Data Structures and Algorithms

Lesson 2. Python Data Types and Structures

Alexandra Ciobotaru

25 feb 2022

Syllabus

- **Lesson 1. Introduction: Algorithms, Data Structures and Cognitive Science.**
- **Lesson 2. Python Data Types and Structures.**
- Lesson 3. Principles of Algorithm Design.
- Lesson 4. Lists and Pointer Structures.
- Lesson 5. Stacks and Queues.
- Lesson 6. Trees.
- Lesson 7. Hashing and Symbol Tables.
- Lesson 8. Graphs and Other Algorithms.
- Lesson 9. Searching.
- Lesson 10. Sorting. Mid-term evaluation quiz (30%)
- Lesson 11. Selection Algorithms.
- Lesson 12. String Algorithms and Techniques.
- Lesson 13. Design Techniques and Strategies.
- Lesson 14. Implementations, Applications and Tools.
- Lab (20%) + Project (50%).

Overview of data types and objects

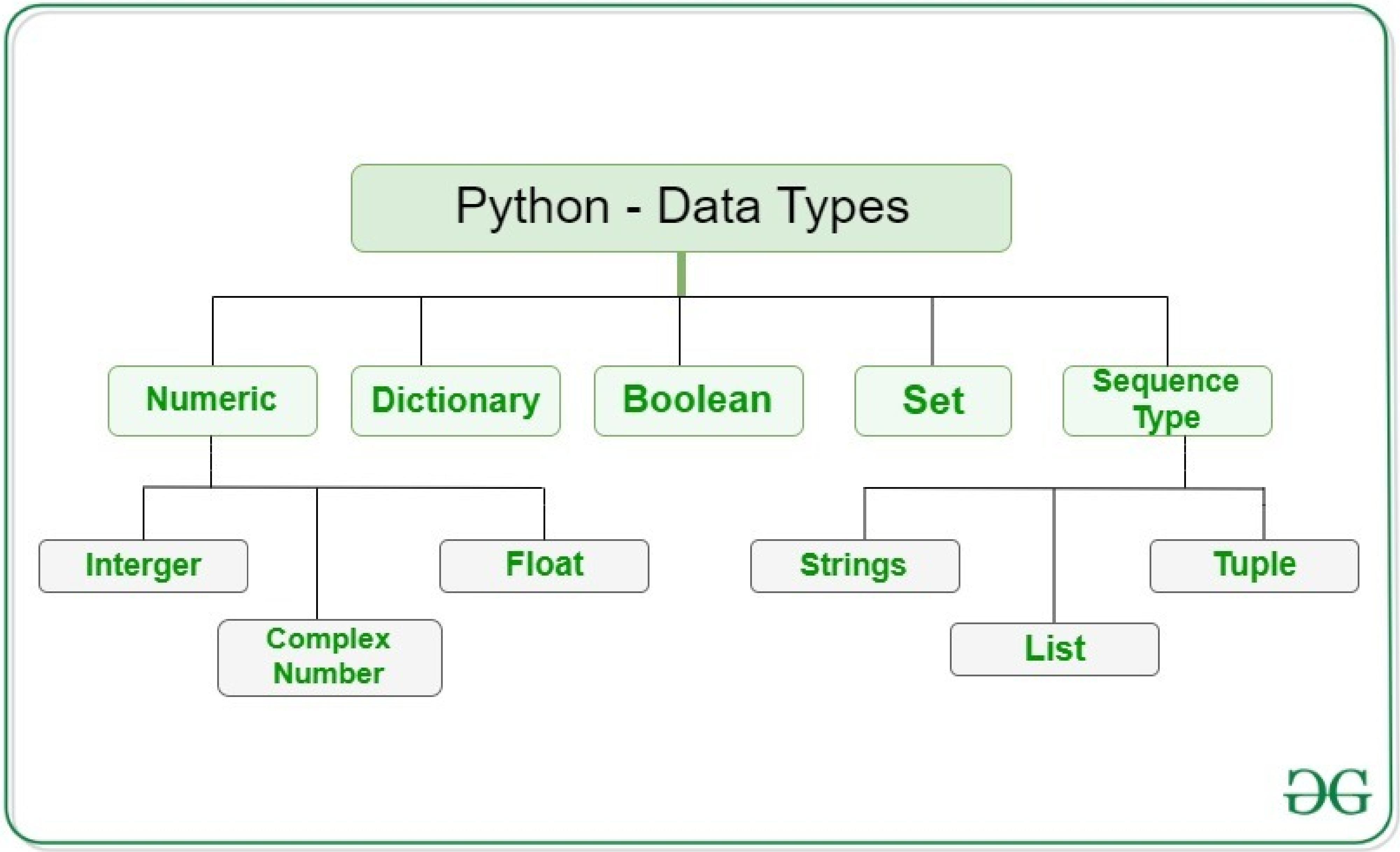
- Python contains various built-in data types (are also called Abstract Data Types).
- These include:
  - None type (`None`),
  - ***numeric*** types (`int`, `float`, `complex`, `bool`),
  - ***sequence*** types (`str`, `list`, `tuple`, `range`),
  - ***mapping*** types (`dict`, `set, frozenset`).
- It is also possible to create user-defined objects, such as functions or classes.

