Iterators
Class outline:

- Iterators
- For loops with iterators
- Built-in functions for iterators
# Iterators

An **iterator** is an object that provides sequential access to values, one by one.

- `iter(iterable)` returns an iterator over the elements of an iterable.
- `next(iterator)` returns the next element in an iterator.

```python
toppings = ["pineapple", "pepper", "mushroom", "roasted red pepper"]

topperator = iter(toppings)
next(iter)
next(iter)
next(iter)
next(iter)
next(iter)
next(iter)
next(iter)
next(iter)
```
Iterators

An **iterator** is an object that provides sequential access to values, one by one.

- `iter(iterable)` returns an iterator over the elements of an iterable.
- `next(iterator)` returns the next element in an iterator.

```python
toppings = ['pineapple', 'pepper', 'mushroom', 'roasted red pepper']

topperator = iter(toppings)
next(iter) # 'pineapple'
next(iter) # 'pepper'
next(iter) # 'mushroom'
next(iter) # 'roasted red pepper'
next(iter)
```
Iterators

An **iterator** is an object that provides sequential access to values, one by one.

- `iter(iterable)` returns an iterator over the elements of an iterable.
- `next(iterator)` returns the next element in an iterator.

```python
toppings = ['pineapple', 'pepper', 'mushroom', 'roasted red pepper']

topperator = iter(toppings)
next(iter)  # 'pineapple'
next(iter)  # 'pepper'
next(iter)  # 'mushroom'
next(iter)  # 'roasted red pepper'
next(iter)  # ❌ StopIteration exception
```
A useful detail

Calling `iter()` on an iterator just returns the iterator:

```python
numbers = ["一つ", "二つ", "三つ"]
num_iter = iter(numbers)
num_iter2 = iter(num_iter)

assert num_iter is num_iter2
```
Iterables

Lists, tuples, dictionaries, strings, and ranges are all iterable objects.

```python
my_order = ["Yuca Shepherds Pie", "Pão de queijo", "Guaraná"]

ranked_chocolates = ("Dark", "Milk", "White")

best_topping = "pineapple"

scores = range(1, 21)

prices = {"pineapple": 9.99, "pen": 2.99, "pineapple-pen": 19.99}
```
Making iterators for iterables

`iter()` can return an iterator for any iterable object.

```python
my_order = ["Yuca Shepherds Pie", "Pão de queijo", "Guaraná"]
order_iter = iter(order)
next(order_iter)

ranked_chocolates = ("Dark", "Milk", "White")
chocolate_iter = iter(ranked_chocolates)
next(chocolate_iter)

best_topping = "pineapple"
topping_iter = iter(best_topping)
next(topping_iter)

scores = range(1, 21)
``score_iter = iter(scores)
next(score_iter)```
Making iterators for iterables

`iter()` can return an iterator for any iterable object.

```python
my_order = ["Yuca Shepherds Pie", "Pão de queijo", "Guaraná"]
order_iter = iter(order)
next(order_iter)  # "Yuca Shepherds Pie"

ranked_chocolates = ("Dark", "Milk", "White")
chocolate_iter = iter(ranked_chocolates)
next(chocolate_iter)

best_topping = "pineapple"
topping_iter = iter(best_topping)
next(topping_iter)

scores = range(1, 21)
score_iter = iter(scores)
next(score_iter)
```
iter() can return an iterator for any iterable object.

```python
my_order = ['Yuca Shepherds Pie', 'Pão de queijo', 'Guaraná']
order_iter = iter(my_order)
next(order_iter)  # "Yuca Shepherds Pie"

ranked_chocolates = ('Dark', 'Milk', 'White')
chocolate_iter = iter(ranked_chocolates)
next(chocolate_iter)  # "Dark"

best_topping = "pineapple"
topping_iter = iter(best_topping)
next(topping_iter)

scores = range(1, 21)
score_iter = iter(scores)
next(score_iter)
```
Making iterators for iterables

`iter()` can return an iterator for any iterable object.

```python
my_order = ["Yuca Shepherds Pie", "Pão de queijo", "Guaraná"]
order_iter = iter(order)
next(order_iter)  # "Yuca Shepherds Pie"

ranked_chocolates = ("Dark", "Milk", "White")
chocolate_iter = iter(ranked_chocolates)
next(chocolate_iter)  # "Dark"

best_topping = "pineapple"
topping_iter = iter(best_topping)
next(topping_iter)  # "p"

scores = range(1, 21)
score_iter = iter(scores)
next(score_iter)
```
Making iterators for iterables

`iter()` can return an iterator for any iterable object.

