Revised: 7/27/2020

Dictionaries and Sets

Michael C. Hackett

Assistant Professor, Computer Science

Community College of Philadelphia

Lecture Topics

- Dictionary Basics
  - Key-Value Pairs
  - Adding, Changing and Deleting KVPs
  - Iterating Over a Dictionary's Keys and Values
- Dictionary Functions
  - Clear Function
  - Get Function
  - Pop Function
- Copying Dictionaries
- Merging Dictionaries

- Set Basics
  - Retrieving Elements
  - Iterating Over a Set
  - Adding Elements to Sets
  - Deleting Elements from Sets
- Set Functions
  - Unions
  - Intersections
  - Differences
  - Symmetric Differences
  - Subsets
  - Supersets

Colors/Fonts

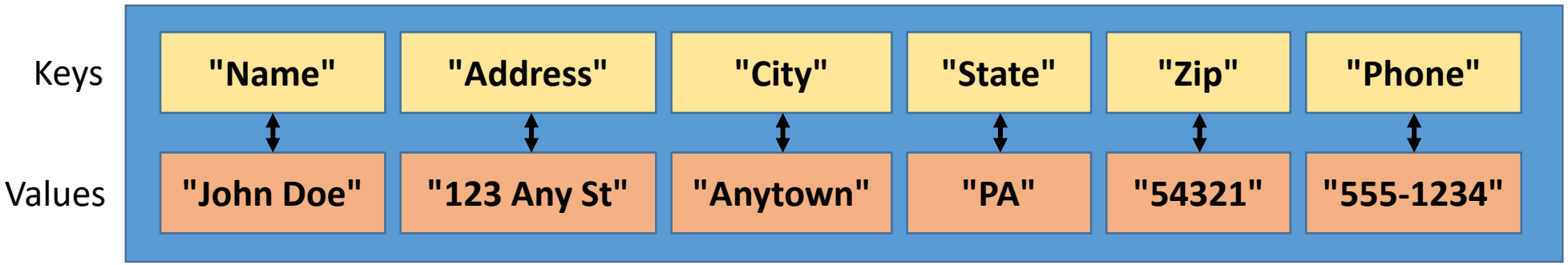
- Global Variable Names – Brown
- Local Variable Names – Lt Blue
- Literals – Blue
- Keywords – Orange
- Operators/Punctuation – Black
- Functions – Purple
- Parameters – Gold
- Comments – Gray
- Modules – Pink
- Object/Class Names – Green

Source Code – **Consolas**
Output – Courier New

What is a Dictionary?

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

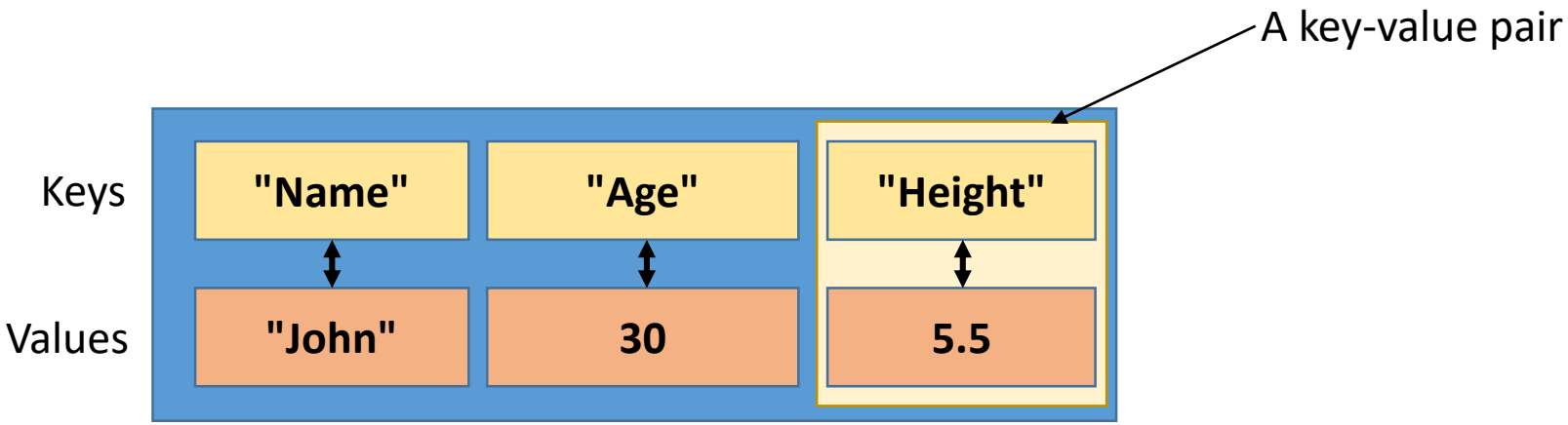
- A dictionary's values are accessed by a key (not an index).
  - Like in a real dictionary where its words each have a definition, in a Python dictionary its keys correspond to a value.



