Saving Keystrokes in Python

A Brief Guide to Fearlessly Using Python Iterators

Me

Familiarity with Python?

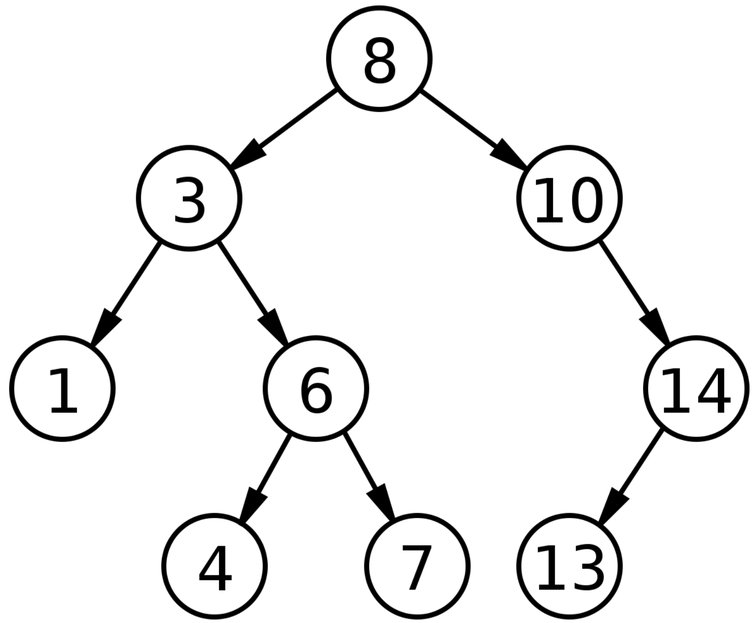
Goals of this Talk:

1. Given *Python code that makes heavy use of iterators*, you should be able to **correctly translate** it into an imperative version that has the same output (without a rise in blood pressure)

2. Given *Python code that uses an imperative style,* you should be able to **rewrite** it into a more concise version using iterators

What is an Iterator?

- An object used to traverse a collection or data-structure
  - Will visit each element once
  - Lazily evaluated
- Not necessarily finite



Iterators in Python

- Must implement the Iterator interface (aka the Iterator Protocol)
  - Two methods, `__iter__` and `__next__`
- Iterators are *consumable*
- Iterators contain state information
- Iterators are *lazy* (usually)
- The `__iter__` method usually returns self
- builtin function `iter` calls `__iter__` on an object
- builtin function `next` calls `__next__` on an object

iterator.py

```
from abc import ABC, abstractmethod

class Iterator(ABC):

    @abstractmethod
    def __iter__(self):
        pass

    @abstractmethod
    def __next__(self):
        pass
```

Iterable Objects

- Only implement the `__iter__` method
- Datastructures are usually *iterable* and not not *iterators*
- All *iterators* are *iterable*, but not all *iterables* are *iterators*

```
>>> my_list = [1,2,3]
>>> my_list
[1, 2, 3]
>>> iter(my_list)
<list_iterator object at 0x10c854410>
```

iterator.py

```
class Iterable(ABC):

    @abstractmethod
    def __iter__(self):
        pass
```

Iterators in Python - Under the Hood

forloop.py

```
mylist = []

for value in range(16):
    mylist.append(value)

# prints: [0, 1, 2, 3, ..., 15]
print(mylist)
```


```
# This accomplishes the same thing
mylist2 = []

myiter = iter(range(16))
while True:
    try:
        value = next(myiter)
    except StopIteration:
        break

    mylist2.append(value)

# prints: [0, 1, 2, 3, ..., 15]
print(mylist2)
```

Types of Iterators in Python

Standard Iterators

- Implements the Iterator interface

three_types.py

```
# Iterable Objects
class SquareNumbersObject(Iterator):

    def __init__(self, numbers: List[float]):
        self.numbers = numbers
        self.index = 0

    def __iter__(self):
        return self

    def __next__(self) -> float:
        if self.index < len(self.numbers):
            value = self.numbers[self.index]
            self.index += 1
            return value * value
        else:
            raise StopIteration()
```

```
>>> from three_types import SquareNumbersObject
>>> my_numbers = list(range(6))
>>> list(SquareNumbersObject(my_numbers))
[0, 1, 4, 9, 16, 25]
```

