AI 2024 Online Summer Internship

Name: Rasikh Ali

Email: rasikhali1234@gmail.com

Table of Contents

1. Assignment 1

[go back to the top]

For each Concept in the Initial Tutorial(for e.g., Variables, String, List, Tuple, Dictionary etc.) write down

- Definition
- Purpose
- Importance
- Applications
- Strengths
- Weaknesses
- Suitable to Use
- Given and Task
- Input and Output
- At least 3 – 5 Examples (Not Given in the Initial Tutorial)
- At least 1 Example to Solve Real-world Problems

Note: You can add Concepts in your Assignment Solution that are not mentioned in the Initial Tutorial

1.1 Data types and variables

[go back to the top]

When your program carries out computations, you will want to store values so that you can use them later. In a Python program, you use variables to store values. In this section, you will learn how to define and use variables.

> A **variable** is a storage location in a computer program. Each variable has a *name* and holds a *value*.

You use the **assignment statement** to place a value into a variable. Here is an example:

In [1]:
```
numberOfBooks = 10
```

The left-hand side of an assignment statement consists of a variable. The right-hand side is an expression that has a value. That value is stored in the variable. The first time a variable is assigned a value, the variable is created and initialized with that value. After a variable has been defined, it can be used in other statements. For example,

In [2]:
```
print(numberOfBooks)
```

```
10
```

will print the value stored in the variable numberOfBooks. If an existing variable is assigned a new value, that value replaces the previous contents of the

variable. For example,

In [3]:
```
numberOfBooks = 15
print(numberOfBooks)
```

```
15
```

changes the value contained in variable numberOfBooks from 5 to 10.

Note that in Python, it is not necessary to define what type of value they will store. The type of variable will be determined after the value is assigned to it. For example,

In [4]:
```
numberOfBooks = 15                  #integer type variable
counter = 120.00                    #float type variable
subject = "Artificial Intelligence"  #string type variable

print(numberOfBooks)
print(counter)
print(subject)
```

```
15
120.0
Artificial Intelligence
```

> **Note that a variable must be created and initialized before it can be used for the first time.** For example, if the variable **b** has not been created and the following code is run, we will get an error saying that *b has not yet been created*.
> `a = b + 100`

1.1.1 Number types

[go back to the top]

In Python, there are several different types of numbers. An **integer** value is a whole number without a fractional part. For example, there must be an integer number of books in any pack of books — you cannot have a fraction of a book. In Python, this type is called **int**.

When a fractional part is required (such as in the number 0.355), we use floating-point numbers, which are called **float** in Python. When a value such as 6 or 0.355 occurs in a Python program, it is called a **number literal**. If a number literal has a decimal point, it is a floating-point number; otherwise, it is an integer. For example,

In [5]:
```
5     #integer
```

Out[5]:
```
5
```

In [6]:
```
5 + 12  #integer
```

Out[6]:
```
17
```

In [7]:
```
5.9 + 5.12   #floating-point
```

Out[7]:
```
11.02
```

In [8]:
```
5 ** 5    #exponent
```

Out[8]:
```
3125
```

1.1.2 Comments

[go back to the top]

Comments are the explanations of what code is doing. As your programs get more complex, you should add comments, explanations for human readers of your code. For example, here is a comment that explains the value used in a variable:

In [9]:
```
numberOfBooks = 15            #integer type variable
```

The interpreter does not execute comments at all. It ignores everything from a # delimiter to the end of the line.

1.1.3 Strings

[go back to the top]

Programs are also used to process text, not just numbers. Text consists of characters: letters, numbers, punctuation, spaces, and so on. A string is a sequence of characters. For example, the string `"Hello"` is a sequence of five characters.

A string can be stored in a variable as:

In [10]:
```
userGreeting = "Hello User"
```

and later accessed when needed just as numerical values can be:

In [11]:
```
print(userGreeting)
```

```
Hello User
```

A **string literal** denotes a particular string (such as "Hello"), just as a number literal (such as 2) denotes a particular number. In Python, string literals are specified by enclosing a sequence of characters within a matching pair of either single or double quotes.

In [12]:
```python
print("This is a string.", 'This aswell.', """Also this""")
```

```
This is a string. This aswell. Also this
```

By allowing both types of delimiters, Python makes it easy to include an apostrophe or quotation mark within a string.

In [13]:
```python
message = 'He said "Hello Ali"'
print(message)
```

```
He said "Hello Ali"
```

The number of characters in a string is called the *length* of the string. For example, the length of "Harry" is 5. You can compute the length of a string using Python's `len` function:

In [14]:
```python
length = len("Rasikh Ali")    # Length is 6
print(length)
```

```
10
```

A string of length 0 is called the *empty string*. It contains no characters and is written as "" or ''.

Concatenation and repetition

Given two strings, such as "Harry" and "Morgan", you can **concatenate** them to one long string. The result consists of all characters in the first string, followed by all characters in the second string. In Python, you use the `+` operator to concatenate two strings. For example,

In [15]:
```python
firstName = "Rasikh"
lastName = "Ali"

name = firstName + " " + lastName
print(name)
```

```
Rasikh Ali
```

Note how we have concatenated a space between the two strings in the assignment to the **name** variable above.