```python
my_order = ['Yuca Shepherds Pie', 'Pão de queijo', 'Guaraná']
order_iter = iter(order)
next(order_iter)  # "Yuca Shepherds Pie"

ranked_chocolates = ('Dark', 'Milk', 'White')
chocolate_iter = iter(ranked_chocolates)
next(chocolate_iter)  # "Dark"

best_topping = 'pineapple'
topping_iter = iter(best_topping)
next(topping_iter)  # "p"

scores = range(1, 21)
score_iter = iter(scores)
next(score_iter)  # 1
```
Making iterators for dictionaries

In Python 3.6+, items in a dict are ordered according to when they were added.

```python
prices = {"pineapple": 9.99, "pen": 2.99, "pineapple-pen": 19.99}
```

An iterator for the keys:

```python
price_iter = iter(prices.keys())
next(price_iter)
```

An iterator for the values:

```python
price_iter = iter(prices.values())
next(price_iter)
```

An iterator for key/value tuples:

```python
price_iter = iter(prices.items())
next(price_iter)
```
Making iterators for dictionaries

In Python 3.6+, items in a dict are ordered according to when they were added.

```
prices = {"pineapple": 9.99, "pen": 2.99, "pineapple-pen": 19.99}
```

An iterator for the keys:

```
price_iter = iter(prices.keys())
next(price_iter)  # "pineapple"
```

An iterator for the values:

```
price_iter = iter(prices.values())
next(price_iter)
```

An iterator for key/value tuples:

```
price_iter = iter(prices.items())
next(price_iter)
```
Making iterators for dictionaries

In Python 3.6+, items in a dict are ordered according to when they were added.

```python
prices = {
    "pineapple": 9.99,
    "pen": 2.99,
    "pineapple-pen": 19.99
}

An iterator for the keys:

```python
price_iter = iter(prices.keys())
next(price_iter)  # "pineapple"
```

An iterator for the values:

```python
price_iter = iter(prices.values())
next(price_iter)  # 9.99
```

An iterator for key/value tuples:

```python
price_iter = iter(prices.items())
next(price_iter)
```
Making iterators for dictionaries

In Python 3.6+, items in a dict are ordered according to when they were added.

```python
prices = {"pineapple": 9.99, "pen": 2.99, "pineapple-pen": 19.99}
```

An iterator for the keys:

```python
price_iter = iter(prices.keys())
next(price_iter)  # "pineapple"
```

An iterator for the values:

```python
price_iter = iter(prices.values())
next(price_iter)  # 9.99
```

An iterator for key/value tuples:

```python
price_iter = iter(prices.items())
next(price_iter)  # ("pineapple", 9.99)
```
For loops
For loop with iterator

When used in a for loop, Python will call `next()` on the iterator in each iteration:

```python
ums = range(1, 4)
um_iter = iter(nums)
for num in nums:
    print(num)
```
For loops with used-up iterators

```python
nums = range(1, 4)
num_iter = iter(nums)
first = next(num_iter)

for num in num_iter:
    print(num)
```
For loops with used-up iterators

```python
nums = range(1, 4)
num_iter = iter(nums)
first = next(num_iter)

for num in num_iter:
    print(num)
```

Iterators are mutable! Once the iterator moves forward, it won't return the values that came before.

```python
nums = range(1, 4)
sum = 0
num_iter = iter(nums)

for num in num_iter:
    print(num)
for num in num_iter:
    sum += num
```
Iterating over iterables

If you want all the items from start to finish, it's better to use a for-in loop.

```python
my_order = ["Yuca Shepherds Pie", "Pão de queijo", "Guaraná"]
for item in my_order:
    print(item)
lowered = [item.lower() for item in my_order]

ranked_chocolates = ("Dark", "Milk", "White")
for chocolate in ranked_chocolates:
    print(chocolate)

prices = {"pineapple": 9.99, "pen": 2.99, "pineapple-pen": 19.99}
for product in prices:
    print(product, " costs ", prices[product])
discounted = {item: prices[item] * 0.75 for item in prices}

best_topping = "pineapple"
for letter in best_topping:
    print(letter)
```
Useful built-in functions
## Functions that return iterables

<table>
<thead>
<tr>
<th><strong>Function</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>list(iterable)</code></td>
<td>Returns a list containing all items in <code>iterable</code></td>
</tr>
<tr>
<td><code>tuple(iterable)</code></td>
<td>Returns a tuple containing all items in <code>iterable</code></td>
</tr>
<tr>
<td><code>sorted(iterable)</code></td>
<td>Returns a sorted list containing all items in <code>iterable</code></td>
</tr>
</tbody>
</table>
# Functions that return iterators