Generators

- Functions that use the *yield* keyword create generator objects
- Python automagically creates an instance an object implementing the Iterator interface

three_types.py

```
# Generator
def square_numbers_generator(numbers: List[float]) -> Iterator[float]:
    index = 0

    while index < len(numbers):
        value = numbers[index]
        yield value * value
        index += 1
```

```
>>> from three_types import square_numbers_generator
>>> my_numbers = list(range(6))
>>> list(square_numbers_generator(my_numbers))
[0, 1, 4, 9, 16, 25]
```

Comprehension

- Can create lists, generators, sets, and dictionaries
- Only lazy in the case of generators

three_types.py

```
def square_numbers_comprehension(numbers: List[float]) -> List[float]:
    return [x * x for x in numbers]
```

```
>>> squared_numbers = [x * x for x in range(6)]
>>> squared_numbers
[0, 1, 4, 9, 16, 25]
```

Comprehension

List creation

```
>>> [x * x for x in range(6)]
[0, 1, 4, 9, 16, 25]
```

Generator creation

```
>>> (x * x for x in range(6))
<generator object <genexpr> at 0x10ec65250>
```

Set creation

```
>>> {x * x for x in range(6)}
{0, 1, 4, 9, 16, 25}
```

Dictionary creation

```
>>> {x: x * x for x in range(6)}
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Comprehension Predicates

generators.py

```
# Using a predicate
squared_even_numbers_generator: Iterator[int] = \
        (x * x for x in range(24) if x % 2 == 0)
```

output expression input iterable predicate

Nested Comprehensions

comprehension.py

```
size = 5
matrix: List[List[float]] = np.random.rand(size, size)


# Nested iterators to flatten matrix
flattened_matrix: List[float] = \
        [matrix[row, column] for row in range(size) for column in range(size)]


# Same as
flattened_matrix2: List[float] = []
for row in range(size):
    for column in range(size):
        flattened_matrix2.append(matrix[row, column])
```

Nested Comprehensions

comprehension.py

