Python is a modern, easy-to-learn, object-oriented programming language. It has a powerful

set of built-in data types and easy-to-use control constructs. Since Python is an interpreted

language, it is most easily reviewed by simply looking at and describing interactive sessions.

indexing [ ] Access an element of a sequence

concatenation + Combine sequences together

repetition \* Concatenate a repeated number of times

membership **in** Ask whether an item is in a sequence

length **len** Ask the number of items in the sequence

slicing [ : ] Extract a part of a sequence

In addition to the numeric and boolean classes, Python has a number of very powerful builtin

collection classes. Lists, strings, and tuples are ordered collections that are very similar in

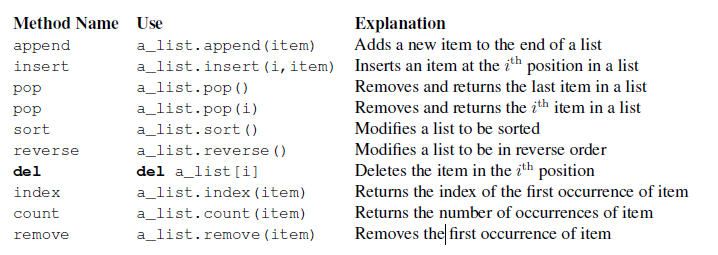
general structure but have specific differences that must be understood for them to be used

properly. Sets and dictionaries are unordered collections.

>>> my\_list = [0] \* 6

>>> my\_list

[0, 0, 0, 0, 0, 0]



>>> (54).\_\_add\_\_(21)

75

>>> list(range(10))

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> range(5,10)

range(5, 10)

>>> list(range(5,10))

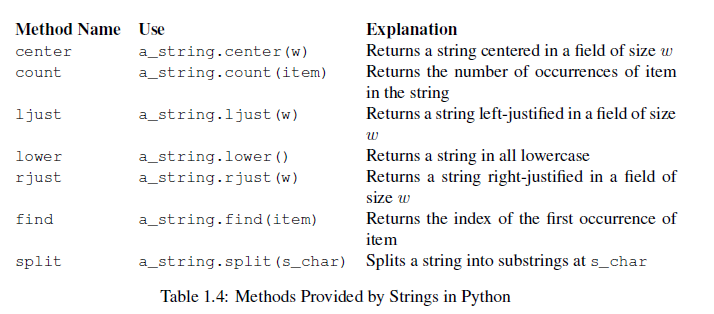
[5, 6, 7, 8, 9]

>>> list(range(5,10,2))

[5, 7, 9]

>>> list(range(10,1,-1))

[10, 9, 8, 7, 6, 5, 4, 3, 2]



A major difference between lists and strings is that lists can be modified while strings cannot.

This is referred to as mutability. Lists are mutable; strings are immutable. For example, you

can change an item in a list by using indexing and assignment. With a string that change is not

allowed.

Tuples are very similar to lists in that they are heterogeneous sequences of data. The difference

is that a tuple is immutable, like a string. A tuple cannot be changed. Tuples are written as

comma-delimited values enclosed in parentheses. As sequences, they can use any operation

described above. For example,

>>> my\_tuple = (2,True,4.96)

>>> my\_tuple

(2, True, 4.96)

>>> len(my\_tuple)

3

>>> my\_tuple[0]

2

>>> my\_tuple \* 3

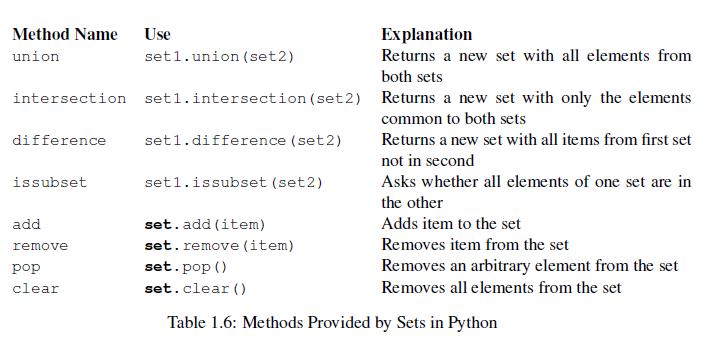
(2, True, 4.96, 2, True, 4.96, 2, True, 4.96)

A set is an unordered collection of zero or more immutable Python data objects. Sets do not

allow duplicates and are written as comma-delimited values enclosed in curly braces. The

empty set is represented by **set()**. Sets are heterogeneous, and the collection can be assigned

to a variable as below.

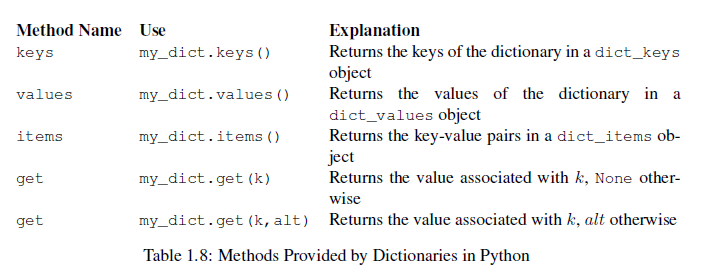


Our final Python collection is an unordered structure called a dictionary. Dictionaries are

collections of associated pairs of items where each pair consists of a key and a value. This

key-value pair is typically written as **key:value**. Dictionaries are written as comma-delimited

key:value pairs enclosed in curly braces



user\_name = input("Please enter your name ")

print("Your name in all capitals is",user\_name.upper(),

"and has length", len(user\_name))

print("%s is %d years old." % (name, age))

This simple example illustrates a new string expression. The **%** operator is a string operator

called the format operator. The left side of the expression holds the template or format string,

and the right side holds a collection of values that will be substituted into the format string.