When the expression to the left or the right of a + operator is a string, the other one must also be a string or a syntax error will occur. You cannot concatenate a string with a numerical value.

You can also produce a string that is the result of repeating a string multiple times. For example, suppose you need to print a dashed line. Instead of specifying a literal string with 50 dashes, you can use the `*` operator to create a string that is comprised of the string "-" repeated 50 times. For example,

In [16]:
```python
dashes = "<=>" * 30
print(dashes)
```

```
<=><=><=><=><=><=><=><=><=><=><=><=><=><=><=><=><=><=><=><=><=><=><=><=><=><=><=><=><=><=>
```

A string of any length can be repeated using the `*` operator. For example, the following statements repeat the string "Hello" five times:

In [17]:
```python
message = "Testing Message..."
print(message * 5)
```

```
Testing Message...Testing Message...Testing Message...Testing Message...Testing Message...
```

Strings and characters

Strings are sequences of *Unicode* characters. You can access the individual characters of a string based on their position within the string. This position is called the *index* of the character. The first character has index 0, the second has index 1, and so on. For example, in the string "Harry",

- the character *H* is at index 0,
- the character *a* is at index 1,
- the character *r* is at index 2,
- the character *r* is at index 3,
- and the character *y* is at index 4.

Note that there is **no character** at index **5**.

An individual character is accessed using a special *subscript notation* in which the position is enclosed within square brackets **[ ]**. For example, if the variable name is defined as:

In [18]:
```python
name = "Rasikh"
```

the statements:

In [19]:
```python
first = name[0]
last = name[5]
print(first, last)
```

```
R h
```

extract two different characters from the string. The first statement extracts the first character as the string "H" and stores it in variable `first`. The second statement extracts the character at position 4, which in this case is the last character, and stores it in variable `last`.

The index value must be within the valid range of character positions or an *index out of range* exception will be generated at run time.

By using this subscript notation, we can also extract a *substring* from a string. For example,

In [20]:
```
middle = name[1:5]
print(middle)
```

```
asik
```

We get the string "arr", starting from index 1 and up to index 3. That is, it *excludes the last index 4*.

Updating strings

Just like other variables, the value in a string variable can be changed or replaced by assigning the string variable a new string value. For example,

In [21]:
```
name = "Rasikh Ali"
print(name)
```

```
Rasikh Ali
```

This will replace the earlier value "Harry" by "John Doe" in the **name** string variable.

> Note that a value stored in a string variable can only be replaced with another string literal.

We can also use the substring operation and the `+` operator to replace a string value as follows:

In [22]:
```
originalString = "Testing Message!"
updatedString = originalString[:8] + "Python"
print("Updated string: ", updatedString)
```

```
Updated string:  Testing Python
```

This will replace the "Hello World!" placed in the variable **originalString** by taking a substring consisting of the first 6 characters, including the space character, and adding this substring with another string "Python" to produce the **updatedString** "Hello Python".

String methods

Python strings are created as **objects**, that is, they have values as well as certain behaviors. The value can be simple, such as a string, or the number of characters stored in a string. The behavior of an object is given through its **methods**. A method, like a function, is a collection of programming instructions that carry out a particular task.

There are many built-in methods that can be applied to strings in Python. For example, you can apply the upper method to any string, like this:

In [23]:
```
name = "Rasikh Ali"
uppercaseName = name.upper() # Sets uppercaseName to "JOHN SMITH"
print(uppercaseName)
```

```
RASIKH ALI
```

Note that the method name follows the object, and that a dot (.) separates the object and method name.

There is another string method called lower that yields the lowercase version of a string:

In [24]:
```
print(name.lower()) # Prints john smith
```

```
rasikh ali
```

Deleting strings

To delete a string, we will use a function called `del` as follows:

In [25]:
```
del name
```

This deletes the object reference placed in the **name** variable. Now, any attempt to access this variable will result in an error:

In [26]:
```
print(name)    #error: name has been deleted, so it is no longer defined
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[26], line 1
----> 1 print(name)

NameError: name 'name' is not defined
```

Note that we can reuse string variables after applying the function `del` by assigning them new values again:

In [27]:
```
name = "Rasikh Ali"
print(name)
```

```
Rasikh Ali
```

Now, we can reuse the variable **name** in the rest of the code.

Finding characters in a string

We can use the **in** operator to find whether any character exists in a string. This operator will return **True** if the character exists at any location in a string, **False**

otherwise.

In [28]:
```
name = "Rasikh Ali"

print('s' in name)
print('S' in name)

exists = 'i' in name
print(exists)
```

```
True
False
True
```

Note how we have used variable **exists** to get the output of the **in** operator and store it in the **exists** variable. We can also use the logical operator **not** to reverse the output of the **in** operator. That is, it will return **True** if the specified character does not exist in the string, and **False** otherwise:

In [29]:
```
print('B' not in name)
print('h' not in name)
```

```
True
False
```

As we can see from the output above, both the characters *B* and *u* do not exist in the string literal "John Smith", so we get the output of **True** in both of above statements.

