Containers

Class outline:

- Lists
- Containment
- For statements
- Ranges
- List comprehensions
- String literals

Lists

Lists

A list is a container that holds a sequence of related pieces of information.

The shortest list is an empty list, just 2 square brackets:

```
members = []
```

Lists can hold any Python values, separated by commas:

```
members = ["Pamela", "Tinu", "Brenda", "Kaya"]

ages_of_kids = [1, 2, 7]

prices = [79.99, 49.99, 89.99]

digits = [2//2, 2+2+2+2, 2, 2*2*2]

remixed = ["Pamela", 7, 79.99, 2*2*2]
```

List length

Use the global `len()` function to find the length of a list.

```
attendees = ["Tammy", "Shonda", "Tina"]

print(len(attendees))

num_of_attendees = len(attendees)
print(num_of_attendees)
```

What could go wrong with storing the length?

List length

Use the global `len()` function to find the length of a list.

```
attendees = ["Tammy", "Shonda", "Tina"]

print(len(attendees))   #  3

num_of_attendees = len(attendees)
print(num_of_attendees)
```

What could go wrong with storing the length?

Accessing list items (brackets)

Each list item has an index, starting from 0.

```
letters = ['A', 'B', 'C']
# Index:   0     1     2
```

Access each item by putting the index in brackets:

```
letters[0]
letters[1]
letters[2]
letters[3]
```

```
curr_ind = 1
letters[curr_ind]
```

Accessing list items (brackets)

Each list item has an index, starting from 0.

```
letters = ['A', 'B', 'C']
# Index:   0     1     2
```

Access each item by putting the index in brackets:

```
letters[0]  # 'A'
letters[1]  # 'B'
letters[2]  # 'C'
letters[3]
```

```
curr_ind = 1
letters[curr_ind] # 'B'
```

Accessing list items (brackets)

Each list item has an index, starting from 0.

```
letters = ['A', 'B', 'C']
# Index:   0     1     2
```

Access each item by putting the index in brackets:

```
letters[0]  # 'A'
letters[1]  # 'B'
letters[2]  # 'C'
letters[3]  #  Error!
```

```
curr_ind = 1
letters[curr_ind] # 'B'
```

Accessing list items (brackets)

Each list item has an index, starting from 0.

```
letters = ['A', 'B', 'C']
# Index:   0     1     2
```

Access each item by putting the index in brackets:

```
letters[0]  # 'A'
letters[1]  # 'B'
letters[2]  # 'C'
letters[3]  #  Error!
```

```
curr_ind = 1
letters[curr_ind] # 'B'
```

Accessing list items (brackets)

Each list item has an index, starting from 0.

```
letters = ['A', 'B', 'C']
# Index:   0     1     2
```

Access each item by putting the index in brackets:

```
letters[0]  # 'A'
letters[1]  # 'B'
letters[2]  # 'C'
letters[3]  #  Error!
```

```
curr_ind = 1
letters[curr_ind] # 'B'
```

Negative indices are also possible:

```
letters[-1]  # 'C'
letters[-2]  # 'B'
letters[-4]  #  Error!
```

Accessing list items (function)

It's also possible to use a function from the operator module:

```
from operator import getitem

getitem(letters, 0)
```

List concatenation

Add two lists together using the `+` operator:

```
boba_prices = [5.50, 6.50, 7.50]
smoothie_prices = [7.00, 7.50]
all_prices = boba_prices + smoothie_prices
```

Or the `add` function:

```
from operator import add

boba_prices = [5.50, 6.50, 7.50]
smoothie_prices = [7.00, 7.50]
all_prices = add(boba_prices, smoothie_prices)
```

List repetition

Concatenate the same list multiple times using the `*` operator:

```
boba_prices = [5.50, 6.50, 7.50]

more_boba = boba_prices * 3
```

Or the `mul` function:

```
from operator import mul

boba_prices = [5.50, 6.50, 7.50]
more_boba = mul(boba_prices, 3)
```

All together now:

```
digits = [1, 9, 8, 4]
together = [6, 2, 4] + digits * 2
```