```
# Nested iterators to construct a matrix from a sequence
reconstructed_matrix: List[List[float]] = \
        [[flattened_matrix[row * size + column] for column in range(size)] for row in range(size)]
```

comprehension.py

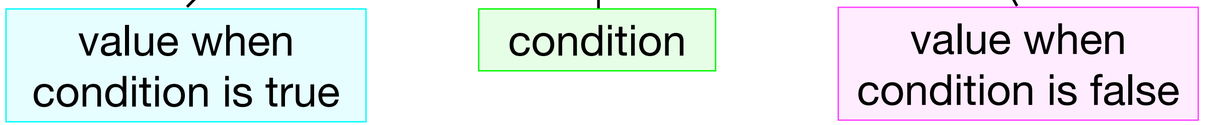
```
# Same as
reconstructed_matrix2: List[List[float]] = []
for row in range(size):
    row_list = []
    for column in range(size):
        row_list.append(flattened_matrix[row * size + column])
    reconstructed_matrix2.append(row_list)
```

If Expressions

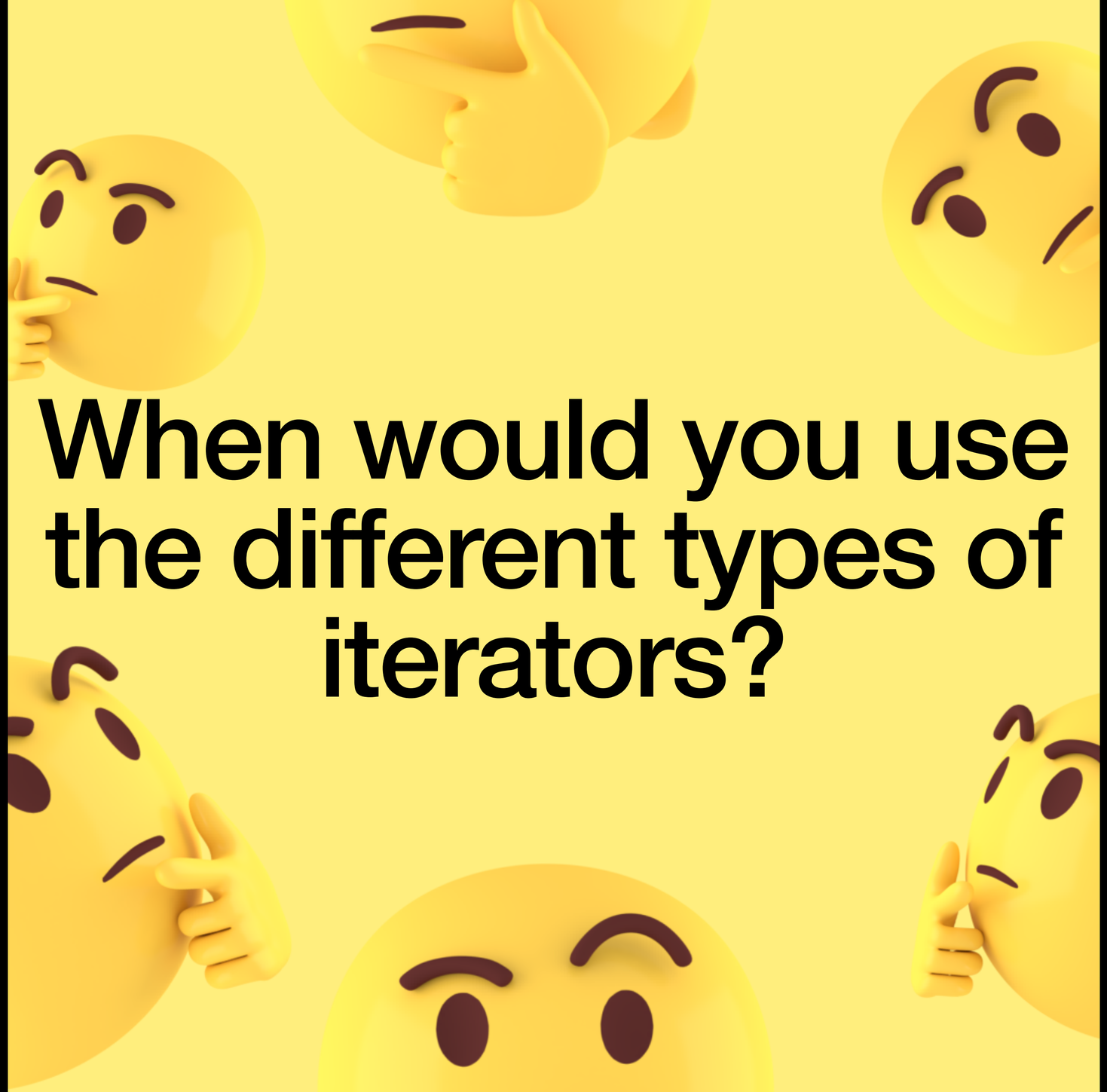
- Basically just an inline if statement
- else branch required

```
# Identity matrix using nested comprehension and an if expression
identity = [[1 if row == column else 0 for column in range(size)] for row in range(size)]
```

generators.py