Formatting strings

We may use the special string **substitution** placeholders, or **format specifiers** while working with strings. Two format specifiers are **%s** that is used to replace a string, and **%d** that is used to replace an integer with strings. For example:

In [30]:
```
print("My name is %s and I am %d years old" % ("Rasikh Ali", 23))
```

```
My name is Rasikh Ali and I am 23 years old
```

In the above example, **%s** placeholder is replaced with the string **John Smith**, and **%d** placeholder is replaced with the number **35**.

> Placeholders must be correctly specified and the values must be correctly defined, otherwise errors will be reported. For example, in the above example, **%s** would need a matching string value to be defined. Any other value would be an error.

For example, the below code produces an error as **%d** is being replaced with a value that is a string:

In [31]:
```
print("My name is %s and I am %d years old" % (23, "Rasikh Ali"))   #error: %d expects a number!
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[31], line 1
----> 1 print("My name is %s and I am %d years old" % (23, "Rasikh Ali"))

TypeError: %d format: a number is required, not str
```

1.2 Decisions

[go back to the top]

The **if** statement is used to implement a decision. When a condition is fulfilled, one set of statements is executed. Otherwise, another set of statements is executed.

Some constructs in Python are **compound statements**, which span multiple lines and consist of a *header* and a **statement block**. The **if** statement is an example of a compound statement.

Let's take an example that whenever at a store, if the customer's total sales are more than **100**, the store would give a discount of **5%** on the total sales amount. The printed receipt for a customer reflects this discount. If the customer's total sales are less than **100**, the store does not give any discount.

We will write the corresponding code in Python as follows:

In [32]:
```
totalSales = 50
if totalSales > 100.0 :     # The header ends in a colon.
    discount = totalSales * 0.05     # Lines in the block are indented to the same level
    totalSales = totalSales - discount
    print("You received a discount of", discount)
else:
    print("You sales are less than 100, there are no discounts")
```

```
You sales are less than 100, there are no discounts
```

Compound statements require a colon (:) at the end of the header. The statement block is a group of one or more statements, all of which are indented to the same indentation level. A statement block begins on the line following the header and ends at the first statement indented less than the first statement in the block. You can use any number of spaces to indent statements within a block, but all statements within the block must have the same indentation level. Note that comments are not statements and thus can be indented to any level.

1.2.1 Relational operators

[go back to the top]

Every if statement contains a condition. In many cases, the condition involves comparing two values. For example, in the previous examples we tested `totalSales > 100.0`. The comparison **>** is called a **relational operator**. Python has six relational operators.

- **>** (Greater-than)
- **<** (Less-than)
- **>=** (Greater-than or equals to)
- **<=** (Less-than or equals to)
- **==** (Equals to)
- **!=** (Not equals to)

All relational operators return **True** if the comparison results in true, otherwise **False**. We can compare numbers and strings using the relational operators. For example,

In [33]:
```
print(1 < 3)
```

```
True
```

In [34]:
```
print(15 <= 23)
```

```
True
```

In [35]:
```
"Rasikh Ali" == "Ali Rasikh"
```

Out[35]:
```
False
```

In [36]:
```
"Rasikh Ali" != "Ali Rasikh"
```

Out[36]:
```
True
```

We can also use **Boolean operators** to combine and compare multiple conditions at the same time. There are three **Boolean operators: and, or,** and **not**.

The condition of the test has two parts, joined by the **and** operator. Each part is a Boolean value that can be true or false. The combined expression is true if both individual expressions are true. If either one of the expressions is false, then the result is also false.

In [37]:
```
(1 == 1) and (5 < 2)
```

Out[37]:
```
False
```

For the **or** operator, the combined expression is true if any individual expression is true. If neither one of the expressions is true, then the result is also false.

In [38]:
```
(1 < 0) or (2 > 3)
```

Out[38]:
```
False
```

The **not** operator simply inverts the result - that is if the result of some comparison is true, it returns false.

In [39]:
```
not(1 > 2)
```

Out[39]:
```
True
```

We can use these operators while writing if statements:

In [40]:
```
a = 100
b = 200
if a < b or b > 100:
    print("Save")
else:
    print("Cancel")
```

```
Save
```

1.3 Loops

[go back to the top]

In Python, the **while** statement implements a repetition. It has the form:

```
while condition :     statements
```

As long as the condition remains true, the statements inside the while statement are executed. This statement block is called the body of the while statement. For example,

In [41]:
```
timer = 1
while timer <= 10 :
    print(timer)
    timer = timer + 1
```

```
1
2
3
4
5
6
7
8
9
10
```

Often, we will need to visit each character in a string.

The **for** loop makes this process particularly easy to program. For example, suppose we want to print a string, with one character per line. We cannot simply print the string using the print function. Instead, we need to iterate over the characters in the string and print each character individually. Here is how you use the for loop to accomplish this task:

In [42]:
```
cityName = "Lahore"
for letter in cityName :
    print(letter)
```

```
L
a
h
o
r
e
```

Loops that iterate over a range of integer values are very common. To simplify the creation of such loops, Python provides the **range** function for generating a sequence of integers that can be used with the for loop. The loop:

In [43]:
```
for i in range(1, 10) :        # i = 1, 2, 3, ..., 9
    print(i)
```

```
1
2
3
4
5
6
7
8
9
```

prints the sequential values from 1 to 9. The range function generates a sequence of values based on its arguments. The first argument of the range function is the first value in the sequence. Values are included in the sequence while they are less than the second argument.

