Revised: 1/2/2021

Sequence Types

Michael C. Hackett

Assistant Professor, Computer Science

Community College of Philadelphia

Lecture Topics

- Strings
  - Characters and Indexes
  - Length
  - Iterating Over String Characters
  - Replacing Parts of a String
- Lists
  - Retrieving and Changing Elements
  - Iterating Over a List
  - Adding to Lists
  - Deleting from Lists
  - Copying Lists
- Tuples
- Dictionaries
  - Adding a KVP
  - Updating a KVP
  - Deleting a KVP
  - Turning Two Lists into a Dictionary
- Sets
  - Adding to Sets
  - Deleting from Sets
  - Turning a List into a Set
  - Turning a Set into a List
  - Set Theory/Operations

Strings

- Strings are a sequence type and are comprised of characters.
  - Characters can be letters, numbers, symbols and whitespace.

- Every character in a string has an index.

```
example = "Example String"
```

E	x	a	m	p	l	e		S	t	r	i	n	g
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Characters and Indexes

- Characters in a string can be accessed using subscript notation.

```
example = "Example String"
first_character = example[0]
print(first_character)
print(example[8])
```

```
E
S
```

Characters and Indexes

- Strings are immutable.
  - Characters in a string cannot be changed.

```
example = "Example String"
example[13] = "G"
print(example)
```

```
Traceback (most recent call last):
  File "C:\testing\examples.py", line 8, in <module>
    example[13] = "G"
TypeError: 'str' object does not support item assignment
>>>
```

Characters and Indexes

- Attempting to access an index that does not exist will raise in an IndexError exception.

```
example = "Example String"
character = example[20]
print(character)
```

```
Traceback (most recent call last):
  File "C:\testing\examples.py", line 8, in <module>
    character = example[20]
IndexError: string index out of range
>>>
```

String Length

- A string's length is the total number of characters it contains.
  - Use Python's built-in len function to return the length of a string.

```
example = "Example String"
length = len(example)
print(length)
```

```
14
```

Iterating Over String Characters

- A for loop can iterate over a string's characters.
  - To loop through all or part of a string's characters:

```
example = "Example"
for index in range(0, len(example)) :
  print(example[index])
```

  - To loop through all of a string's characters:

```
example = "Example"
for character in example :
  print(character)
```

```
E
x
a
m
p
l
e
```

Replacing Parts of a String

- The string's replace function replaces part of a string with new data.
- Two arguments (both strings)- first is the string to find, second is what to replace it with. *CASE SENSITIVE*

```
orig_string = "Today is Monday."
new_string = orig_string.replace("Monday", "Tuesday")
print(new_string)
```

```
Today is Tuesday.
```

Note the value of orig_string **does not change**.
The replace method returns a new string with every sequence of the first argument replaced with the second argument.

Replacing Parts of a String

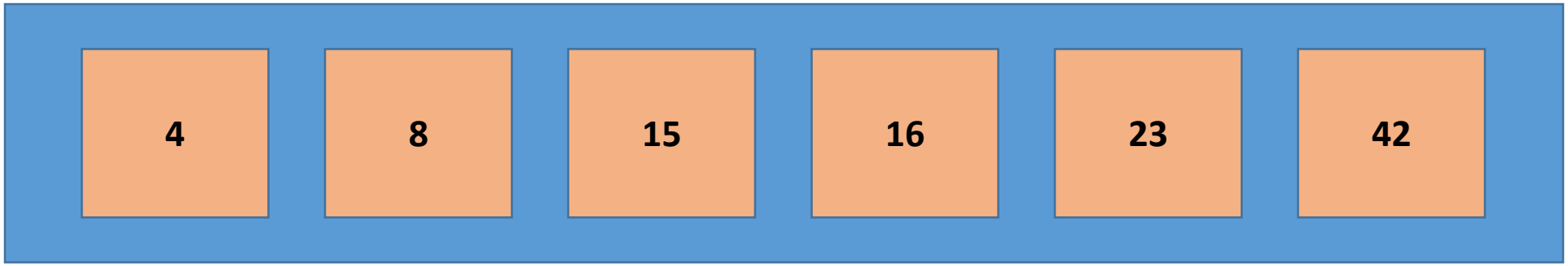
- The string's replace function replaces all matches.

```
orig_string = "Today is Monday."
new_string = orig_string.replace("day", "night")
print(new_string)
```

```
Tonight is Monnight.
```