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>reversed(sequence)</code></td>
<td>Iterate over item in sequence in reverse order</td>
</tr>
<tr>
<td>(See example in PythonTutor)</td>
<td></td>
</tr>
<tr>
<td><code>zip(*iterables)</code></td>
<td>Iterate over co-indexed tuples with elements from each of the iterables</td>
</tr>
<tr>
<td>(See example in PythonTutor)</td>
<td></td>
</tr>
<tr>
<td><code>map(func, iterable, ...)</code></td>
<td>Iterate over func(x) for x in iterable</td>
</tr>
<tr>
<td>Same as [func(x) for x in iterable]</td>
<td></td>
</tr>
<tr>
<td>(See example in PythonTutor)</td>
<td></td>
</tr>
<tr>
<td><code>filter(func, iterable)</code></td>
<td>Iterate over x in iterable if func(x)</td>
</tr>
<tr>
<td>Same as [x for x in iterable if func(x)]</td>
<td></td>
</tr>
<tr>
<td>(See example in PythonTutor)</td>
<td></td>
</tr>
</tbody>
</table>
Built-in map function

\(\text{map}(\text{func}, \text{iterable}): \text{Applies } \text{func}(x) \text{ for } x \text{ in } \text{iterable} \) and returns an \text{iterator}

```python
def double(num):
    return num * 2

for num in map(double, [1, 2, 3]):
    print(num)

for word in map(lambda text: text.lower(), ["SuP", "HELLO", "Hi"])�:
    print(word)```
Built-in map function

\[
\text{map(func, iterable): Applies func(x) for x in iterable and returns an iterator}
\]

def double(num):
    return num * 2

for num in map(double, [1, 2, 3]):
    print(num)

for word in map(lambda text: text.lower(), ["SuP", "HELLO", "Hi"]:)
    print(word)

Turn the iterator into a list using list()

doubled = list(map(double, [1, 2, 3]))

lowered = list(map(lambda text: text.lower(), ["SuP", "HELLO", "Hi"]))
Exercise: Termified

Let's implement this without using a list comprehension.

```python
def termified(n, term):
    """Returns every the result of calling TERM
    on each element in the range from 0 to N (inclusive).
    """

    >>> termified(5, lambda x: 2 ** x)
    [1, 2, 4, 8, 16, 32]
    """
```
Exercise: Termified (solution)

Using map:

```python
def termified(n, term):
    """Returns every the result of calling TERM on each element in the range from 0 to N (inclusive).
    """
    return list(map(term, range(n + 1)))

>>> termified(5, lambda x: 2 ** x)
[1, 2, 4, 8, 16, 32]
"""
```
Exercise: Termified (solution)

Using map:

```python
def termified(n, term):
    """Returns every the result of calling TERM on each element in the range from 0 to N (inclusive).
    """
    return list(map(term, range(n + 1)))

>>> termified(5, lambda x: 2 ** x)
[1, 2, 4, 8, 16, 32]
"""```

Compare to list comprehension version:

```python
def termified(n, term):
    return [term(x) for x in range(n + 1)]
```
**Built-in filter function**

**filter(func, iterable)**: Returns an iterator from the items of **iterable** where **func(item)** is true.

```python
def is_fourletterword(text):
    return len(text) == 4

for word in filter(is_fourletterword, ['braid', 'bode', 'brand', 'band']):
    print(word)

for num in filter(lambda x: x % 2 == 0, [1, 2, 3, 4]):
    print(num)
```
Built-in filter function

**filter(func, iterable)**: Returns an iterator from the items of **iterable** where **func(item)** is true.

```python
def is_fourletterword(text):
    return len(text) == 4

for word in filter(is_fourletterword, ['braid', 'bode', 'brand', 'band']):
    print(word)

for num in filter(lambda x: x % 2 == 0, [1, 2, 3, 4]):
    print(num)
```

Turn the iterator into a list using **list()**

```python
filtered = list(is_fourletterword, ['braid', 'bode', 'brand', 'band'])

evens = list(filter(lambda x: x % 2 == 0, [1, 2, 3, 4]))
```
Exercise: Divisors

Let's implement this without using a list comprehension.

```python
def divisors(n):
    """Returns all the divisors of N."

    >>> divisors(12)
    [1, 2, 3, 4, 6]
    """
```
Exercise: Divisors (solution)

Using filter:

def divisors(n):
    """Returns all the divisors of N."

