كلية
الحاسبات والمعلومات
كلية معتمدة من الهيئة القومية
لضمان جودة التعليم والاعتماد

Assiut University

Course Title: Data Structures and Algorithms
Course Code: CS211

Prof. Dr. Khaled F. Hussain

Reference

- Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser, “Data Structures and Algorithms in Python”

Objects in Python

- Python is an object-oriented language, and ***classes*** form the basis for all data types.
- Identifiers in Python are ***case-sensitive.***
- Unlike Java and C++, Python is a ***dynamically typed*** language, as there is no advance declaration associating an identifier with a particular data type. An identifier can be associated with any type of object, and it can later be reassigned to another object of the same (or different) type.
- A programmer can establish an ***alias*** by assigning a second identifier to an existing object.
  - Once an alias has been established, either name can be used to access the underlying object.
  - However, if one of the *names* is reassigned to a new value using a subsequent assignment statement, that does not affect the aliased object, rather it breaks the alias.
  - Example:
    - t1 = 50.5
    - t2=t1 # t1 and t2 points to the float object contains 50.5
    - t1=t1+10 # The result is stored as a new floating-point instance.
    - # t1 points to the float object contains 60.5 and t2 points to the float object contains 50.5

Creating and Using Objects

- The process of creating a new instance of a class is known as ***instantiation***.
  - In general, the syntax for instantiating an object is to invoke the ***constructor*** of a class.
- A class is ***immutable*** if each object of that class has a fixed value upon instantiation that cannot subsequently be changed. For example, the float class is immutable.
  - bool, int, float, tuple, str, frozenset classes are ***immutable***
  - list***,*** set, dict classes are **not** ***immutable***

Creating and Using Objects (Cont.)

- There are four collection data types in the Python programming language:
  - **List** is a collection which is ordered and changeable (**not *immutable*)**. Allows duplicate members.
  - **Tuple** is a collection which is ordered and unchangeable **(*immutable*).** Allows duplicate members.
  - **Set** is a collection which is unordered, changeable, and unindexed. No duplicate members.
  - **Dictionary** is a collection which is ordered and changeable. No duplicate members.

Creating and Using Objects (Cont.)

- The bool Class*:*
    - The default constructor, bool( ), returns False
    - Python allows the creation of a Boolean value from a nonboolean type using the syntax bool(foo) for value foo.
    - Numbers evaluate to False if zero, and True if nonzero.
    - strings and lists, evaluate to False if empty and True if nonempty.

Creating and Using Objects (Cont.)

- The int Class
  - Unlike Java and C++, which support different integral types with different precisions (e.g., int, short, long), Python automatically chooses the internal representation for an integer based upon the magnitude of its value.
  - Example of such literals are respectively 0b1011, 0o52, and 0x7f (binary, octal, hexadecimal).
  - The integer constructor, int( ), returns value 0 by default.
  - For example, if f represents a floating-point value, the syntax int(f) produces the *truncated* value of f. For example, both int(3.14) and int(3.99) produce the value 3

Creating and Using Objects (Cont.)

- The str Class
  - str class is specifically designed to efficiently represent an immutable sequence of characters, based upon the Unicode international character set.
  - String literals can be enclosed in single quotes, as in ‘hello’ , or double quotes, as in "hello".
  - Python also supports using the delimiter or """ to begin and end a string literal. The advantage of such triple-quoted strings is that newline characters can be embedded naturally (rather than escaped as \n).
  - Unicode characters can be included, such as ‘20\u20AC’ for the string 20€.

Creating and Using Objects (Cont.)

- The tuple Class
  - The `tuple` class provides an immutable version of a sequence
  - `()` being an empty tuple.
  - To express a tuple of length one as a literal, a comma must be placed after the element
  - For example, `(17,)` is a one-element tuple- the expression `(17)` is viewed as a simple parenthesized numeric expression.