```
>>> size = 5
>>> identity = [[1 if row == column else 0 for column in range(size)] for row in range(size)]
>>> _ = [print(row) for row in identity]
[1, 0, 0, 0, 0]
[0, 1, 0, 0, 0]
[0, 0, 1, 0, 0]
[0, 0, 0, 1, 0]
[0, 0, 0, 0, 1]
```


When would you use the different types of iterators?

Composing and Consuming Iterators

Built-in Functions

- Python provides a rich set of functions to compose, consume, and chain iterators
- Key functions include:
  - enumerate
  - filter
  - list, set, and dict
  - map
  - reduce
  - reversed
  - zip

zip - Built-in Functions

- Combines two or more iterables and outputs tuples
- Will stop when the end of any iterator is reached

Input Iterables

$[x_1, x_2 \ldots, x_n]$

$[y_1, y_2 \ldots, y_m]$

Where m >= n

Enumerate

Output Iterator

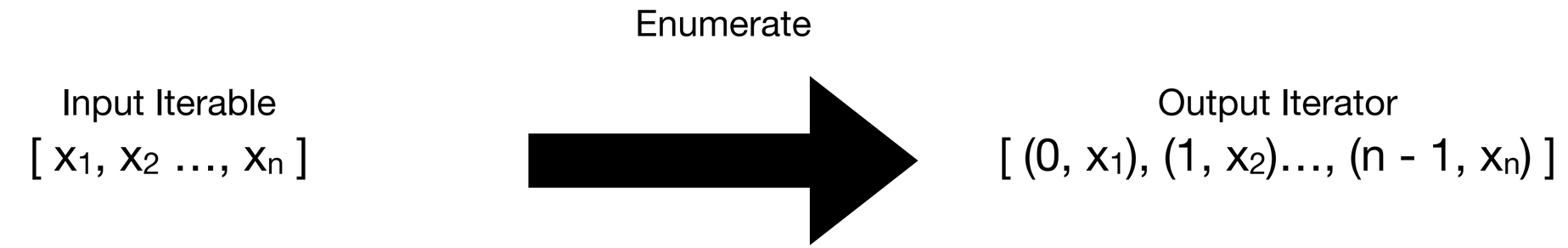
$[(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)]$

```
# Columns of a CSV file or something
first_names: List[str] = \
        ['Josh',  'Jeff',  'Jessica', 'John', 'Joanne']
last_names: List[str] = \
        ['Karns', 'Salad', 'Brown',   'Doe',  'Lee']

full_names: List[str] = \
```

builtins.py

enumerate - Built-in Functions

- Pairs input iterator values with their index in a tuple



builtins.py

```
# Columns of a CSV file or something
first_names: List[str] = \
        ['Josh',  'Jeff',  'Jessica', 'John', 'Joanne']
last_names: List[str] = \
        ['Karns', 'Salad', 'Brown',   'Doe',  'Lee']

full_names: List[str] = \
        [f'{first} {last}' for first, last in zip(first_names, last_names)]

index_to_full_name: Dict[int, str] = \
```

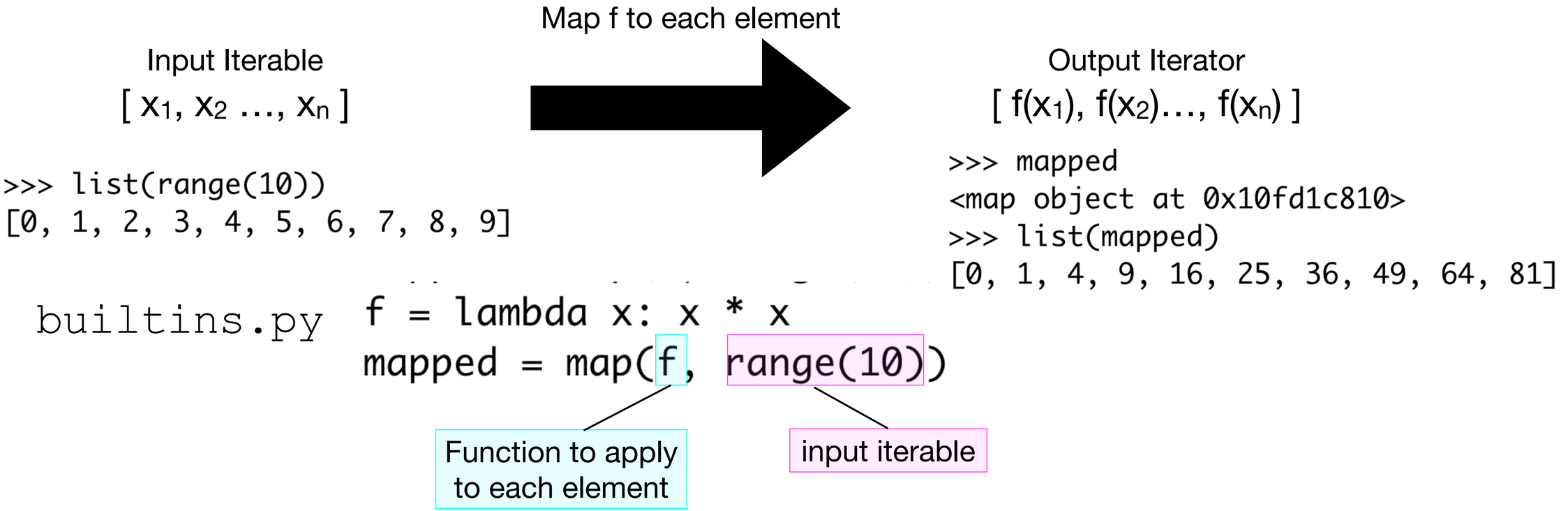
list - Built-in Functions

- Takes an *iterable* as input, turning it into a list
  - the set and dict functions are similar to this
- Can be used to evaluate lazy iterators