Lists

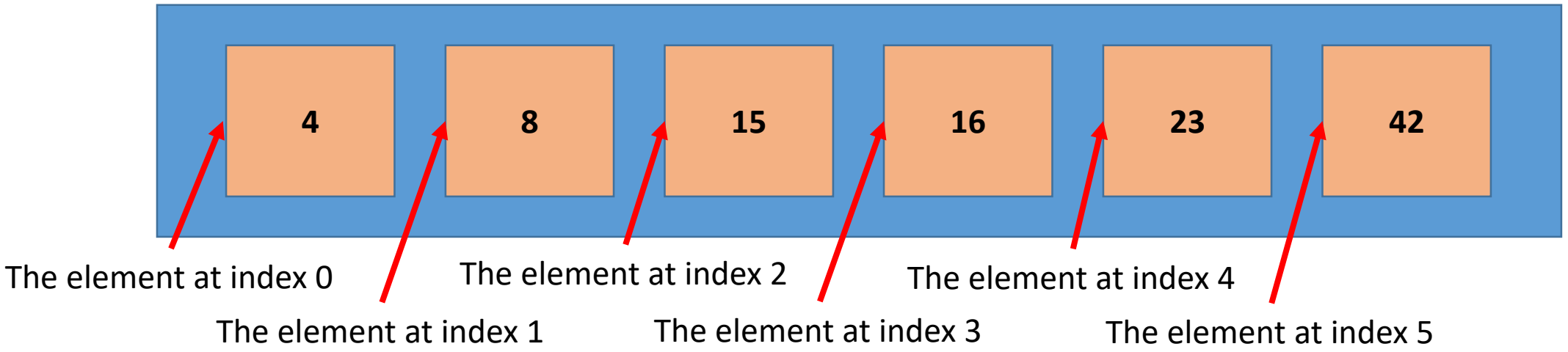
- An ***list*** is a sequence object that has multiple values.
  - Another way to look at it is a variable that has multiple values.



A list of ints

Lists

- An ***index*** (or ***subscript***) is the number representing the position of a list element.
  - First index is always zero.
  - The index is always an int.
- An ***element*** is the data or object referenced by an index.



Lists

- The elements are comma separated, enclosed in square brackets.

```
numbers = [4, 8, 15, 16, 23, 42]
values = [35.6, 32.76, 51.4]
pets = ["dog", "cat", "bird", "fish"]
```

- The data types of a list may vary.

```
mixed_values = [35.6, 15, "cat"]
```

- An empty list:

```
example = []
```

Lists

- When passed to the print function, the entire list is printed.
  - Includes commas and brackets.
  - Useful for testing/debugging.

```
numbers = [4, 8, 15, 16, 23, 42]
print(numbers)
```

```
[4, 8, 15, 16, 23, 42]
```

Lists

- A list's *length* is the total number of elements contained within it.
  - Python's built-in len function returns the length of a sequence data type.

```
pets = ["dog", "cat", "bird", "fish"]
length = len(pets)
print(length)
```

```
4
```

Lists

- Elements of a list are referenced using *subscript notation*.
  - Specify the index of the list's element.

```
numbers = [4, 8, 15, 16, 23, 42]
test_value1 = numbers[0]
print(test_value1)

test_value2 = numbers[4]
print(test_value2)

print(numbers[2])
```

```
4
23
15
```

Lists

- Lists are *mutable*, meaning the elements can be changed.
  - Specify the index of the list's element and assign to it a new value.

```
values = [35.6, 32.76, 51.4]
print(values[1])
values[1] = 27.21
print(values[1])
```

```
32.76
27.21
```

Lists

- Negative indexes retrieve elements relative to the end of the list.

```
numbers = [4, 8, 15, 16, 23, 42]
print(numbers[-1])
numbers[-4] = 100
print(numbers[-4])
print(numbers[2])
```

Length – 1 = 6 – 1 = **5**

Length – 4 = 6 – 4 = **2**

```
42
100
100
```

Lists

- An IndexError exception will be raised if you try to access an index that does not exist.

```
numbers = [4, 8, 15, 16, 23, 42]
print(numbers[10])
```

```
Traceback (most recent call last):
  File "C:\testing\examples.py", line 8, in <module>
    print(numbers[10])
IndexError: list index out of range
>>>
```

Lists

- For loops can iterate over the values of a list.

```
numbers = [4, 8, 15, 16, 23, 42]
for number in numbers :
  print(number)

print()

pets = ["dog", "cat", "bird", "fish"]
for animal in pets :
  print(animal)
```

```
4
8
15
16
23
42

dog
cat
bird
fish
```

Lists

- For loops (using the range function) can iterate over the entire list or a segment of the list.

```
numbers = [4, 8, 15, 16, 23, 42]
for i in range(0, 3) :
  print(numbers[i])

print()

pets = ["dog", "cat", "bird", "fish"]
for i in range(1, len(pets)) :
  print(pets[i])
```

```
4
8
15

cat
bird
fish
```