The set and frozenset Classes

- The set and frozenset Classes
    - Python’s set class represents the mathematical notion of a set, namely a collection of elements, without duplicates, and without an inherent order to those elements.
    - This is based on a data structure known as a ***hash table***.
    - The set does not maintain the elements in any particular order.
    - Only instances of ***immutable*** types can be added to a Python set such as integers, floating-point numbers, and character strings
    - The frozenset class is an immutable form of the set type, so it is legal to have a set of frozensets.
    - Python uses curly braces { and } as delimiters for a set.
    - {17} or { red , green , blue }.
    - The exception to this rule is that { } does not represent an empty set; for historical reasons, it represents an empty dictionary
    - Instead, the constructor syntax set( ) produces an empty set.
    - For example, set(‘hello’) produces { ‘h’ , ‘e’ , ‘l’ , ‘o’ }.

The set and frozenset Classes (Cont.)

- Sets and frozensets support the following operators:

key in s	containment check
key not in s	non-containment check
s1 == s2	s1 is equivalent to s2
s1 != s2	s1 is not equivalent to s2
s1 <= s2	s1 is subset of s2
s1 < s2	s1 is proper subset of s2
s1 >= s2	s1 is superset of s2
s1 > s2	s1 is proper superset of s2
s1 | s2	the union of s1 and s2
s1 & s2	the intersection of s1 and s2
s1 – s2	the set of elements in s1 but not s2
s1 ˆ s2	the set of elements in precisely one of s1 or s2

The set and frozenset Classes (Cont.)

```
myset = {"apple", "banana", "cherry"}
print(myset)
```

- Note: The values True and 1 are considered the same value in sets, and are treated as duplicates:
  - False and 0 is considered the same value:

```
thisset = {"apple", "banana", "cherry", True, 1, 2}
print(thisset)
```

The set and frozenset Classes (Cont.)

```
print(len(thisset))
```

- Using the set() constructor to make a set:

```
thisset = set(("apple", "banana", "cherry")) # note the double round-brackets
print(thisset)

thisset = {"apple", "banana", "cherry"}
for x in thisset:
  print(x)
```

The set and frozenset Classes (Cont.)

```
thisset = {"apple", "banana", "cherry"}
print("banana" in thisset)
```

```
thisset = {"apple", "banana", "cherry"}
thisset.add("orange")
print(thisset)
```

- To add items from another set into the current set, use the update() method.

```
thisset = {"apple", "banana", "cherry"}
tropical = {"pineapple", "mango", "papaya"}
thisset.update(tropical)
print(thisset)
```

The set and frozenset Classes (Cont.)

- The object in the update() method does not have to be a set, it can be any iterable object (tuples, lists, dictionaries etc.).

```
thisset = {"apple", "banana", "cherry"}
mylist = ["kiwi", "orange"]
thisset.update(mylist)
print(thisset)

thisset = {"apple", "banana", "cherry"}
thisset.remove("banana")
print(thisset)
```

The set and frozenset Classes (Cont.)

- Remove a random item by using the pop() method:

```
thisset = {"apple", "banana", "cherry"}
x = thisset.pop()
print(x)
print(thisset)

thisset = {"apple", "banana", "cherry"}
thisset.clear()
print(thisset)
```

The set and frozenset Classes (Cont.)

- The del keyword will delete the set completely:

```
thisset = {"apple", "banana", "cherry"}
del thisset
print(thisset) # Error
```

The set and frozenset Classes (Cont.)

```
set1 = {"a", "b", "c"}
set2 = {1, 2, 3}
set3 = set1.union(set2)
print(set3)
```

- The union() method allows you to join a set with other data types, like lists or tuples.
- Join a set with a tuple:

```
x = {"a", "b", "c"}
y = (1, 2, 3)
z = x.union(y)
print(z)
```

The set and frozenset Classes (Cont.)

```
set1 = {"apple", "banana", "cherry"}
set2 = {"google", "microsoft", "apple"}
set3 = set1.intersection(set2)
print(set3)
```

- Use & for intersection of sets:

```
set1 = {"apple", "banana", "cherry"}
set2 = {"google", "microsoft", "apple"}
set3 = set1 & set2
print(set3)
```

- The & operator only allows you to join sets with sets, and not with other data types like you can with the intersection() method.