All data types in Python are objects

In fact, pretty much everything is an object in Python, including modules, classes, and functions, as well as literals such as strings and integers.

- Each object in Python has a type, a value, and an identity.
- When we write greet="hello world", we are creating an *instance* of a string object with the value "hello world" and the identity of greet.
- The identity (`id`) of an object acts as a pointer to the object's location in memory. The type (`type`) of an object, also known as the object's class, describes the object's internal representation, as well as the methods and operations it supports.

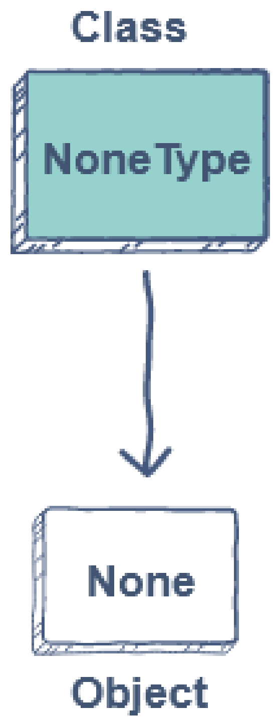
```
>>> x = 5
>>> y = 5
>>> id(x)
140722153297696
id(y)
140722153297696
x is y
True
x == y
True
type(x)
<class 'int'>
```


Python - Data Types
Numeric
Dictionary
Boolean
Set
Sequence Type
Interger
Complex Number
Float
Strings
List
Tuple

None type

- None is used to show the absence of a value;
- The None type is immutable;
- It is similar to null in many programming languages, such as C and C++;
- Objects return None when there is actually nothing to return;
- None is often used as a default value in function arguments to detect whether a function call has passed a value or not.

```
# Declaring a None variable
var = None
if var is None: # Checking if the variable is None
  print("None")
else:
  print("Not None")
```



Numeric types: int, float, complex

- Integer (int) data type only holds integer numbers, which could be positive or negative.

  ```
  x = int(20)
  ```

- Float data types are used to hold decimal numerical values i.e. 2.13, 3.14 etc.

  ```
  x = float(20.5)
  ```

  ```
  >>> a=4
  >>> b=5
  >>> d= b/a
  >>> print(d, "is of type", type(d))
  ```

  Let’s remember // and %

- Complex Number data types is used to keep complex numbers in it. Complex numbers are those numerical values which have real & imaginary part.

  ```
  x = complex(1+3j)
  ```

Complex Numbers

A Complex Number consist of a Real Part and an Imaginary Part

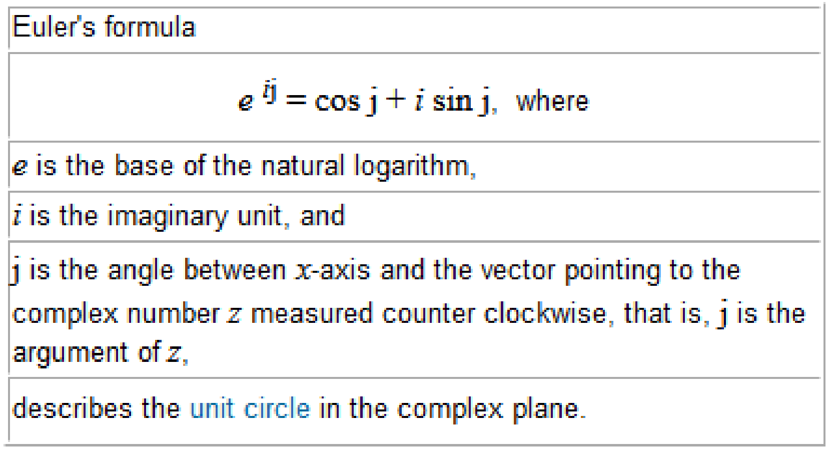
A nice story about how complex numbers were invented: https://youtu.be/cUzklzVXJwo