Adding to Lists

- Values can be added/concatenated to a list using the addition operator ONLY if the values are in list form.

```
numbers = [4, 8, 15, 16, 23, 42]
numbers = numbers + 100
print(numbers)
```

Error

```
numbers = [4, 8, 15, 16, 23, 42]
numbers = numbers + [100]
print(numbers)
```

```
[4, 8, 15, 16, 23, 42, 100]
```

Adding to Lists

- Two lists are merged/concatenated together when combined using the addition operator.

```
numbers = [4, 8, 15, 16, 23, 42]
numbers = numbers + [100, 101, 102]
print(numbers)
```

```
[4, 8, 15, 16, 23, 42, 100, 101, 102]
```

Adding to Lists

- Another way to add values to a list is with the addition combined assignment operator.

```
numbers = [4, 8, 15, 16, 23, 42]
numbers += [100, 101, 102]
print(numbers)
```

```
[4, 8, 15, 16, 23, 42, 100, 101, 102]
```

Adding to Lists

- A list's append function can add a single element to the end of a list.

```
numbers = [4, 8, 15, 16, 23, 42]
numbers.append(100)
print(numbers)
```

```
[4, 8, 15, 16, 23, 42, 100]
```

Adding to Lists

- A list's append function can only add a single element to the end of a list.
  - Does not have to be in list form.

```
numbers = [4, 8, 15, 16, 23, 42]
numbers.append(100)
print(numbers)
```

- Concatenating data to the end of a list using the addition/combined assignment operator can be used to add one or multiple elements.
  - Must be in list form.

```
numbers = [4, 8, 15, 16, 23, 42]
numbers = numbers + [100]
print(numbers)
```

```
numbers = [4, 8, 15, 16, 23, 42]
numbers += [100, 101, 102]
print(numbers)
```

Adding to Lists

- A list's insert function places a value at a specified index.
  - The existing elements are shifted over to make room.
  - First argument is the index.
    - If the specified index is beyond the length of the list, the value will be inserted at the end of the list.
  - Second argument is the value to insert.

```
numbers = [10, 20, 40, 50]
numbers.insert(2, 30)
print(numbers)
```

```
[10, 20, 30, 40, 50]
```

Deleting from Lists

- To remove an element by index, use the **del** (delete) keyword to remove it.
  - Any subsequent elements will be shifted over.

```
numbers = [4, 8, 15, 16, 23, 42]
del numbers[3]
print(numbers)
```

```
[4, 8, 15, 23, 42]
```

Deleting from Lists

- To remove an element by value, the list's remove function will delete the element.
  - Only removes the first match.
  - Case-sensitive.

```
pets = ["dog", "cat", "bird", "cat", "fish"]
pets.remove("cat")
print(pets)
```

```
["dog", "bird", "cat", "fish"]
```

Deleting from Lists

- If the element is not found, a ValueError exception will be raised.

```
pets = ["dog", "cat", "bird", "cat", "fish"]
pets.remove("CAT")
print(pets)
```

```
Traceback (most recent call last):
  File "C:\testing\examples.py", line 9, in <module>
    pets.remove("CAT")
ValueError: list.remove(x): x not in list
>>>
```

Deleting from Lists

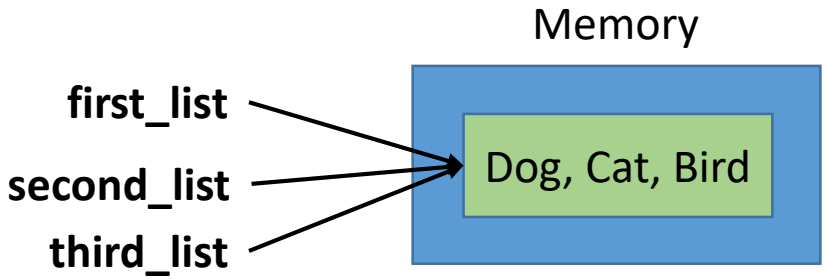
- The value to delete must be known in order to use the list's remove function.
  - May raise a ValueError exception.

- When deleting using the del keyword, only the index must be known.
  - May raise an IndexError exception if the index does not exist.