List repetition

Concatenate the same list multiple times using the `*` operator:

```
boba_prices = [5.50, 6.50, 7.50]

more_boba = boba_prices * 3
```

Or the `mul` function:

```
from operator import mul

boba_prices = [5.50, 6.50, 7.50]
more_boba = mul(boba_prices, 3)
```

All together now:

```
digits = [1, 9, 8, 4]
together = [6, 2, 4] + digits * 2 # [6, 2, 4, 1, 9, 8, 4, 1, 9, 8,
```

Nested Lists

Since Python lists can contain any values, an item can itself be a list.

```
gymnasts = [ ["Brittany", 9.15, 9.4, 9.3, 9.2],
             ["Lea", 9, 8.8, 9.1, 9.5],
             ["Maya", 9.2, 8.7, 9.2, 8.8] ]
```

- What's the length of `gymnasts`?
- What's the length of `gymnasts[0]`?

Nested Lists

Since Python lists can contain any values, an item can itself be a list.

```
gymnasts = [ ["Brittany", 9.15, 9.4, 9.3, 9.2],
             ["Lea", 9, 8.8, 9.1, 9.5],
             ["Maya", 9.2, 8.7, 9.2, 8.8] ]
```

- What's the length of `gymnasts`? 3
- What's the length of `gymnasts[0]`?

Nested Lists

Since Python lists can contain any values, an item can itself be a list.

```
gymnasts = [ ["Brittany", 9.15, 9.4, 9.3, 9.2],
             ["Lea", 9, 8.8, 9.1, 9.5],
             ["Maya", 9.2, 8.7, 9.2, 8.8] ]
```

- What's the length of `gymnasts`? 3
- What's the length of `gymnasts[0]`? 5

Accessing nested list items

```
gymnasts = [
                ["Brittany", 9.15, 9.4, 9.3, 9.2],
                ["Lea", 9, 8.8, 9.1, 9.5],
                ["Maya", 9.2, 8.7, 9.2, 8.8]
            ]
```

Access using bracket notation, with more brackets as needed:

```
gymnasts[0]
gymnasts[0][0]
gymnasts[1][0]
gymnasts[1][4]
gymnasts[1][5]
gymnasts[3][0]
```

Accessing nested list items

```
gymnasts = [
                ["Brittany", 9.15, 9.4, 9.3, 9.2],
                ["Lea", 9, 8.8, 9.1, 9.5],
                ["Maya", 9.2, 8.7, 9.2, 8.8]
            ]
```

Access using bracket notation, with more brackets as needed:

```
gymnasts[0]     # ["Brittany", 9.15, 9.4, 9.3, 9.2]
gymnasts[0][0]
gymnasts[1][0]
gymnasts[1][4]
gymnasts[1][5]
gymnasts[3][0]
```

Accessing nested list items

```
gymnasts = [
                ["Brittany", 9.15, 9.4, 9.3, 9.2],
                ["Lea", 9, 8.8, 9.1, 9.5],
                ["Maya", 9.2, 8.7, 9.2, 8.8]
            ]
```

Access using bracket notation, with more brackets as needed:

```
gymnasts[0]     # ["Brittany", 9.15, 9.4, 9.3, 9.2]
gymnasts[0][0]  # "Brittany"
gymnasts[1][0]
gymnasts[1][4]
gymnasts[1][5]
gymnasts[3][0]
```

Accessing nested list items

```
gymnasts = [
                ["Brittany", 9.15, 9.4, 9.3, 9.2],
                ["Lea", 9, 8.8, 9.1, 9.5],
                ["Maya", 9.2, 8.7, 9.2, 8.8]
            ]
```

Access using bracket notation, with more brackets as needed:

```
gymnasts[0]     # ["Brittany", 9.15, 9.4, 9.3, 9.2]
gymnasts[0][0]  # "Brittany"
gymnasts[1][0]  # "Lea"
gymnasts[1][4]
gymnasts[1][5]
gymnasts[3][0]
```