> Note that the ending value (the second argument to the range function) is not included in the output sequence.

Another example of calculating the factorial of a number using the **for** loop may be written as:

In [44]:
```
fact = 1
N = 5
for i in range (1, N + 1) :
    fact *= i    #same as fact = fact * i

print(fact)
```

```
120
```

1.4 Lists

[go back to the top]

Lists are the fundamental mechanism in Python for collecting multiple values. Lists are indispensable when multiple values are being stored and there are many operations that are being performed on them.

As an example, suppose that we are storing scores obtained by students in a quiz. Suppose that there are 10 students. To store these scores, we would need to create 10 individual variables. Lets also suppose that we need to find the highest quiz score among all scores. We would need to write a function, say **max()** that finds the highest score among all variables. Now let's extend this problem to 100 students. There are now 100 scores that are needed to be stored. We would need to rewrite the **max()** function so that now it handles 100 scores. This is not a good program as it does not extend or scale appropriately. Also, to make it work effectively with data as the data size increases, we would need to make major modifications in this **max()** function.

One way to remedy this scenario is by using a *List*. We now take a look on how to create and use Lists.

> a list is an ordered collection of data that is referred to by a single variable name. (in most other computer languages, the same concept is called an array.)

> Lists are objects in Python. They provide many useful methods for manipulating list elements.

> Whenever we are looking to work with multiple values, we should use Lists.

Creating lists

We create a list and specify the initial values that are to be stored in the new list:

In [45]:
```
scores = [54, 67.5, 80, 95]    #list containing 10 elements
print(scores)

names = ["Rasikh", "Ali", "Ahmed", "Muneeb"]
print(names)

temperatures = [32, 33.5, 34, 35]
print(temperatures)
```

```
[54, 67.5, 80, 95]
['Rasikh', 'Ali', 'Ahmed', 'Muneeb']
[32, 33.5, 34, 35]
```

The square brackets indicate that we are creating a list. The items are stored as comma-separated and in the order they are provided. You will want to store the list in a variable so that you can access and manipulate it later.

> In a list, order is important, and the order of the elements in a list never changes (unless you explicitly do so). Because the order of elements in a list is important, you refer to each element in a list using its index (its position within the list).

Accessing List elements

A list is a sequence of elements, each of which has an integer position or index. To access a list element, you specify which index you want to use. That is done with the **subscript operator ([ ])** in the same way that you access individual characters in a string. Indexes in Lists always start with zero: that is the first element in a List will be at index 0.

For example,

In [46]:
```
print(scores[2])    #Prints the element at index 2
```

```
80
```

Both lists and strings are sequences, and the [ ] operator can be used to access an element in any sequence.

There are two differences between lists and strings. Lists can hold values of any type, whereas strings are sequences of characters. Moreover, strings are immutable—you cannot change the characters in the sequence. But lists are mutable. You can replace one list element with another, like this:

In [47]:
```
scores[2] = 87   #replace the value at index 2 with 87
print(scores[2])
```

```
87
```

Now the element at index 2 is filled with 87.

> Trying to access an element that does not exist in the list is a serious error. For example, if values has ten elements, you are not allowed to access values[20]. Attempting to access an element whose index is not within the valid index range is called an **out-of-range error** or a **bounds error**. When an **out-of-range error** occurs at run time, it causes a runtime exception.

We can also select a range or specific values from the list:

In [48]:
```
print("scores[1]:", scores[1])   #prints value at index 1 (54)
print("scores[1:3]: ", scores[1:3])   #prints starting index 1 to 2 (ignores index 3)!
```

```
scores[1]: 67.5
scores[1:3]:  [67.5, 87]
```

We can use the `len` function to obtain the length of the list; that is, the number of elements:

In [49]:
```
print(len(scores))    #prints the number of elements in a list

listSize = len(scores)  #stores the number of elements in a variable
print(listSize)
```

```
4
4
```

Updating List elements

We can update any list element by assigning the list index a new value as follows:

In [50]:
```
subjects = ["Physics", "Computer Science", 1997, 2000]
print(subjects)

subjects[0] = "Discrete Mathematics"   #replace value at index 0
print(subjects)
```

```
['Physics', 'Computer Science', 1997, 2000]
['Discrete Mathematics', 'Computer Science', 1997, 2000]
```

We can also use the `append()` method to add elements in a list. The `append()` method always adds elements at the end of a List.

In [51]:
```
print("List size before appending new element: ", len(subjects))
subjects.append("Psychology")
print(subjects)
print("List size after appending new element: ", len(subjects))

print()   #blank line

subjects.append(2018)
print(subjects)
```

```
List size before appending new element:  4
['Discrete Mathematics', 'Computer Science', 1997, 2000, 'Psychology']
List size after appending new element:  5

['Discrete Mathematics', 'Computer Science', 1997, 2000, 'Psychology', 2018]
```

The size, or length, of the list increases after each call to the append method. Any number of elements can be added to a list.