Copying Lists

- Copying a list like the example below creates a ***shallow copy.***
  - Shallow copies are multiple variables referencing the same data.

```
first_list = ["Dog", "Cat", "Bird"]
second_list = first_list
third_list = first_list
```



Copying Lists

- Since the variables reference the same list, changing one appears to change any others.

```
first_list = ["Dog", "Cat", "Bird"]
second_list = first_list
print(first_list[0])                Dog

second_list[0] = "Fish"
print(first_list[0])                Fish
```

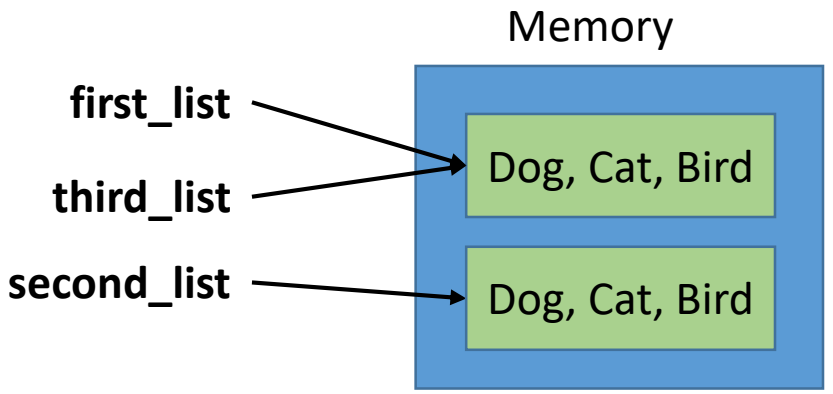
Copying Lists

- The **is** keyword tests if two variables reference the same object.
  - In other words, the is keyword will determine if two list variables are shallow copies.

```
first_list = ["Dog", "Cat", "Bird"]
second_list = ["Dog", "Cat", "Bird"]
third_list = first_list #Shallow Copy

if first_list is third_list :
  print("These lists are shallow copies")

if first_list is second_list :
  print("These lists are shallow copies")
```



Copying Lists

- To create a second, separate list with the same contents you need to perform a ***deep copy***.
  - A deep copy copies the contents of one list into a second list.

```
original = [3, 5, 7, 9]
copy = []
```

Empty List

```
for element in original :
  copy.append(element)
```

Deep Copies

- Since the variables reference different lists, changing one does not alter the original.

```
original = [3, 5, 7, 9]
copy = []

for element in original :
  copy.append(element)

print(original[0])                3
copy[0] = 99
print(original[0])                3
```

Copying Lists

- An alternative, simpler way to deep copy a list.
  - Concatenate the original list with an empty list.

```
original = [3, 5, 7, 9]
copy = [] + original

for element in copy :
  print(element)
```

```
3
5
7
9
```

Tuples

- A ***tuple*** is a sequence type and is very much like a list, however tuples are immutable.
  - The elements in a tuple cannot be changed.

- Tuples contain comma separated values in parentheses.
  - The elements/values can be of different types.

```
numbers = (4, 8, 15, 16, 23, 42)
values = (35.6, 32.76, 51.4)
pets = ("dog", "cat", "bird", "fish")
mixed_values = (35.6, 15, "cat")
```

Tuples

- Tuples that contain only one element must include a trailing comma.
  - Python interpreter treats the parentheses as part of an arithmetic expression:

```
example = (4)  #Creates an int
```

  - Python interpreter treats the parentheses as part of a tuple:

```
example = (4,) #Creates a tuple
```

Tuples

- A tuple's length, like a list, is the total number of elements contained within it.
  - Python's built-in len function returns the length of a sequence data type.

```
pets = ("dog", "cat", "bird", "fish")
length = len(pets)
print(length)
```

```
4
```

Tuples

- Elements of a tuple are referenced using subscript notation.
  - Specify the index of the tuple's element.

```
numbers = (4, 8, 15, 16, 23, 42)
test_value1 = numbers[0]
print(test_value1)

test_value2 = numbers[4]
print(test_value2)

print(numbers[2])
```

```
4
23
15
```

Tuples

- An IndexError exception will be raised if you try to access an index that does not exist.

```
numbers = (4, 8, 15, 16, 23, 42)
print(numbers[10])
```

```
Traceback (most recent call last):
  File "C:\testing\examples.py", line 14, in <module>
    print(numbers[10])
IndexError: tuple index out of range
>>>
```

Tuples

- Negative indexes retrieve elements relative to the end of the tuple.

```
numbers = (4, 8, 15, 16, 23, 42)
print(numbers[-1])
print(numbers[-4])
print(numbers[2])
```

Length – 1 = 6 – 1 = **5**

Length – 4 = 6 – 4 = **2**

```
42
15
15
```

Tuples

- When passed to the print function, the entire tuple is printed.
  - Includes commas and parentheses.
  - Useful for testing/debugging.

```
numbers = (4, 8, 15, 16, 23, 42)
print(numbers)
```

```
(4, 8, 15, 16, 23, 42)
```

Tuples

- For loops can iterate over the values of a tuple.

```
numbers = (4, 8, 15, 16, 23, 42)
for number in numbers :
  print(number)

print()

pets = ("dog", "cat", "bird", "fish")
for animal in pets :
  print(animal)
```

```
4
8
15
16
23
42

dog
cat
bird
fish
```