$$a + bi$$

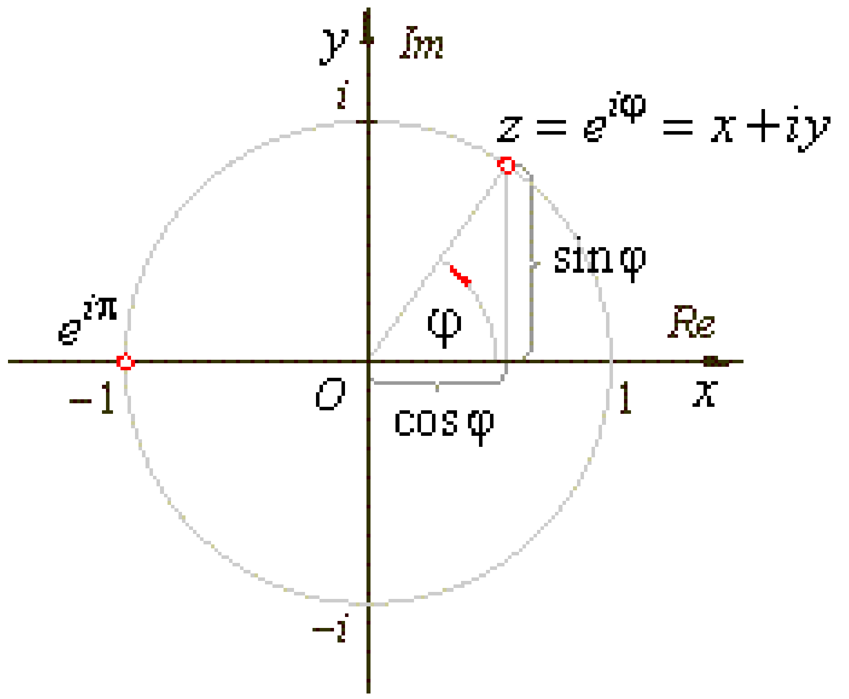
Real Part Imaginary Part

$$i^2 = -1$$

$$i = \sqrt{-1}$$



Euler's formula
$e^{ij} = \cos j + i \sin j$, where
e is the base of the natural logarithm,
i is the imaginary unit, and
j is the angle between x-axis and the vector pointing to the complex number z measured counter clockwise, that is, j is the argument of z,
describes the unit circle in the complex plane.



Polar or trigonometric notation of complex numbers
More on the topic: http://www.nabla.hr/FU-ImComplNumb4.htm

Booleans

- Can be either True of False

```
x = bool(1)
```

- Boolean Operations:

Operator	Example
not x	It returns False if x is True, and returns True if x is False
x and y	It returns True if x and y are both True; otherwise, it returns False
x or y	It returns True if either x or y is True; otherwise, it returns False

- The comparison operators (<, <=, >, >=, ==, and !=) work with numbers, lists, and other collection objects and return True if the condition holds.

Sequence types: string, list, tuple, range

- Sequences are ordered sets of objects indexed by non-negative integers.
- Strings = sequences of characters
- **x = str("Hello World")**
- Lists and tuples = sequences of arbitrary objects
    - Lists are mutable – **x = list([1,2,3])**
    - Tuples are immutable – **x = tuple((1,2,3))**
- Let's remember: ***casting*** = forcing a variable into another data type
- We can apply **len, min, max, sum, all, any** on all sequences

```
>>> x = [True, False]
>>> print(all(x))
False
>>> print(any(x))
True
```

Sequence types: string, list, tuple, range

- All sequences support the following operations:

Operation	Description
s+r	Concatenates two sequences of the same type
s*n	Makes n copies of s, where n is an integer.
s[i]	Indexing returns the i element of s
s[i:j:stride]	Slicing returns elements between i and j with optional stride
x in s	Returns True if the x element is in s.
s not in s	Returns True if the x element is not in s

- Let's remember: **append, extend, reverse, sort, remove** for list type

Tuples

- Tuples are **hashable**, which means we can sort lists of them and they can be used as keys to dictionaries.

```
>>> print(tuple('sequence'))
('s', 'e', 'q', 'u', 'e', 'n', 'c', 'e')
>>> tpl=('a','b','c')
>>> x,y,z = tpl
>>> x
'a'
>>> y
'b'
>>> z
'c`
>>> c = 1,2,3
>>> type(c)
<class 'tuple'>
```

Most operators, such as those for slicing and indexing, work as they do on lists. However, because tuples are immutable, trying to modify an element of a tuple will give you TypeError.