```
>>> numbers = range(10)
>>> numbers
range(0, 10)
>>> list(numbers)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

map - Built-in Functions

- Applies a function to each element of an iterator
- Evaluated lazily



filter - Built-in Functions

- Removes elements that do not meet a predicate
- Evaluated lazily

Filter by predicate *is_odd*

Input Iterable

[0, 1, 2, 3, 4, …]

Output Iterator

[1, 3, 5, …]

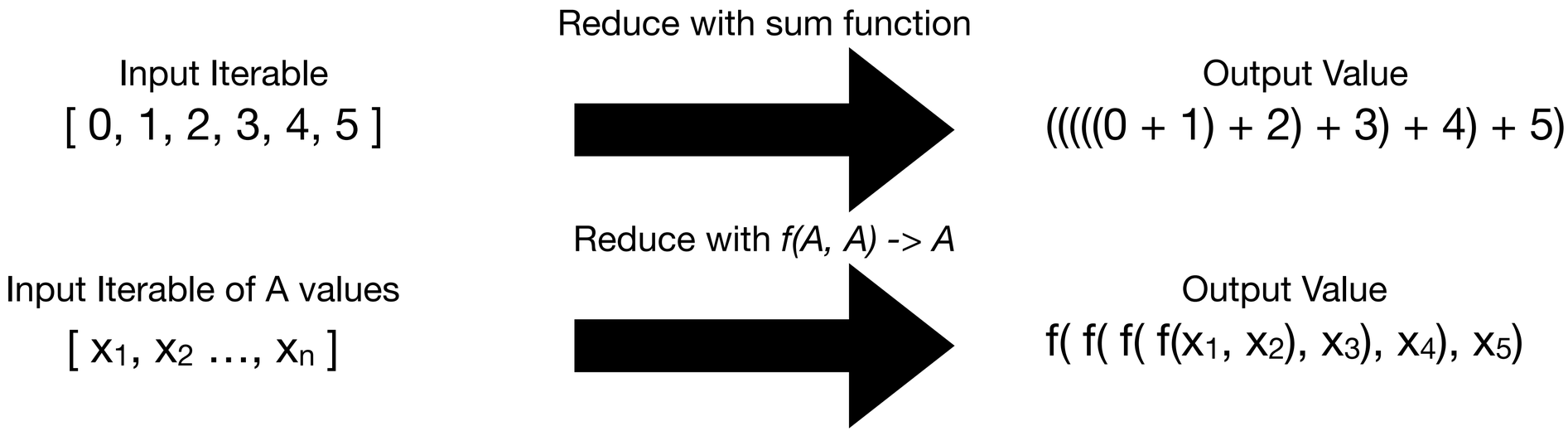
```
>>> numbers_iterable = range(10)
>>> list(numbers_iterable)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> print(list(odd_numbers_iterator))
[1, 3, 5, 7, 9]
```

```
>>> odd_numbers_iterator = filter(is_odd, numbers_iterable)
>>> odd_numbers_iterator
<filter object at 0x10343d7d0>
```