Tuples

- For loops can iterate over the values of a tuple.

```
numbers = (4, 8, 15, 16, 23, 42)
for i in range(0, 3) :
  print(numbers[i])

print()

pets = ("dog", "cat", "bird", "fish")
for i in range(1, len(pets)) :
  print(pets[i])
```

```
4
8
15

cat
bird
fish
```

Tuples

- While elements of a tuple cannot be changed and new elements cannot be appended to a tuple, tuples *can* be concatenated together.

```
numbers = (4, 8, 15, 16, 23, 42)
numbers = numbers + 100
print(numbers)
```


```
numbers = (4, 8, 15, 16, 23, 42)
numbers = numbers + (100)
print(numbers)
```

```
(4, 8, 15, 16, 23, 42, 100)
```

Tuples

- Two tuples are concatenated together when combined using the addition operator.

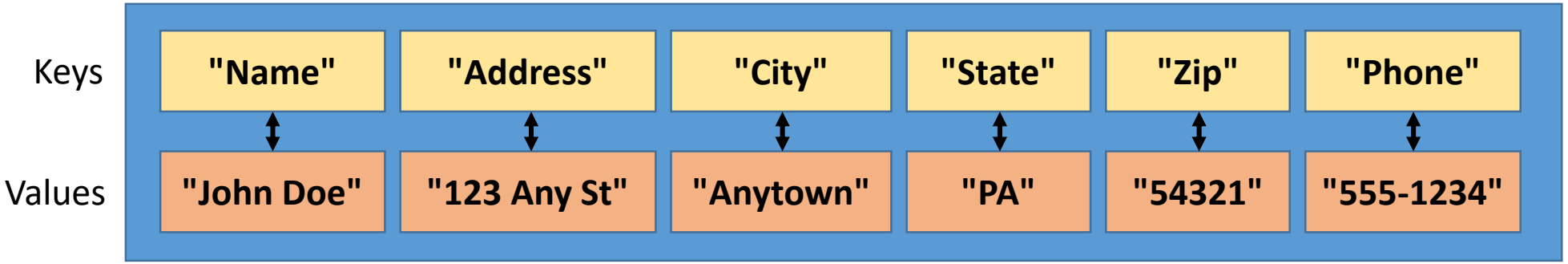
```
numbers = (4, 8, 15, 16, 23, 42)
numbers = numbers + (100, 101, 102)
print(numbers)
```

```
(4, 8, 15, 16, 23, 42, 100, 101, 102)
```

Dictionaries

- A ***dictionary*** is a mapping object that contains a collection of data.

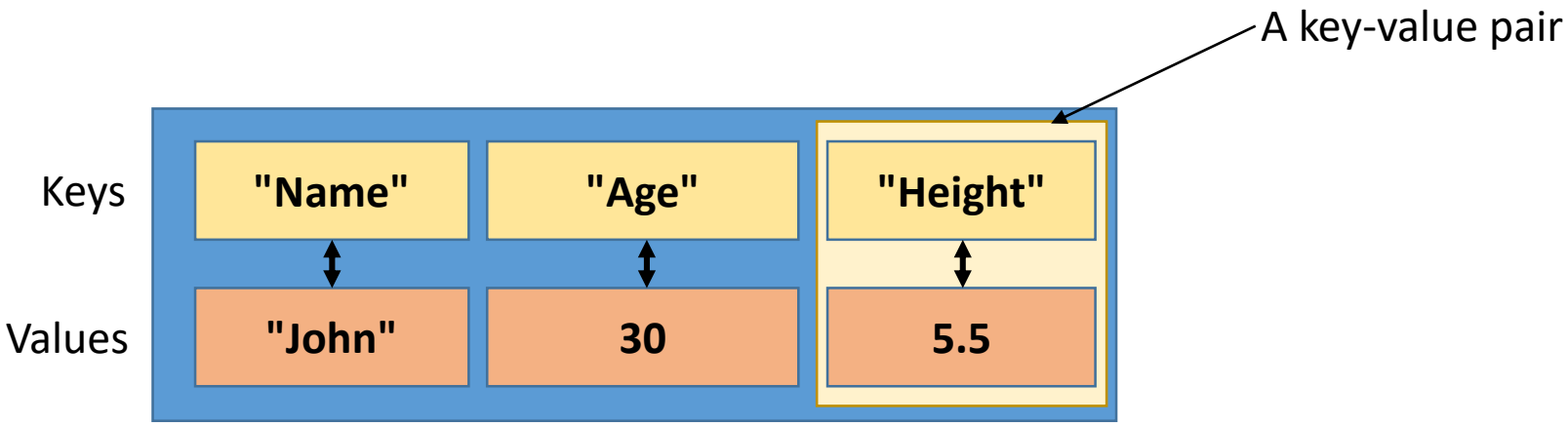
- A dictionary's values are accessed by a key (not an index).
  - Like in a real dictionary where its words each have a definition, in a Python dictionary its keys correspond to a value.