If the order of the elements does not matter, appending new elements is sufficient. Sometimes, however, the order is important and a new element has to be inserted at a specific position in the list.

Suppose that in the above **subjects** list, we would like to add a new element at the very beginning or index 0 of the list. The **insert** method of the List is used for this purpose. While using the **insert** method, first the index where the new value is to be inserted is given, then the new value is given.

For example,

In [52]:
```
print(subjects, "Length: ", len(subjects))
print()

print("Inserting new value at index 0")
subjects.insert(0, 2009)

print()
print(subjects, "Length: ", len(subjects))
```

```
['Discrete Mathematics', 'Computer Science', 1997, 2000, 'Psychology', 2018] Length:  6

Inserting new value at index 0

[2009, 'Discrete Mathematics', 'Computer Science', 1997, 2000, 'Psychology', 2018] Length:  7
```

All of the elements at and following position 0 are moved down by one position to make room for the new element, which is inserted at position 0. After each call to the insert method, the size of the list is increased by 1.

Removing an element

The **pop** method removes the element at a given position. For example, suppose we start with the list:

In [53]:
```
friends = ["Rasikh", "Ali", "Ahmed", "Muneeb"]
```

To remove the element at index position 1 ("Cindy") in the friends list, you use the command:

In [54]:
```
friends.pop(1)
```

Out[54]:
```
'Ali'
```

All of the elements following the removed element are moved up one position to close the gap. The size of the list is reduced by 1. The index passed to the **pop** method must be within the valid range.

The element removed from the list is returned by the **pop** method. This allows you to combine two operations in one—accessing the element and removing it:

In [55]:
```
print("The removed item is", friends.pop(1))
print("List now contains elements: ", friends)
```

```
The removed item is Ahmed
List now contains elements:  ['Rasikh', 'Muneeb']
```

If you call the **pop** method without an argument, it removes and returns the last element of the list. For example, `friends.pop()` removes "Bill".

Similar to the **pop** method, we can also use the **del** function to remove an element from a list by giving the index of the element to be removed as a subscript with the name of the list. For example,

In [56]:
```
print("Before the removal of the first element: ", friends)
del friends[0]    #removes the first element

print()
print("After the removal of the first element: ", friends)
```

```
Before the removal of the first element:  ['Rasikh', 'Muneeb']

After the removal of the first element:  ['Muneeb']
```

The **remove** method removes an element by value instead of by position. For example, suppose we want to remove the string "Cari" from the friends list but we do not know where it's located in the list. Instead of having to find the position, we can use the **remove** method:

In [57]:
```
friends.remove("Muneeb")
print("List now contains elements: ", friends)
```

```
List now contains elements:  []
```

1.5 Sets

[go back to the top]

A set is a container that stores a collection of **unique** values. Unlike a list, the elements or members of the set are *not stored in any particular order* and *cannot be accessed by position*. The operations available for use with a set are the same as the operations performed on sets in mathematics. Because sets need not maintain a particular order, set operations are much faster than the equivalent list operations.

Because sets cannot have multiple occurrences of the same element, it makes sets highly useful to efficiently remove duplicate values from a list or tuple and to perform common math operations like unions and intersections.

> Whenever we are looking to work with multiple distinct or unique values, we should use Sets instead of Lists.

Creating and using sets

To create a set with initial elements, you can specify the elements enclosed in braces, just like in mathematics:

In [58]:
```
colors = { "Red", "Green", "Blue" }
print(colors)

points = { 5, 10, 6, 2, 8, 3 }
print(points)

addresses = { "123-Elm Street", 100, "101 Main Blvd", 25.5 }
print(addresses)
```

```
{'Green', 'Blue', 'Red'}
{2, 3, 5, 6, 8, 10}
{'123-Elm Street', '101 Main Blvd', 100, 25.5}
```

Elements of a set are comma separated. We can also have an empty Set as:

In [59]:
```
emptySet = {}
print(emptySet)
```

```
{}
```

Alternatively, you can use the **set** function to convert any sequence into a set:

In [60]:
```
names = ["Rasikh", "Ali", "Ahmed", "Muneeb"]
cast = set(names)

print(cast, "Size of set: ", len(cast))
```

```
{'Rasikh', 'Ahmed', 'Ali', 'Muneeb'} Size of set:  4
```

As with any container, you can use the **len** function to obtain the number of elements in a set:

In [61]:
```
numberOfCharacters = len(cast)
print(numberOfCharacters)
```

```
4
```

To determine whether an element is contained in the set, use the **in** operator or its inverse, the **not in** operator:

In [62]:
```
if "Rasikh" in cast :
    print("Rasikh is included in the set of cast.")
else :
    print("Rasikh is not a character in the show.")
```

```
Rasikh is included in the set of cast.
```

Because sets are unordered, you cannot access the elements of a set by position as you can with a list. Instead, use a **for** loop to iterate over the individual elements:

In [63]:
```
print("The cast of characters includes:")
for character in cast :
    print(character)
```

```
The cast of characters includes:
Rasikh
Ahmed
Ali
Muneeb
```

> Note that the order in which the elements of the set are visited depends on how they are stored internally.