Note that the number of values in the collection on the right side corresponds with the number

of **%** characters in the format string. Values are taken in order, left to right from the collection

and inserted into the format string.

word\_list = ['cat','dog','rabbit']

letter\_list = [ ]

for a\_word in word\_list:

for a\_letter in a\_word:

letter\_list.append(a\_letter)

print(letter\_list)

if n < 0:

print("Sorry, value is negative")

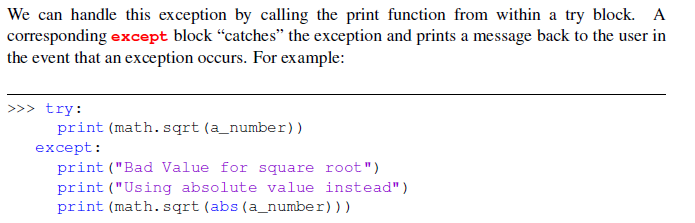
else:

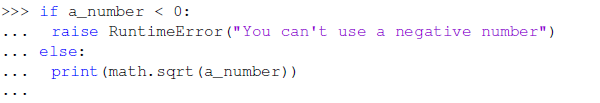
print(math.sqrt(n))

>>> sq\_list = [x \* x for x in range(1, 11) if x % 2 != 0]

>>> sq\_list

[1, 9, 25, 49, 81]





The first method that all classes should

provide is the constructor. The constructor defines the way in which data objects are created.

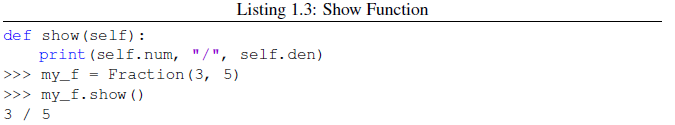
To create a **Fraction** object, we will need to provide two pieces of data, the numerator

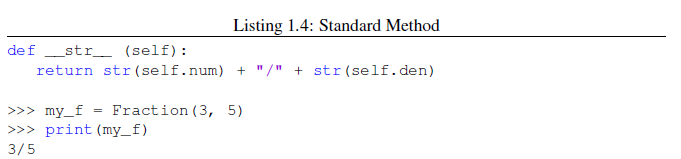
and the denominator. In Python, the constructor method is always called \_\_init\_\_(

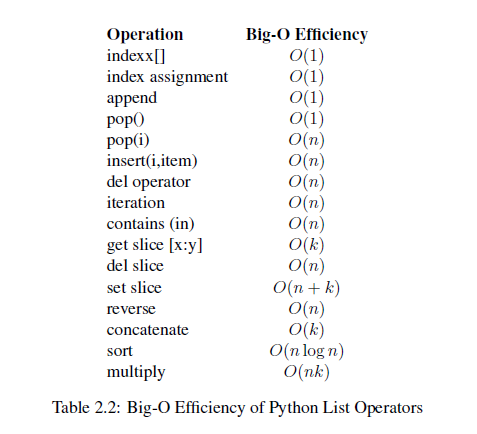
To create an instance of the **Fraction** class, we must invoke the constructor. This happens

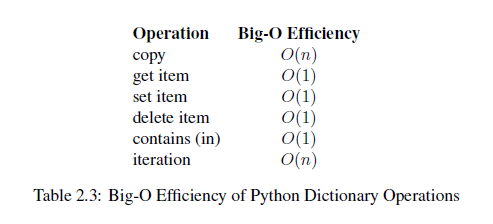
by using the name of the class and passing actual values for the necessary state (note that we

never directly **invoke \_\_init\_\_**).









In the process we will confirm that the contains operator for lists is 𝑂(𝑛) and the contains operator for dictionaries is 𝑂(1)

We will begin our study of data structures by considering four simple but very powerful concepts.

Stacks, queues, deques, and lists are examples of data collections whose items are ordered

depending on how they are added or removed. Once an item is added, it stays in that

position relative to the other elements that came before and came after it. Collections such as

these are often referred to as linear data structures.

Linear structures can be thought of as having two ends. Sometimes these ends are referred to as

the “left” and the “right” or in some cases the “front” and the “rear.” You could also call them

the “top” and the “bottom.”

STACK is LIFO

The simple parentheses checker from the previous section can easily be extended to handle

these new types of symbols. Recall that each opening symbol is simply pushed on the stack to

wait for the matching closing symbol to appear later in the sequence. When a closing symbol

does appear, the only difference is that we must check to be sure that it correctly matches the

type of the opening symbol on top of the stack. If the two symbols do not match, the string is

not balanced. Once again, if the entire string is processed and nothing is left on the stack, the

string is correctly balanced.

The queue abstract data type is defined by the following structure and operations. A queue is

structured, as described above, as an ordered collection of items which are added at one end,

called the “rear,” and removed from the other end, called the “front.” Queues maintain a FIFO

ordering property. The queue operations are given below.

• Queue() creates a new queue that is empty. It needs no parameters and returns an empty

queue.

• enqueue(item) adds a new item to the rear of the queue. It needs the item and returns

nothing.

• dequeue() removes the front item from the queue. It needs no parameters and returns the

item. The queue is modified.

• is\_empty() tests to see whether the queue is empty. It needs no parameters and returns a

boolean value.

• size() returns the number of items in the queue. It needs no parameters and returns an

integer.

In order to implement an unordered list, we will construct what is commonly known as a linked

list. Recall that we need to be sure that we can maintain the relative positioning of the items.

However, there is no requirement that we maintain that positioning in contiguous memory

Like the robots of Asimov, all recursive algorithms must obey three important laws:

1. A recursive algorithm must have a base case.

2. A recursive algorithm must change its state and move toward the base case.

3. A recursive algorithm must call itself, recursively.