Accessing nested list items

```
gymnasts = [
                ["Brittany", 9.15, 9.4, 9.3, 9.2],
                ["Lea", 9, 8.8, 9.1, 9.5],
                ["Maya", 9.2, 8.7, 9.2, 8.8]
            ]
```

Access using bracket notation, with more brackets as needed:

```
gymnasts[0]     # ["Brittany", 9.15, 9.4, 9.3, 9.2]
gymnasts[0][0]  # "Brittany"
gymnasts[1][0]  # "Lea"
gymnasts[1][4]  # 9.5
gymnasts[1][5]
gymnasts[3][0]
```

Accessing nested list items

```
gymnasts = [
                ["Brittany", 9.15, 9.4, 9.3, 9.2],
                ["Lea", 9, 8.8, 9.1, 9.5],
                ["Maya", 9.2, 8.7, 9.2, 8.8]
            ]
```

Access using bracket notation, with more brackets as needed:

```
gymnasts[0]     # ["Brittany", 9.15, 9.4, 9.3, 9.2]
gymnasts[0][0]  # "Brittany"
gymnasts[1][0]  # "Lea"
gymnasts[1][4]  # 9.5
gymnasts[1][5]  #  IndexError!
gymnasts[3][0]
```

Accessing nested list items

```
gymnasts = [
                ["Brittany", 9.15, 9.4, 9.3, 9.2],
                ["Lea", 9, 8.8, 9.1, 9.5],
                ["Maya", 9.2, 8.7, 9.2, 8.8]
            ]
```

Access using bracket notation, with more brackets as needed:

```
gymnasts[0]    # ["Brittany", 9.15, 9.4, 9.3, 9.2]
gymnasts[0][0] # "Brittany"
gymnasts[1][0] # "Lea"
gymnasts[1][4] # 9.5
gymnasts[1][5] #  IndexError!
gymnasts[3][0] #  IndexError!
```

Containment

Containment operator

Use the `in` operator to test if value is inside a container:

```
digits = [2, 8, 3, 1, 8, 5, 3, 0, 7, 1]

1 in digits

3 in digits

4 in digits

not (4 in digits)
```

Containment operator

Use the `in` operator to test if value is inside a container:

```
digits = [2, 8, 3, 1, 8, 5, 3, 0, 7, 1]

1 in digits # True

3 in digits # True

4 in digits # False

not (4 in digits) # True
```

For statements

For loop

The for loop syntax:

```
for <value> in <sequence>:
    <statement>
    <statement>
```

The for loop provides a cleaner way to write many `while` loops, as long as they are iterating over some sort of sequence.

```
def count(s, value):
    total = 0
    for element in s:
        if element == value:
            total = total + 1
    return total
```

For statement execution procedure

```
for <name> in <expression>:
    <suite>
```

1. Evaluate the header `<expression>`, which must yield an iterable value (a sequence)
2. For each element in that sequence, in order:
    1. Bind `<name>` to that element in the current frame
    2. Execute the `<suite>`

Looping through nested lists

```
gymnasts = [
                ["Brittany", 9.15, 9.4, 9.3, 9.2],
                ["Lea", 9, 8.8, 9.1, 9.5],
                ["Maya", 9.2, 8.7, 9.2, 8.8]
            ]
```

Use a nested `for-in` loop:

```
for gymnast in gymnasts:
    for data in gymnast:
        print(data, end="|")
```

Remember what type of data is being stored in the loop variable!

Sequence unpacking in for statements

```
pairs = [[1, 2], [2, 2], [3, 2], [4, 4]]
same_count = 0

for x, y in pairs:
    if x == y:
        same_count = same_count + 1
```

Each name is bound to a value, like in multiple assignment.

Ranges

The range type

A range represents a sequence of integers.

```
... -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5...
                range(-2, 3)
```

If just one argument, range starts at 0 and ends just before it:

```
for num in range(6):
    print(num)        # 0, 1, 2, 3, 4, 5
```

If two arguments, range starts at first and ends just before second:

```
for num in range(1, 6):
    print(num)        # 1, 2, 3, 4, 5
```

List comprehensions

List comprehension syntax

A way to create a new list by "mapping" an existing list.