Dictionaries

- An element in a dictionary is called a ***key-value pair*** or ***KVP***

- Each key in a dictionary references a value.
  - A dictionary key can be any data type (ints, strings, etc)
  - A dictionary value can be any data type.
  - A key and its value do not have to be the same data type.



Dictionaries

- The key-value pairs are comma separated, enclosed in curly braces.
  - Key-value pair syntax is **key:value**

```
ages = {"Adam":41, "Bill":38.5, "Carol":45}

employees = {1001:"Joe", 1005:"Kathy", 1003:"Lou"}
```

- Dictionaries are always unordered.
  - It is not necessary to ever have to sort a dictionary since it is not indexed.

Dictionaries

- A dictionary's length is the total number of KVPs contained within it.
  - Python's built-in len function returns the length of a mapping data type.

```
employees = {1001:"Joe", 1005:"Kathy", 1003:"Lou"}
length = len(employees)
print(length)
```

```
3
```

Dictionaries

- Values in a dictionary can be referenced in a fashion similar to subscript notation.
  - Specify the key of the desired value.

```
employees = {1001:"Joe", 1005:"Kathy", 1003:"Lou"}
value = employees[1005]
print(value)

ages = {"Adam":41, "Bill":38.5, "Carol":45}
print(ages["Carol"])
```

```
Kathy
45
```

Dictionaries

- A KeyError exception will be raised if you try to access the value of a key that does not exist.

```
employees = {1001:"Joe", 1005:"Kathy", 1003:"Lou"}
value = employees[1010]
print(value)
```

```
Traceback (most recent call last):
  File "C:\testing\examples.py", line 8, in <module>
    value = employees[1010]
KeyError: 1010
>>>
```

Dictionaries

- When passed to the print function, the entire dictionary is printed.
  - Includes commas, colons and braces.
  - Useful for testing/debugging.

```
employees = {1001:"Joe", 1005:"Kathy", 1003:"Lou"}
print(employees)
```

```
{1001: 'Joe', 1005: 'Kathy', 1003: 'Lou'}
```

Adding a KVP

- Elements are added to a dictionary in a fashion similar to subscript notation.
  - Specify the key for the new value.

```
employees = {1001:"Joe", 1005:"Kathy", 1003:"Lou"}
employees[1002] = "Mary"
print(employees)
```

```
{1001: 'Joe', 1005: 'Kathy', 1003: 'Lou', 1002: 'Mary'}
```

Updating a KVP

- Values in a dictionary are mutable.
  - Specify the key and assign a new value to it.

```
employees = {1001:"Joe", 1005:"Kathy", 1003:"Lou"}
employees[1001] = "Joseph"
print(employees)
```

```
{1001: 'Joseph', 1005: 'Kathy', 1003: 'Lou'}
```

Deleting a KVP

- Use the **del** (delete) keyword to remove a key-value pair.
  - Reference the key to remove it and it's value.

```
ages = {"Adam":41, "Bill":38.5, "Carol":45}
del ages["Bill"]
print(ages)
```

```
{'Adam':41, 'Carol':45}
```

Iterating Over a Dictionary

- For loops iterate over the keys of a list.

```
employees = {1001:"Joe", 1005:"Kathy", 1003:"Lou"}
for id in employees :
  print(id)

print()

ages = {"Adam":41, "Bill":38.5, "Carol":45}
for name in ages :
  print(name)
```

```
1001
1005
1003

Adam
Bill
Carol
```

Iterating Over a Dictionary

- Use each key to iterate over the values.

```
employees = {1001:"Joe", 1005:"Kathy", 1003:"Lou"}
for id in employees :
  print(employees[id])

print()

ages = {"Adam":41, "Bill":38.5, "Carol":45}
for name in ages :
  print(ages[name])
```

```
Joe
Kathy
Lou

41
38.5
45
```

Turning Two Lists into a Dictionary

- Python's built-in dict and zip functions can be use to create a dictionary based on the contents of two lists.
  - The first argument/list will be used as the keys.
  - The second argument/list will be used as the values.

```
key_list = ["Up", "Down", "Left", "Right"]
value_list = ["North", "South", "West", "East"]
directions= dict(zip(key_list, value_list))
print(directions["Down"])
print(directions)
```

```
South
{'Up': 'North', 'Down': 'South', 'Left': 'West', 'Right': 'East'}
```

Sets

- A **set** is a collection of elements that:
  - Contains no duplicates
    - All elements in a set must be unique.
  - Has no ordering
    - Unlike lists or tuples, the elements in a set have no order

- Elements in a set can be different data types.