Key-Value Pairs

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

- 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.



Creating a Dictionary

- 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.

Getting the Length of a Dictionary

- 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
```

Retrieving a Value from a Dictionary

- 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
```

KeyError Exception

- 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
>>>
```

KeyError Exception

- KeyError exceptions can be handled using a try…except statement.

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

```
Invalid key
```

Printing a Dictionary

- 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 Key-Value Pair to a Dictionary

- 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'}
```

Changing a Value in a Dictionary

- 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 Key-Value Pairs from Lists

- 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
```

Determining if an Key Exists in a Dictionary

- An if statement can be used to find if an key is present in a dictionary.
  - Utilizes the **in** keyword.
  - Can be useful to avoid KeyError exceptions.

```
employees = {1001:"Joe", 1005:"Kathy", 1003:"Lou"}
key = 7001
if key in employees :
  print("Key exists in the dictionary")
else :
  print("Key does not exist in the dictionary")
```

```
Key does not exist in the dictionary
```

Clear Function

- A dictionary's clear function removes all key-value pairs from the dictionary.

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

```
{ }
```

Get Function

- A dictionary's get function returns a value based on the key passed to it as an argument.
  - Will not raise a KeyError exception. Returns None if the key does not exist.

```
ages = {"Adam":41, "Bill":38.5, "Carol":45}
age = ages.get("Bill")
print(age)
age = ages.get("Dave")
print(age)
```

```
38.4
None
```

Pop Function

- A dictionary's pop function returns a value based on the key passed to it as an argument.
  - The key-value pair is then removed.
  - Will not raise a KeyError exception. Returns None if the key does not exist.

```
ages = {"Adam":41, "Bill":38.5, "Carol":45}
age = ages.pop("Bill")
print(age)
print(ages)
```

```
38.5
{'Adam': 41, 'Carol': 45}
```

Copying Dictionaries

- Assigning a dictionary's variable to another variable creates a shallow copy.

```
ages = {"Adam":41, "Bill":38.5, "Carol":45}
ages2 = ages
ages2["Adam"] = 100
print(ages["Adam"])
```

```
100
```

Copying Dictionaries

- A dictionary's copy function returns a deep copy.

```
ages = {"Adam":41, "Bill":38.5, "Carol":45}
ages2 = ages.copy()
ages2["Adam"] = 100
print(ages["Adam"])
```

```
41
```

Merging Dictionaries

- A dictionary's update function appends another dictionary's key-value pairs to itself.

```
employees = {1001:"Joe", 1005:"Kathy", 1003:"Lou"}
new_employees = {1006:"Mary", 1008:"Nick"}
employees.update(new_employees)
print(employees)
print(new_employees)
```

```
{1001: 'Joe', 1005: 'Kathy', 1003: 'Lou', 1006: 'Mary', 1008: 'Nick'}
{1006: 'Mary', 1008: 'Nick'}
```

Turning 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 values that works like a mathematical set.
- All elements in a set must be unique.
- Sets are unordered.
- Elements in a set can be different data types.

Creating a Set

- 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"}
```

Printing a Set

- 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}
```

Creating a Set

- 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}
```