Short version:

```
[<map exp> for <name> in <iter exp>]
```

```
odds = [1, 3, 5, 7, 9]
evens = [(num + 1) for num in odds]
```

List comprehension syntax

A way to create a new list by "mapping" an existing list.

Short version:

```
[<map exp> for <name> in <iter exp>]
```

```
odds = [1, 3, 5, 7, 9]
evens = [(num + 1) for num in odds]
```

Long version (with filter):

```
[<map exp> for <name> in <iter exp> if <filter exp>]
```

```
temps = [60, 65, 71, 67, 77, 89]
hot = [temp for temp in temps if temp > 70]
```

List comprehension execution procedure

```
[<map exp> for <name> in <iter exp> if <filter exp>]
```

- Add a new frame with the current frame as its parent
- Create an empty result list that is the value of the expression
- For each element in the iterable value of `<iter exp>`:
  - Bind `<name>` to that element in the new frame from step 1
  - If `<filter exp>` evaluates to a true value, then add the value of `<map exp>` to the result list

```
letters = ['a', 'b', 'c', 'd', 'e', 'f', 'm', 'n', 'o', 'p']
word = [letters[i] for i in [3, 4, 6, 8]]
```

 View in PythonTutor

Exercise: Divisors

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

    >>> divisors(12)
    [1, 2, 3, 4, 6]
    """
```

Exercise: Divisors (solution)

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

    >>> divisors(12)
    [1, 2, 3, 4, 6]
    """
    return [x for x in range(1, n) if n % x == 0]
```

Exercise: Frontloaded

```
def front(s, f):
    """Return S but with elements chosen by F at the front.

    >>> front(range(10), lambda x: x % 2 == 1)  # odds in front
    [1, 3, 5, 7, 9, 0, 2, 4, 6, 8]
    """
```

Exercise: Frontloaded (solution)

```
def front(s, f):
    """Return S but with elements chosen by F at the front.

    >>> front(range(10), lambda x: x % 2 == 1)  # odds in front
    [1, 3, 5, 7, 9, 0, 2, 4, 6, 8]
    """
    return [e for e in s if f(e)] + [e for e in s if not f(e)]
```

String literals

What's in a string?

Representing data:

```
'2,400' '2.400' '1.2e-5'
```

Representing language:

```
"""Se lembra quando a gente
Chegou um dia a acreditar
Que tudo era pra sempre
Sem saber
Que o pra sempre sempre acaba"""
```

Representing programs:

```
'curry = lambda f: lambda x: lambda y: f(x, y)'
```

String literals: 3 forms

Single quoted strings and double quoted strings are equivalent:

```
'您好, I am a string, hear me roar !'
"I've got an apostrophe"
```

Multi-line strings automatically insert new lines:

```
"""The Zen of Python
claims, Readability counts.
Read more: import this."""
# 'The Zen of Python\nclaims, Readability counts.\nRead more: impo
```

The `\n` is an **escape sequence** signifying a line feed.

Strings are similar to lists

```
alfabeto = 'abcdefghijklmnñopqrstuvwxyz'

len(alfabeto)  # 27

alfabeto[14] + "andu" # ñandu

alfabeto + ' ¡Ya conoces el ABC!'
```

Differences between strings & lists

A single-character string is the same as the character itself.

```
initial = 'P'
initial[0] == initial
```

The `in` operator will match substrings:

```
'W' in 'Where\'s Waldo'      # True
'Waldo' in  'Where\'s Waldo'
```

Differences between strings & lists

A single-character string is the same as the character itself.

```
initial = 'P'
initial[0] == initial  # True
```

The `in` operator will match substrings:

```
'W' in 'Where\'s Waldo'      # True
'Waldo' in  'Where\'s Waldo'
```

Differences between strings & lists

A single-character string is the same as the character itself.

```
initial = 'P'
initial[0] == initial  # True
```

The `in` operator will match substrings:

```
'W' in 'Where\'s Waldo'      # True
'Waldo' in  'Where\'s Waldo' # True
```