Adding and removing elements

Like lists, sets are mutable collections, so you can add and remove elements. For example, suppose we need to add more characters to the set cast created in the previous section. Use the **add** method to add elements:

In [64]:
```
cast.add("Musahaf")
print(cast, "Size: ", len(cast))
```

```
{'Rasikh', 'Muneeb', 'Ahmed', 'Musahaf', 'Ali'} Size:  5
```

If the element being added is not already contained in the set, it will be added to the set and the size of the set increased by one. Remember, however, that a set cannot contain duplicate elements. If you attempt to add an element that is already in the set, there is no effect and the set is not changed.

In [65]:
```
cast.add("Anas")
print(cast)
```

```
{'Rasikh', 'Muneeb', 'Ahmed', 'Musahaf', 'Anas', 'Ali'}
```

The **update** function in set adds elements from a set (passed as an argument) to the set, that is, it can add multiple elements in a set. Any duplicates elements are ignored during the **update**.

In [66]:
```
list1 = [1, 2, 3]
list2 = [3, 5, 6, 7]
list3 = [10, 11, 12]

# Lists converted to sets
set1 = set(list2)
set2 = set(list1)

# Update method
set1.update(set2)

# Print the updated set
print(set1)

# List is passed as an parameter which gets automatically converted to a set
set1.update(list3)
print(set1)
```

```
{1, 2, 3, 5, 6, 7}
{1, 2, 3, 5, 6, 7, 10, 11, 12}
```

There are two methods that can be used to remove individual elements from a set. The **discard** method removes an element if the element exists.

In [67]:
```
cast.discard("Muneeb")
print(cast, "Size: ", len(cast))
```

```
{'Rasikh', 'Ahmed', 'Musahaf', 'Anas', 'Ali'} Size:  5
```

but has no effect if the given element is not a member of the set:

In [68]:
```
cast.discard("Aliyan")     # Has no effect
print(cast)
```

```
{'Rasikh', 'Ahmed', 'Musahaf', 'Anas', 'Ali'}
```

The **remove** method, on the other hand, removes an element if it exists, but raises an exception if the given element is not a member of the set:

In [69]:
```
cast.remove("Aliyan")     # Raises an exception
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[69], line 1
----> 1 cast.remove("Aliyan")

KeyError: 'Aliyan'
```

Finally, the **clear** method removes all elements of a set, leaving the empty set:

In [70]:
```
cast.clear()    # cast now has size 0
print(cast, "Size: ", len(cast))
```

```
set() Size:  0
```

Set Union, Intersection, and Difference

The union of two sets contains all of the elements from both sets, with duplicates removed. Use the **union** method to create the union of two sets in Python. For example:

In [71]:
```
canadian = { "Red", "White" }         # flag colors
british = { "Red", "Blue", "White" }  #flag colors
italian = { "Red", "White", "Green" } #flag colors

inEither = british.union(italian) # The set {"Blue", "Green", "White", "Red"}
print(inEither, "Size: ", len(inEither))
```

```
{'Green', 'Blue', 'Red', 'White'} Size:  4
```

Both the british and italian sets contain the colors "Red" and "White", but the union is a set and therefore contains only one instance of each color.

> Note that the union method returns a new set. It does not modify either of the sets in the call.

The intersection of two sets contains all of the elements that are in both sets. To create the intersection of two Python sets, use the **intersection** method:

In [72]:
```
inBoth = british.intersection(italian)    # The set {"White", "Red"}
print(inBoth, "Size: ", len(inBoth))
```

```
{'White', 'Red'} Size:  2
```

Finally, the difference of two sets results in a new set that contains those elements in the first set that are not in the second set. For example, the difference

between the Italian and the British colors is the set containing only "Green".

Use the **difference** method to find the set difference:

In [73]:
```
print("Colors that are in the Italian flag but not the British:")
print(italian.difference(british)) # Prints {'Green'}
```

```
Colors that are in the Italian flag but not the British:
{'Green'}
```

When forming the union or intersection of two sets, the order does not matter. For example, `british.union(italian)` is the same set as `italian.union(british)`. But the order matters with the **difference** method. The set returned by:

In [74]:
```
print(british.difference(italian))
```

```
{'Blue'}
```

is {"Blue"}.

1.6 Dictionaries

[go back to the top]

A dictionary is a container that keeps associations between *keys* and *values*. Every key in the dictionary has an associated value. Keys are unique, but a value may be associated with several keys. The dictionary structure is also known as a *map* because it maps a unique key to a value. It stores the keys, values, and the associations between them.

> We will use a dictionary in all application where a value to be searched is associated with a particular and unique key. For example, in a contact list, a person's name acts as a key that is associated with one or more numbers. Similarly, the address book can also said to be an example of a dictionary.

> In all applications, dictionary searches will always outperform set or list searches as the keys are stored in a way to optimize search on keys.

> Hashing is an example of a dictionary implementation, and is an example of an instantaneous search costing O(1).

Creating dictionaries

Suppose you need to write a program that looks up the phone number for a person in your mobile phone's contact list. You can use a dictionary where the names are keys and the phone numbers are values. The dictionary also allows you to associate more than one person with a given number.