Creating an Empty Set

- 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)
```

```
{}
```

Getting the Length of a Set

- 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
```

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]
```

Retrieving or Changing an Element in a Set

- 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 also merge sets together.

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)
```

Error

```
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 Elements from Lists

- 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}
```

Determining if an Element Exists in a Set

- An if statement can be used to find if an element/value is present in a set.
  - Utilizes the **in** keyword.

```
numbers = {4, 8, 15, 16, 23, 42}
value = 7
if value in numbers :
  print("Value exists in list")
else :
  print("Value does not exist in list")
```

```
Value does not exist in list
```

Merging Sets Together

- A set's update function will add values from another set to itself.

```
numbers = {4, 8, 15, 16, 23, 42}
numbers2 = {200, 201}
numbers.update(numbers2)
print(numbers)
print(numbers2)
```

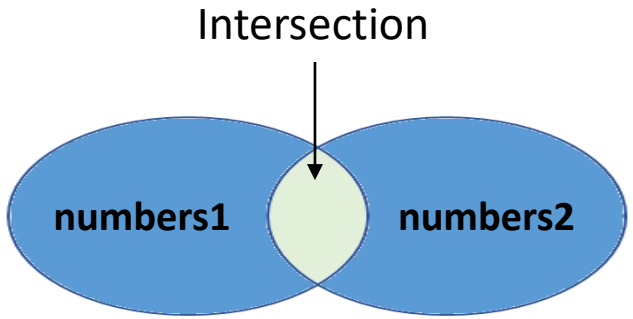
```
{4, 8, 200, 42, 201, 15, 16, 23, 42}
{200, 201}
```

Intersections of Sets

- 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}
```

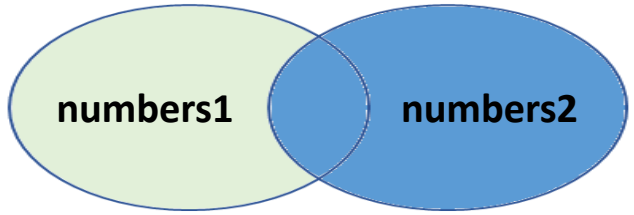


Difference of Sets

- 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}
```

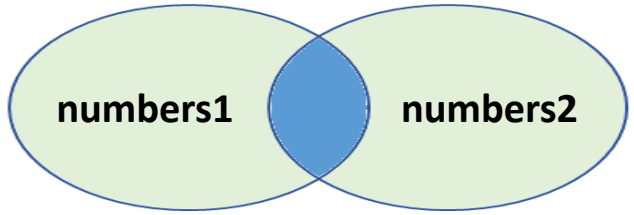


Symmetric Difference of Sets

- 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}
```

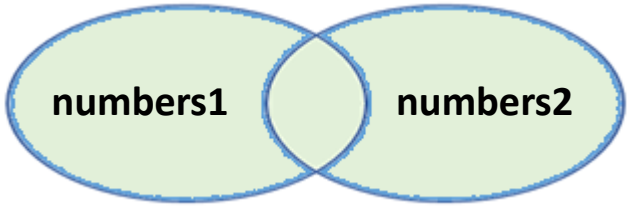


Unions of Sets

- 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}
```



Finding Subsets

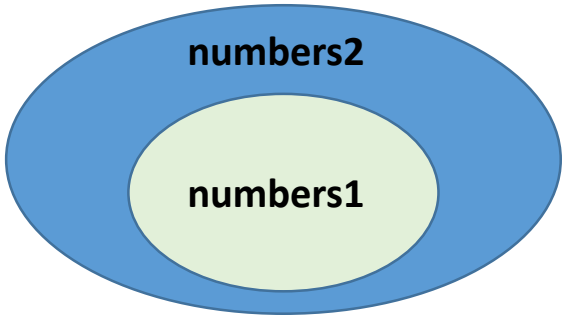
- A **subset** is a set that contains elements also found in another, usually larger set.
  - The set's issubset function returns true if all of its elements are present in the argument set. Otherwise, it returns false.

```
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
```



Finding 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
```