	Iterate over co-indexed tuples with elements from each of the `iterables` (See example in PythonTutor)
`map(func, iterable, ...)`	Iterate over `func(x)` for `x` in `iterable` Same as `[func(x) for x in iterable]` (See example in PythonTutor)
`filter(func, iterable)`	Iterate over `x` in `iterable` if `func(x)` Same as `[x for x in iterable if func(x)]` (See example in PythonTutor)

Using map and filter

```
nums = [1, 2, 3, 4, 5]

for square in map(lambda num: num ** 2, nums):
    print(square)

for odd in filter(lambda num: num % 2 == 1, nums):
    print(odd)
```

Any type of iterable can be passed in!

```
for letter in map(lambda l: l.upper(), "superkalifragilistikexpialigetisch"):
    print(f"{letter}👏")
```

`map()` can process multiple iterables, as long as the lambda accepts that number of arguments:

```
nums2 = (9, 10, 20)
for sum in map(lambda x, y: x+y, nums, nums2):
    print(sum)
```

Things that return iterables

Function	Description
`sorted(iterable, key=None, reverse=False)`	Returns a sorted list containing all items in `iterable`, using optional `key` function for comparison key.

Constructor	Description
`list(iterable)`	Constructs a new `list` containing all items in `iterable`
`tuple(iterable)`	Constructor a new `tuple` containing all items in `iterable`
`set(iterable)`	Constructs a new `set` containing all items in `iterable`

...plus all the functions on previous slide, since an iterator is iterable!

Creating iterables

```
nums = [1, 2, 3, 4, 5]

squares = list(map(lambda num: num ** 2, nums))
odds = tuple(filter(lambda num: num % 2 == 1, nums))
```

Take advantage of optional arguments...

```
grades = [73, 89, 74, 95]
lowest_first = sorted(grades)
highest_first = sorted(grades, reverse=True)
```

```
grades = ["C+", "B+", "C", "A"]
highest_first = sorted(grades)
lowest_first = sorted(grades, reverse=True)
```

Using key functions

Use a key function for sorting complex types:

```
coords = [ (37, -144), (-22, -115), (56, -163) ]

south_first = sorted(coords, key=lambda coord: coord[0])

north_first = sorted(coords, key=lambda coord: coord[0], reverse=True)
```

Using key functions

Use a key function for sorting complex types:

```
coords = [ (37, -144), (-22, -115), (56, -163) ]

south_first = sorted(coords, key=lambda coord: coord[0])

north_first = sorted(coords, key=lambda coord: coord[0], reverse=True)
```

```
coords = [{"lat": 37, "lng":-144},
          {"lat":-22, "lng":-115},
          {"lat":56, "lng":-163}]

south_first = sorted(coords, key=lambda coord: coord["lat"])
```

🤔 Which `coords` do you prefer? What else could you do? Consider readability and error-proneness.

Functions that process iterables

Function	Description
`max(iterable, key=None)`	Return the max value in `iterable`
`min(iterable, key=None)`	Return the min value in `iterable`
`sum(iterable, start)`	Returns the sum of values in `iterable`, initializing sum to `start`
`all(iterable)`	Return `True` if all elements of `iterable` are true (or if `iterable` is empty)
`any(iterable)`	Return `True` if any element of `iterable` is true. Return false if `iterable` is empty.

Processing iterables

```
max_grade = max([73, 89, 74, 95])
max_letter = max(["C+", "B+", "C", "A"])

coords = [ (37, -144), (-22, -115), (56, -163) ]
most_north = max(coords, key=lambda coord: coord[0])
most_south = min(coords, key=lambda coord: coord[0])

total_points = sum([73, 89, 74, 95], 0)

all_true = all([True, True, True, True])
any_true = any([False, False, False, True])
```

Processing iterators (as iterables)

```
numbers = [1, 2, 3]

print(sum(numbers, 0))

print(sum(numbers, 0))
```

Iterators are also iterables...

```
it = iter(numbers)

print(sum(it, 0))

print(sum(it, 0))
```

Processing iterators (as iterables)

```
numbers = [1, 2, 3]

print(sum(numbers, 0))

print(sum(numbers, 0))
```

Iterators are also iterables...

```
it = iter(numbers)

print(sum(it, 0))

print(sum(it, 0))
```

...but they can be exhausted!

All together now!

What type/value do each of these lines return?

```
nums = [1, 2, 3]
letters = ["A", "B", "C"]
```

```
iter(letters)
next(iter(letters))
```