Sets

- Sets are declared as a series of comma-separated values in curly braces.

```
numbers = {4, 8, 15, 16, 23, 42}
values = {35.6, 32.76, 51.4}
pets = {"dog", "cat", "bird", "fish"}
```

- The data types of a series may vary.

```
mixedValues = {35.6, 15, "cat"}
```

Sets

- When passed to the print function, the entire set is printed.
  - Includes commas and brackets.
  - *Will probably be shown in a different order, as order does not matter to a set*.
  - Useful for testing/debugging.

```
numbers = {4, 8, 15, 16, 23, 42}
print(numbers)
```

```
{4, 8, 42, 15, 16, 23}
```

Sets

- The elements in a set are unique.
  - There will be no duplicates.

```
numbers = {4, 8, 8, 15, 16, 23, 42}
print(numbers)
```

```
{4, 8, 42, 15, 16, 23}
```

Sets

- Python's built-in set function(with no arguments) returns an empty set.

```
numbers = set()
print(numbers)
```

```
set()
```

- Python treats {} as an **empty dictionary**, not an empty set.

```
numbers = {}
print(numbers)
```

```
{}
```

Sets

- Unlike the sequence types (lists, tuples, strings), sets do not support indexing.
  - This is because sets are unordered.
  - This makes it impossible to reference a single value from the list.

- While it is not possible to retrieve or change a value (sets are immutable), we can add values to a set and merge sets together.

Sets

- A set's *length* is the total number of elements contained within it.
  - Python's built-in len function returns the length of a sequence data type.

```
pets = {"dog", "cat", "bird", "fish"}
length = len(pets)
print(length)
```

```
4
```

Iterating Over a Set

- For loops can iterate over the values of a set.

```
numbers = {4, 8, 15, 16, 23, 42}
for number in numbers :
  print(number)

print()

pets = {"dog", "cat", "bird", "fish"}
for animal in pets :
  print(animal)
```

```
4
8
42
15
16
23

dog
cat
bird
fish
```

Adding to Sets

- Values can be added to a set using the set's add function.
  - The value passed as the argument will be added to the set.

```
numbers = {4, 8, 15, 16, 23, 42}
numbers = numbers + 100
print(numbers)
```


```
numbers = {4, 8, 15, 16, 23, 42}
numbers.add(100)
print(numbers)
```

```
{4, 100, 8, 42, 15, 16, 23}
```

- Unlike lists, sets cannot be concatenated together.

Deleting from Sets

- To remove an element from a set, call the set's discard function.
  - The value passed as an argument is the value that will be removed from the set.

```
numbers = {4, 8, 15, 16, 23, 42}
numbers.discard(16)
print(numbers)
```

```
{4, 8, 42, 15, 23}
```

  - If the value did not exist, nothing will change in the set.

```
numbers = {4, 8, 15, 16, 23, 42}
numbers.discard(100)
print(numbers)
```

```
{4, 8, 42, 15, 16, 23}
```

Converting a List to a Set

- An argument (like a list or tuple) passed to Python's built-in set function will return that object as a set.
  - Any duplicates will be removed.

```
numbers = [4, 8, 8, 15, 16, 23, 42]
number_set = set(numbers)
print(number_set)
```

```
{4, 8, 42, 15, 16, 23}
```

Converting a Set to a List

- A set passed as an argument to Python's built-in list function will return that set as a list.

```
numbers = {4, 8, 15, 16, 23, 42}
number_list = list(numbers)
print(number_list)
```

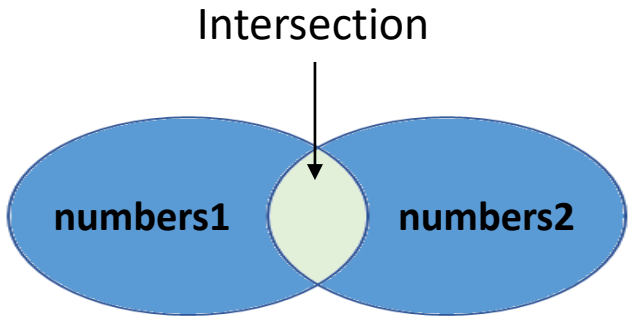
```
[4, 8, 42, 15, 16, 23]
```

Intersection

- An **intersection** of two sets is the set of elements that exist in both sets.

```
numbers1 = {20, 40, 10, 30}
numbers2 = {10, 50, 30, 60}
intersect = numbers1.intersection(numbers2)
print(intersect)
```

```
{10, 30}
```