Dictionaries (As of Python version 3.7, dictionaries are *ordered*)

- A dictionary stores the data in a mapping of key and value pair {key:value}
- Dictionaries are mainly a collection of objects; they are indexed by numbers, strings, or any other immutable objects.
- Keys should be unique in the dictionaries; however, the values in the dictionary can be changed.

```
>>> c= dict(zip(['Monday','Tuesday','Wednesday'],
[1,2,3]))
>>> c
{'Monday': 1, 'Tuesday': 2, 'Wednesday': 3}
>>> c["Thursday"] = 4
>>> c
{'Monday': 1, 'Tuesday': 2, 'Wednesday': 3, 'Thursday': 4}
>>> dict(zip('packt', range(len('packt'))))
{'p': 0, 'a': 1, 'c': 2, 'k': 3, 't': 4}
```

Method	Description
len(d)	Returns total number of items in the dictionary, d.
d.Clear()	Removes all of the items from the dictionary, d.
d.fromkeys(s[,value])	Returns a new dictionary with keys from the s sequence and values set to value.
d.get(k[,v])	Returns d[k] if it is found; otherwise, it returns v (None if v is not given).
d.items()	Returns all of the key:value pairs of the dictionary, d.
d.keys()	Returns all of the keys defined in the dictionary, d.
d.pop(k[,default])	Returns d[k] and removes it from d
d.popitem()	Removes a random key:value pair from the dictionary, d, and returns it as a tuple.
d.update(b)	Adds all of the objects from the b dictionary to the d dictionary.
d.values()	Returns all of the values in the dictionary, d.

Dictionaries example

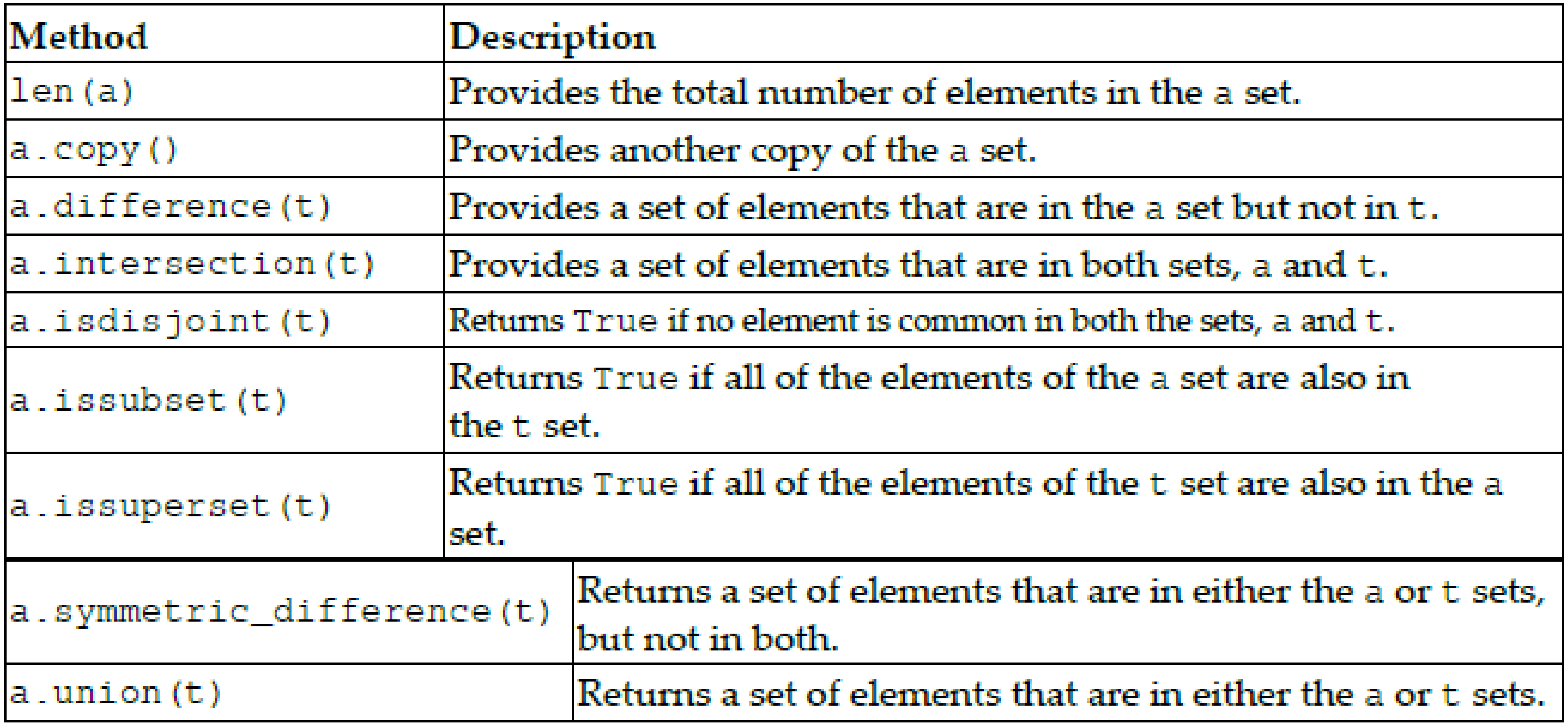
```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict)
print(len(thisdict))
print(thisdict["brand"])
thisdict['colors'] = ['blue','white','green']
thisdict
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'colors':
['blue', 'white', 'green']}
print(thisdict.keys())
dict_keys(['brand', 'model', 'year', 'colors'])
```