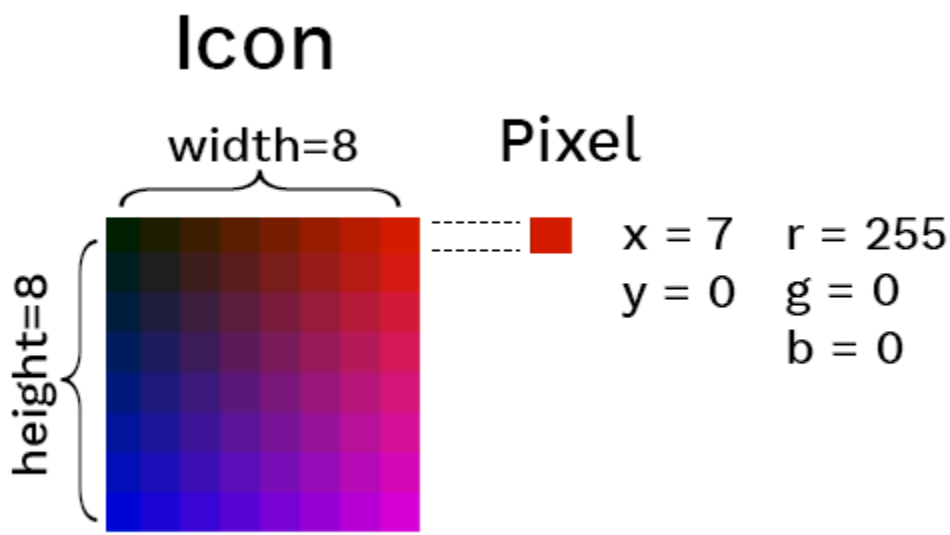
```
map(lambda n: n * 3, nums)
sorted(map(lambda n: n * 3, nums))
max(sorted(map(lambda n: n * 3, nums)))
```

```
zip(nums, letters)
tuple(map(lambda n: n * 3, nums))
list(tuple(map(lambda n: n * 3, nums)))
```

Iterable pixels

An OOP Icon

Goal: Use OOP to represent an Icon with pixels at a particular location with a particular color.



The Color class

```
class Color:

    def __init__(self, r, g, b):
        self.r = r
        self.g = g
        self.b = b

    def __repr__(self):
        return f"Color({self.r},{self.g},{self.b})"

    def to_hex(self):
        return f"#{self.r:02x}{self.g:02x}{self.b:02x}"
```

```
red = Color(255, 0, 0)
print(red.to_hex())
```

The Pixel class

```
class Pixel:
    def __init__(self, x, y, r, g, b):
        self.x = x
        self.y = y
        self.color = Color(r, g, b)

    def __repr__(self):
        return f"Pixel({self.x},{self.y},{self.color})"
```

```
pixel = Pixel(0, 7, Color(255, 0, 0))
print(pixel.color.to_hex())
```

The Icon class

```
class Icon:

    def __init__(self, width, height, pixels=None):
        self.width = width
        self.height = height
        self.pixels = pixels
        if not self.pixels:
            self.pixels = [ Pixel(x, y, 0, 0, 0)
                for x in range(width) for y in range(height)]

    def __repr__(self):
        pixels = ",".join([repr(pixel) for pixel in self.pixels])
        return f"Icon({self.width}, {self.height}, {self.pixels})"

    def __iter__(self):
        for pixel in self.pixels:
            yield pixel
```

```
icon = Icon(2, 2, [Pixel(0, 0, 255, 0, 0),
    Pixel(0, 1, 255, 50, 0),
    Pixel(1, 0, 255, 100, 0),
    Pixel(1, 1, 255, 150, 0)])

for pixel in icon:
    pixel.color.g += 50
```

The Icon class with __getitem__

```
class Icon:

    def __init__(self, width, height, pixels=None):
        self.width = width
        self.height = height
        self.pixels = pixels
        if not self.pixels:
            self.pixels = [ Pixel(x, y, 0, 0, 0)
                for x in range(width) for y in range(height)]

    def __repr__(self):
        pixels = ",".join([repr(pixel) for pixel in self.pixels])
        return f"Icon({self.width}, {self.height}, {self.pixels})"

    def __getitem__(self, index):
        return self.pixels[index]
```

```
icon = Icon(2, 2, [Pixel(0, 0, 255, 0, 0),
    Pixel(0, 1, 255, 50, 0),
    Pixel(1, 0, 255, 100, 0),
    Pixel(1, 1, 255, 150, 0)])

for pixel in icon:
    pixel.color.g += 50
pixel[0].color.b = 255
```

Visual demo

Visit the Repl.it demo to see all the classes used with the Python tkinter library for graphics rendering.

Iterator-producing functions

What happens if we...

map the pixels?

```
changer = lambda p: Pixel(p.x, p.y, p.x * 30, p.color.g + 30, p.y * 30)
icon.pixels = list(map(changer, icon.pixels))
```

filter the pixels?

```
is_odd = lambda p: p.x % 2 == 0
icon.pixels = list(filter(is_odd, icon.pixels))
```

Iterable-processing functions

What happens if we ask for the min and max of the pixels?

```
max_pix = max(icon.pixels)
min_pix = min(icon.pixels)
```

Iterable-processing functions

What happens if we ask for the min and max of the pixels?

```
max_pix = max(icon.pixels)
min_pix = min(icon.pixels)
```

Python doesn't know how to compare `Pixel` instances! Two options:

- Implement dunder methods (`__eq__`, `__lt__`, etc)
- Pass in a key function that returns a numerical value:

```
rgb_adder = lambda pixel: pixel.r + pixel.color.g + pixel.color.b
max_pix = max(icon.pixels, key=rgb_adder)
min_pix = min(icon.pixels, key=rgb_adder)
```

A concept map (complete!?)