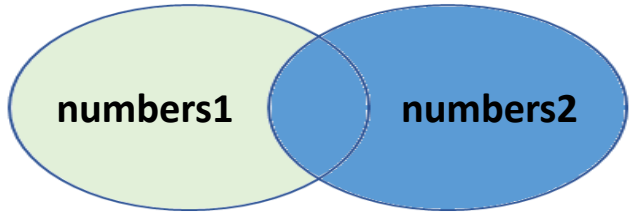


Difference

- A **difference** of two sets is the set of elements that exist only in the first set, but not in the second.

```
numbers1 = {20, 40, 10, 30}
numbers2 = {10, 50, 30, 60}
diff = numbers1.difference(numbers2)
print(diff)
```

```
{20, 40}
```

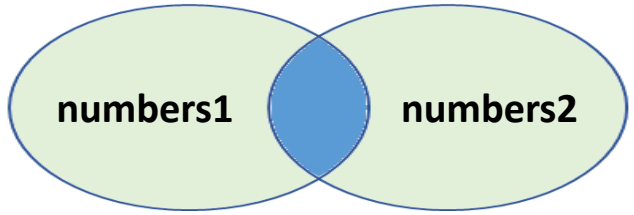


Symmetric Difference

- A **symmetric difference** of two sets is the set of elements not shared between the two sets.
  - It is the opposite of an intersection.

```
numbers1 = {20, 40, 10, 30}
numbers2 = {10, 50, 30, 60}
diff = numbers1.symmetric_difference(numbers2)
print(diff)
```

```
{20, 40, 50, 60}
```

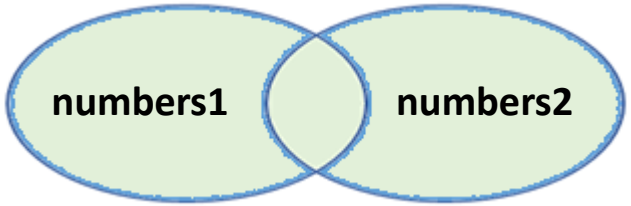


Union

- A **union** of two sets is a set that contains all elements from both sets.

```
numbers1 = {20, 40, 10, 30}
numbers2 = {60, 50, 80, 70}
union = numbers1.union(numbers2)
print(union)
```

```
{70, 10, 80, 20, 30, 40, 50, 60}
```



Subsets

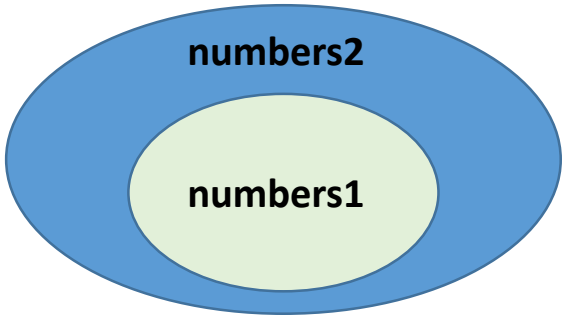
- A **subset** is a set that contains elements also found in another, usually larger set.
  - The set's issubset function returns true if all of its elements are present in the argument set. Otherwise, it returns false.

```
numbers1 = {10, 30}
numbers2 = {10, 50, 30, 60}
is_subset = numbers1.issubset(numbers2)
print(is_subset)
```

```
True
```

```
numbers1 = {10, 20}
numbers2 = {10, 50, 30, 60}
is_subset = numbers1.issubset(numbers2)
print(is_subset)
```

```
False
```



Supersets

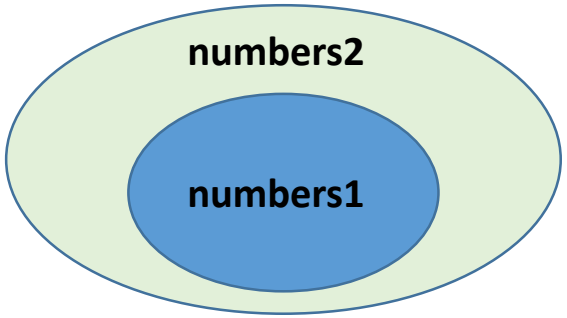
- A **superset** is a set that contains the elements found in another, usually smaller set.
  - The set's issuperset function returns true if all elements are present in the argument set are present. Otherwise, it returns false.

```
numbers1 = {10, 50, 30, 60}
numbers2 = {10, 30}
is_superset = numbers1.issuperset(numbers2)
print(is_superset)
```

```
True
```

```
numbers1 = {10, 50, 30, 60}
numbers2 = {10, 20}
is_superset = numbers1.issuperset(numbers2)
print(is_superset)
```

```
False
```



Sequence Type Comparisons

Structure	Mutable	Ordered	Indexed
String	✗	✓	✓
List	✓	✓	✓
Tuple	✗	✓	✓
Dictionary	✓	✓	✗
Set	✓	✗	✗