Sets

- Sets are typically used to perform mathematical operations such as intersection, union, difference, and complement.
- Sets are mutable – we can add and remove items from them
- Sets cannot contain duplicate items
- Sets are unordered, cannot be indexed and sliced
- Frozensets = immutable sets

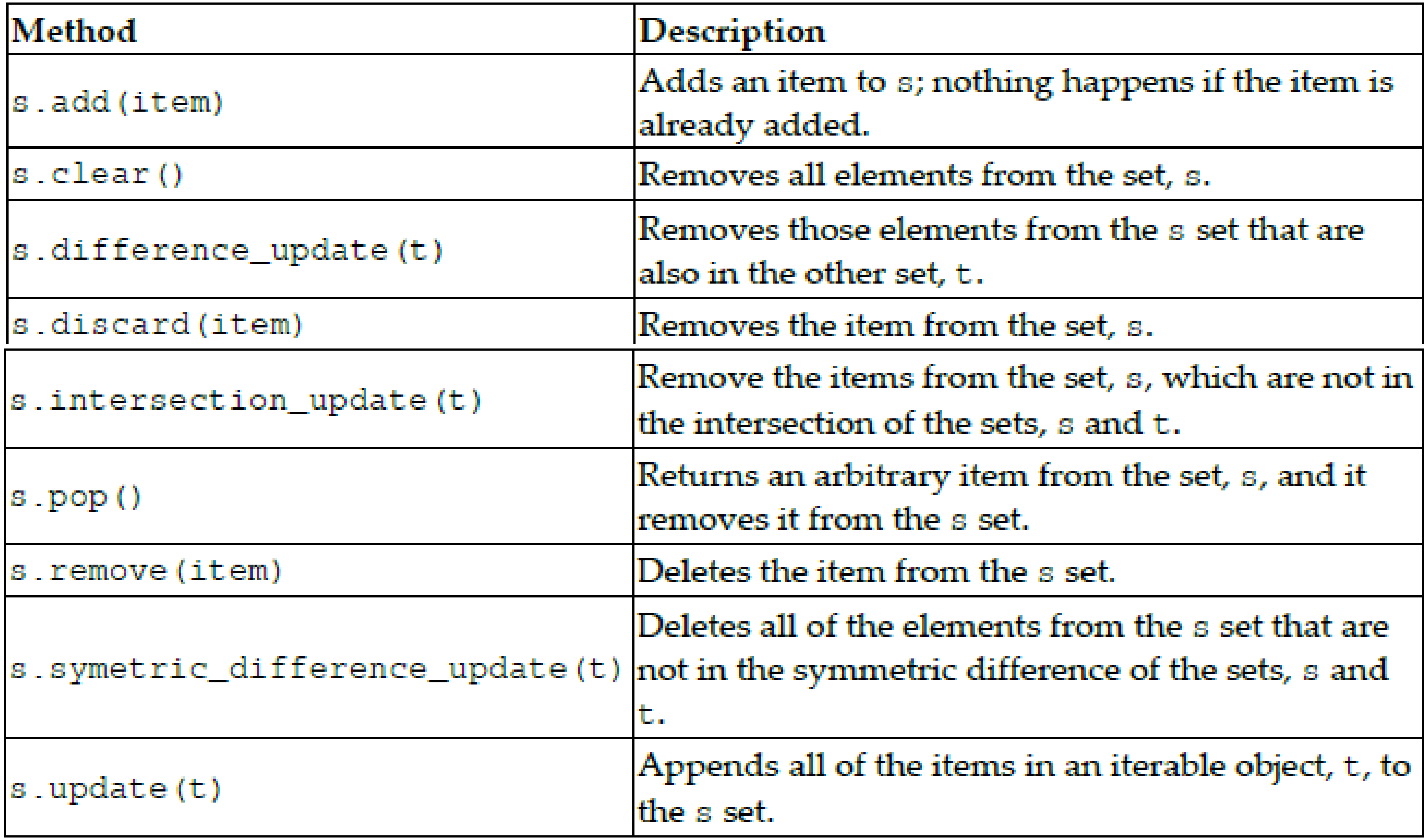
```
>>> c = {1,2,3,4}
>>> c.add(5)
>>> c
{1, 2, 3, 4, 5}
>>> d = frozenset(c)
>>> d.add(7)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'
```