    >>> divisors(12)
    [1, 2, 3, 4, 6]
    """
    return list(filter(lambda x: n % x == 0, range(1, n)))
Exercise: Divisors (solution)

Using filter:

```python
def divisors(n):
    """Returns all the divisors of N."

    >>> divisors(12)
    [1, 2, 3, 4, 6]
    """
    return list(filter(lambda x: n % x == 0, range(1, n)))
```

Compare to list comprehension version:

```python
def divisors(n):
    return [x for x in range(1, n) if n % x == 0]
```
Built-in zip function

```
zip(*iterables): Returns an iterator that aggregates elements from each of the iterables into co-indexed pairs
```

# From:
["one", "two", "three"]
["uno", "dos", "tres"]
Built-in zip function

**zip(**iterables**):** Returns an **iterator** that aggregates elements from each of the **iterables** into co-indexed pairs

<table>
<thead>
<tr>
<th># From:</th>
<th># To:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[&quot;one&quot;, &quot;two&quot;, &quot;three&quot;]</td>
<td>(&quot;one&quot;, &quot;uno&quot;) (&quot;two&quot;, &quot;dos&quot;) (&quot;three&quot;, &quot;tres&quot;)</td>
</tr>
<tr>
<td>[&quot;uno&quot;, &quot;dos&quot;, &quot;tres&quot;]</td>
<td></td>
</tr>
</tbody>
</table>
Built-in zip function

`zip(*iterables)`: Returns an iterator that aggregates elements from each of the iterables into co-indexed pairs

```
# From:
[
  "one",
  "two",
  "three"
]

# To:

->

(
  "one",
  "uno"
)

(
  "two",
  "dos"
)

(
  "three",
  "tres"
)

```

```python
english_nums = ["one", "two", "three"]
spanish_nums = ["uno", "dos", "tres"]

zip_iter = zip(english_nums, spanish_nums)
english, spanish = next(zip_iter)
print(english, spanish)

for english, spanish in zip(english_nums, spanish_nums):
    print(english, spanish)

Turn the iterator into a list using `list()`

zipped = list(zip(english_nums, spanish_nums))
```
Exercise: matches

List comprehensions are allowed for this one...

```python
def matches(a, b):
    """Return the number of values k such that A[k] == B[k]."
    >>> matches([1, 2, 3, 4, 5], [3, 2, 3, 0, 5])
    3
    >>> matches("abdomens", "indolence")
    4
    >>> matches("abcd", "dcba")
    0
    >>> matches("abcde", "edcba")
    1
    >>> matches("abcdefg", "edcba")
    1
    ""
```
def matches(a, b):
    """Return the number of values k such that A[k] == B[k]."
    >>> matches([1, 2, 3, 4, 5], [3, 2, 3, 0, 5])
    3
    >>> matches("abdomens", "indolence")
    4
    >>> matches("abcd", "dcba")
    0
    >>> matches("abcde", "edcba")
    1
    >>> matches("abcdefg", "edcba")
    1
    """
    return sum([1 for a, b in zip(a, b) if a == b])
def list_o_lists(n):
    """Assuming N >= 0, return the list consisting of N lists:
    [1], [1, 2], [1, 2, 3], ... [1, 2, ... N].
    >>> list_o_lists(0)
    []
    >>> list_o_lists(1)
    [[1]]
    >>> list_o_lists(5)
    [[1], [1, 2], [1, 2, 3], [1, 2, 3, 4], [1, 2, 3, 4, 5]]
    """
Exercise: List of lists (solution)

def list_o_lists(n):
    """Assuming N >= 0, return the list consisting of N lists:
    [1], [1, 2], [1, 2, 3], ... [1, 2, ... N].
    >>> list_o_lists(0)
    []
    >>> list_o_lists(1)
    [[1]]
    >>> list_o_lists(5)
    [[1], [1, 2], [1, 2, 3], [1, 2, 3, 4], [1, 2, 3, 4, 5]]
    """
    return [list(range(1, i + 1)) for i in range(1, n+1)]
def palindrome(s):
    """Return whether s is the same sequence backward and forward.
    >>> palindrome([3, 1, 4, 1, 5])
    False
    >>> palindrome([3, 1, 4, 1, 3])
    True
    >>> palindrome('seveneves')
    True
    >>> palindrome('seven eves')
    False
    """
def palindrome(s):
    """Return whether $s$ is the same sequence backward and forward.
    """

    >>> palindrome([3, 1, 4, 1, 5])
    False
    >>> palindrome([3, 1, 4, 1, 3])
    True
    >>> palindrome('seveneves')
    True
    >>> palindrome('seven eves')
    False
    """
    return all([a == b for a, b in zip(s, reversed(s))])
    # OR
    return list(s) == list(reversed(s))
Use cases for iterators
Reasons for using iterators

A code that processes an iterator using `iter()` or `next()` makes few assumptions about the data itself.

- Changing the data storage from a list to a tuple, map, or dict doesn't require rewriting code.
- Others are more likely to be able to use your code on their data.

An iterator **bundles together a sequence and a position** with the sequence in a single object.

- Passing that object to another function always retains its position.
- Ensures that each element of the sequence is only processed once.
- Limits the operations that can be performed to only calling `next()`.
Blackjack demo!
Python Project of The Day!
Mathematical Animation Engine

**Manim:** An open-source Python animation engine for explanatory math videos, first created by Grant Sanderson for 3Blue1Brown videos.

Check out the examples gallery, oooo!