The set and frozenset Classes (Cont.)

```
animals = frozenset(["cat", "dog", "lion"])
print("cat" in animals)
print("elephant" in animals)

animals = ["cat", "dog", "lion"]
 # converting list to frozenset
animals2 = frozenset(animals)
print("frozenset Object is : ", animals2)
```

The set and frozenset Classes (Cont.)

```
# initialize A and B
A = frozenset([1, 2, 3, 4])
B = frozenset([3, 4, 5, 6])

# copying a frozenset
C = A.copy()
print(C)

# union
union_set = A.union(B)
print(union_set)
```

The set and frozenset Classes (Cont.)

```
# intersection
intersection_set = A.intersection(B)
print(intersection_set)

difference_set = A.difference(B)
print(difference_set)

# symmetric_difference
symmetric_difference_set = A.symmetric_difference(B)
print(symmetric_difference_set)
```

- The ^ operator only allows you to join sets with sets, and not with other data types like you can with the symmetric_difference() method.

The set and frozenset Classes (Cont.)

```
Z_union=A | B
print(Z_union)

Z_intersection =A & B
print(Z_intersection)
```

The dict Class

- The dict Class
  - Python's dict class represents a ***dictionary***, or ***mapping***, from a set of distinct ***keys*** to associated ***values***.
  - For example, a dictionary might map from unique student ID numbers, to larger student records (such as the student's name, address, and course grades).
  - Python implements a dict using an almost identical approach to that of a set, but with storage of the associated values.
  - A dictionary literal also uses curly braces, and because dictionaries were introduced in Python prior to sets, the literal form { } produces an empty dictionary.
  - For example, the dictionary thisdict = {"brand": "Ford", "model": "Mustang", "year": 1964 }

The dict Class (Cont.)

• d[key]	value associated with given key
• d[key] = value	set (or reset) the value associated with given key
• del d[key]	remove key and its associated value from dictionary
• key in d	containment check
• key not in d	non-containment check
• d1 == d2	d1 is equivalent to d2
• d1 != d2	d1 is not equivalent to d2

The dict Class (Cont.)

```
thisdict={"brand":"Ford","model":"Mustang","year":19
64}
print(len(thisdict))
```

- Get the value of the "model" key:

```
print(thisdict["model"])
```

- or

```
print(thisdict.get("model"))
```

The dict Class (Cont.)

- Get Keys

```
print(thisdict.keys())
```

- Add a new item to the original dictionary

```
thisdict["color"]="white"
print(thisdict)
```

- Get a list of the values:
- print(thisdict.values())

The dict Class (Cont.)

- Make a change in the dictionary

```
thisdict["year"]=2020
```

- Or

```
thisdict.update({"year":2020})
```

- Get each item in a dictionary, as tuples in a list.

```
print(thisdict.items())
```

The dict Class (Cont.)

- Using the dict() method to make a dictionary:

```
thisdict=dict(name="John", age =
36,country="Norway")
print(thisdict)
```

The dict Class (Cont.)

- Dictionaries do not save two items with the same key:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964,
  "year": 2020
}
print(thisdict)
```

- Result:
- {'brand': 'Ford', 'model': 'Mustang', 'year': 2020}

The dict Class (Cont.)

- Check if "model" is present in the dictionary:

```
if "model" in thisdict:
    print("Yes, 'model' is one of the keys in the thisdict dictionary")
```

- The pop() method removes the item with the specified key name:

```
thisdict.pop("model")
print(thisdict)
```

- The popitem() method removes the last inserted item:

```
thisdict.popitem()
print(thisdict)
```