Methods & Operations for Sets&Frozensets



Method	Description
`len(a)`	Provides the total number of elements in the a set.
`a.copy()`	Provides another copy of the a set.
`a.difference(t)`	Provides a set of elements that are in the a set but not in t.
`a.intersection(t)`	Provides a set of elements that are in both sets, a and t.
`a.isdisjoint(t)`	Returns `True` if no element is common in both the sets, a and t.
`a.issubset(t)`	Returns `True` if all of the elements of the a set are also in the t set.
`a.issuperset(t)`	Returns `True` if all of the elements of the t set are also in the a set.
`a.symmetric_difference(t)`	Returns a set of elements that are in either the a or t sets, but not in both.
`a.union(t)`	Returns a set of elements that are in either the a or t sets.

And only for Mutable Sets:



Method	Description
`s.add(item)`	Adds an item to `s`; nothing happens if the item is already added.
`s.clear()`	Removes all elements from the set, `s`.
`s.difference_update(t)`	Removes those elements from the `s` set that are also in the other set, `t`.
`s.discard(item)`	Removes the item from the set, `s`.
`s.intersection_update(t)`	Remove the items from the set, `s`, which are not in the intersection of the sets, `s` and `t`.
`s.pop()`	Returns an arbitrary item from the set, `s`, and it removes it from the `s` set.
`s.remove(item)`	Deletes the item from the `s` set.
`s.symetric_difference_update(t)`	Deletes all of the elements from the `s` set that are not in the symmetric difference of the sets, `s` and `t`.
`s.update(t)`	Appends all of the items in an iterable object, `t`, to the `s` set.

Modules for data structures and algorithms

- **Collections** - this module implements specialized container datatypes providing alternatives to Python’s general purpose built-in containers, like dict, list, set, and tuple.

namedtuple()	**factory function for creating tuple subclasses with named fields**
deque	list-like container with fast appends and pops on either end
ChainMap	dict-like class for creating a single view of multiple mappings
Counter	dict subclass for counting hashable objects
defaultdict	dict subclass that calls a factory function to supply missing values
UserDict	wrapper around dictionary objects for easier dict subclassing
UserList	wrapper around list objects for easier list subclassing
UserString	wrapper around string objects for easier string subclassing

Arrays

- An array is like a list, but it contains values of the same type!
- A Python array is a **container** that holds multiple elements in a one-dimensional catalog. Each element in an array can be identified by its **respective position**.
- You can easily find any element according to its position in the array container, by indexing:

```
carsList = ["BMW", "AUDI", "TOYOTA", "NISSAN"]
print(carsList[2])
```

Thank you!