Here we create a small dictionary for a contact list that contains four items:

In [75]:
```
contacts = { "Rasikh": 7235591, "Ali": 3841212, "Muneeb": 3841212, "Ahmed": 2213278 }
print(contacts, "Size: ", len(contacts))
```

```
{'Rasikh': 7235591, 'Ali': 3841212, 'Muneeb': 3841212, 'Ahmed': 2213278} Size:  4
```

We can have other examples of dictionaries as follows:

In [76]:
```
grades = { "A": 4, "B": 3, "C": 2.5, "D": 2 }
print(grades, "Size: ", len(contacts))

constants = { "PI": 3.14159, "g": 9.8, "K": 212}
print(constants, "Size: ", len(contacts))
```

```
{'A': 4, 'B': 3, 'C': 2.5, 'D': 2} Size:  4
{'PI': 3.14159, 'g': 9.8, 'K': 212} Size:  4
```

Each key/value pair is separated by a colon. You enclose the key/value pairs in braces, just as you would when forming a set. When the braces contain key/value pairs, they denote a dictionary, not a set. The only ambiguous case is an empty { }. By convention, it denotes an empty dictionary, not an empty set.

You can create a duplicate copy of a dictionary using the **dict** function:

In [77]:
```
oldContacts = dict(contacts)
print(oldContacts)
```

```
{'Rasikh': 7235591, 'Ali': 3841212, 'Muneeb': 3841212, 'Ahmed': 2213278}
```

Accessing dictionary values

The subscript operator [ ] is used to return the value associated with a key. The statement:

In [78]:
```
print("Ali's number is", contacts["Ali"])
```

```
Ali's number is 3841212
```

prints `3841212`.

Note that the dictionary is not a sequence-type container like a list. Even though the subscript operator is used with a dictionary, you cannot access the items by index or position. A value can only be accessed using its associated key.

The key supplied to the subscript operator must be a valid key in the dictionary or a `KeyError` exception will be raised. To find out whether a key is present in the dictionary, use the **in** (or **not in**) operator:

In [79]:
```
if "Ahmed" in contacts :
    print("Ahmed's number is", contacts["Ahmed"])
else :
    print("Ahmed is not in my contact list.")
```

```
Ahmed's number is 2213278
```

Adding and modifying items

A dictionary is a mutable container. That is, you can change its contents after it has been created. You can add a new item using the subscript operator [ ] much as you would with a list:

In [80]:
```
contacts["Ahmed"] = 4578102
print(contacts)
```

```
{'Rasikh': 7235591, 'Ali': 3841212, 'Muneeb': 3841212, 'Ahmed': 4578102}
```

To change the value associated with a given key, set a new value using the [ ] operator on an existing key:

In [81]:
```
contacts["Ahmed"] = 2228102
print(contacts)
```

```
{'Rasikh': 7235591, 'Ali': 3841212, 'Muneeb': 3841212, 'Ahmed': 2228102}
```

Sometimes you may not know which items will be contained in the dictionary when it's created. You can create an empty dictionary like this:

In [82]:
```
favoriteColors = {}
```

and add new items as needed:

In [83]:
```
favoriteColors["Rasikh"] = "Blue"
favoriteColors["Ahmed"] = "Red"
favoriteColors["Ali"] = "Blue"
favoriteColors["Anas"] = "Green"

print(favoriteColors)
```

```
{'Rasikh': 'Blue', 'Ahmed': 'Red', 'Ali': 'Blue', 'Anas': 'Green'}
```

Removing items

To remove an item from a dictionary, call the **pop** method with the key as the argument:

In [84]:
```
contacts.pop("Ali")
print(contacts)
```

```
{'Rasikh': 7235591, 'Muneeb': 3841212, 'Ahmed': 2228102}
```

This removes the entire item, both the key and its associated value. The **pop** method returns the value of the item being removed, so you can use it or store it in a variable:

In [85]:
```
ahmedNumber = contacts.pop("Ahmed")
print(ahmedNumber)
```

```
2228102
```

If the key is not in the dictionary, the **pop** method raises a `KeyError` exception. To prevent the exception from being raised, you can test for the key in the dictionary:

In [86]:
```
if "Rasikh" in contacts :
    contacts.pop("Rasikh")
else:
    print("Rasikh not present in the dictionary")
```

Traversing a Dictionary

You can iterate over the individual keys in a dictionary using a for loop:

In [87]:
```
print("My Contacts:")
for key in contacts :
    print(key)
```

```
My Contacts:
Muneeb
```

Note that the dictionary stores its items in an order that is optimized for efficiency, which may not be the order in which they were added. To access the value associated with a key in the body of the loop, you can use the loop variable with the subscript operator. For example, these statements print both the name and phone number of your contacts:

In [88]:
```
print("My Contacts:")
for key in contacts :
    print("%-10s %d" % (key, contacts[key]))
```

```
My Contacts:
Muneeb     3841212
```

1.7 Tuples

[go back to the top]

Python provides a data type for immutable sequences of arbitrary data. A tuple is very similar to a list, but once created, its contents cannot be modified.

> A tuple is essentially a list that cannot be changed.