reduce - Built-in Functions

- Combines elements of an iterator into a single value using a function

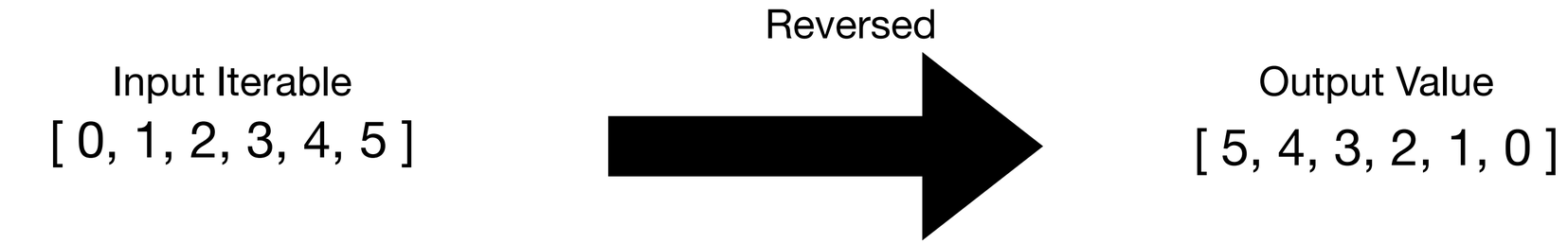


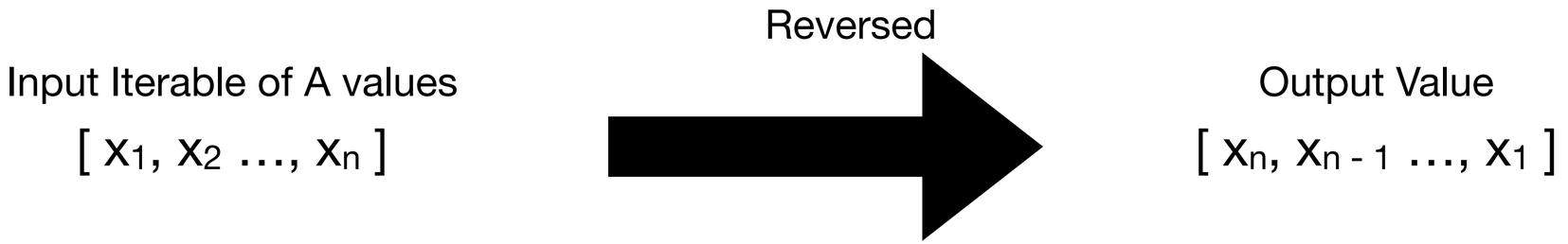
builtins.py

```
# Sum of numbers using reduce
assert sum(range(10)) == reduce(lambda x, y: x + y, range(10))

# Product of numbers
product = lambda sequence: reduce(lambda x, y: x * y, sequence)
assert product([1, 2, 3, 4]) == 1 * 2 * 3 * 4

# Comma separate strings
comma_separate = lambda sequence:
assert comma_separate(range(4)) == '0, 1, 2, 3'
```

reversed - Built-in Functions

- Makes an iterator in reverse order
- May have to evaluate entire input iterator and store results
  - Can be lazily evaluated if iterator supports the sequence protocol





Final Exam

- The sum of the squares of positive odd integers less than 16

```
q0 =    reduce(lambda x, y: x + y,
            map(lambda x: x * x,
                filter(lambda x: x % 2 == 1, range(16))))
```

- The product of odd squares of positive integers less than 16

```
q1 =    reduce(lambda x, y: x * y,
            filter(lambda x: x % 2 == 0,
                map(lambda x: x * x, range(16))))
```

- Iterate for the words of a given sentence that start with vowels

```
def vowel_words(sentence):
    split = sentence.split(' ')
    for word in split:
        if word[0] in 'aeiou':
            yield word
```

- Euclidian distance of vectors v, w of dimension N

```
euclidian_distance = lambda w, v: sum(((wi - vi)**2 for wi,vi in zip(w,v)))**.5
```

With great power comes great responsibility

Cleverness Can Hinder Clarity

- Readability should take precedence over code length
- Using iterators to write bad code will make me sad

oneliners.py

```
qsort = lambda L: [] if L==[] else qsort([x for x in L[1:] if x< L[0]]) + L[0:1] + qsort([x for x in L[1:] if x>=L[0]])

isprime = lambda n: [i for i in range(1 ,n) if n % i == 0] == [1]
```

Further Reading

- code examples made for this lesson
- python iterator documentation
- itertools module documentation
- functools module documentation
- google “python iterators”
- google “python functional programming”