> A tuple is a fixed-length, immutable sequence of Python objects.

A tuple is created by specifying its contents as a comma-separated sequence. You can enclose the sequence in parentheses:

In [89]:
```
triple = (5, 10, 15)
print(triple, "Size: ", len(triple))

data = ("Maths", 98, "Programming", 99, "Research")
print(data, "Size: ", len(data))
```

```
(5, 10, 15) Size:  3
('Maths', 98, 'Programming', 99, 'Research') Size:  5
```

Tuples can contain elements of various types. If you prefer, you can omit the parentheses:

In [90]:
```
triple = 5, 10, 15
```

> The difference between lists and tuples are that tuples cannot be changed.

> For all applications where we need to store immutable data, tuples should be preferred over lists.

Each item stored in a tuple is comma-separated. We can create empty tuples as:

In [91]:
```
counter = ()
print(counter)
```

```
()
```

Accessing values in tuples

We can access an individual element of a tuple by its position using the **[ ]** operator. For example,

In [92]:
```
element = triple[1]
print(element)

print(data[0:3])
```

```
10
('Maths', 98, 'Programming')
```

We can iterate over the elements of a tuple using for loops.

In [93]:
```
for i in range(1, 3) :
    print(data[i])
```

```
98
Programming
```

Updating tuples

As tuples are immutable, that means that we cannot change any element in a tuple. For example, the following action is not allowed on a tuple:

In [94]:
```
data[0] = "Physics"   #error: no changes are allowed!
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[94], line 1
----> 1 data[0] = "Physics"

TypeError: 'tuple' object does not support item assignment
```

We can use other tuples to create new tuples. For example,

In [95]:
```
colors1 = ("Red", "Green", "Blue")
colors2 = ("Cyan", "Magenta", "Yellow", "Kelvin")

colors = colors1 + colors2
print(colors)
```

```
('Red', 'Green', 'Blue', 'Cyan', 'Magenta', 'Yellow', 'Kelvin')
```

Deleting tuples

As tuples are immutable, we cannot delete any element from a tuple. We can however, delete a whole tuple using the **del** function:

In [96]:
```
del colors

print(colors)   #error: no tuple
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[96], line 3
      1 del colors
----> 3 print(colors)

NameError: name 'colors' is not defined
```

When to use tuples?

When a list is represented as a tuple, Python internally organizes the data in a way that it can access each individual element faster than in a list. Therefore, if you want to write code that runs as fast as possible, then look for any case where you have a list that never changes in your program. You can redefine it from a list to a tuple by changing the square brackets to parentheses. Eventually, this concept becomes second nature. You start thinking of unchanging lists as tuples and define them that way right from the start.

There is one additional small benefit to using a tuple. If you have a list of data and you want to ensure that there is no code that makes any changes to it, use a tuple. Any code that attempts to append to, delete from, or modify an element of a tuple will generate an error message. The offending code can quickly be identified and corrected.

1.8 Functions

[go back to the top]

A function is a sequence of instructions with a name. You have already encountered several functions during the discussion of various topics above. For example, the `round` function contains instructions to round a floating-point value to a specified number of decimal places. You *call* a function in order to execute its instructions. For example, consider the following program statement:

In [97]:
```
price = round(6.8275, 2) # Sets result to 6.83
print(price)
```

```
6.83
```

By using the expression `round(6.8275, 2)`, the program *calls* the round function, asking it to round 6.8275 to two decimal digits. The instructions of the `round` function execute and compute the result. The `round` function *returns* its result back to where the function was called and the program resumes execution.

When another function calls the `round` function, it provides "inputs", such as the values 6.8275 and 2 in the call `round(6.8275, 2)`. These values are called the **arguments** of the function call. Note that they are not necessarily inputs provided by a human user. They are simply the values for which we want the function to compute a result. The "output" that the round function computes is called the **return value**.

Functions can receive multiple arguments, but they return only one value. It is also possible to have functions with no arguments. An example is the `random` function that requires no argument to produce a random number.

The return value of a function is returned to the point in your program where the function was called. It is then processed according to the statement containing the function call. For example, suppose your program contains a statement:

In [98]:
```
price = round(6.8275, 2)
print(price)
```

```
6.83
```

When the round function returns its result, the return value is stored in the variable price.

Implementing a function

We will start with a very simple example: a function to compute the volume of a cube with a given side length. When writing this function, you need to:

- Pick a name for the function (cubeVolume).
- Define a variable for each argument (sideLength). These variables are called the **parameter variables**.

Put all this information together along with the def reserved word to form the first line of the function's definition:

```
def cubeVolume(sideLength) :
```

This line is called the **header** of the function. Next, specify the **body** of the function. The body contains the statements that are executed when the function is called.

The volume of a cube of side length s is s × s × s. However, for greater clarity, our parameter variable has been called `sideLength`, not `s`, so we need to compute `sideLength ** 3`.

We will store this value in a variable called `volume`:

```
volume = sideLength ** 3
```

In order to return the result of the function, use the `return` statement:

```
return volume
```

A function is a compound statement, which requires the statements in the body to be indented to the same level. Here is the complete function:

In [99]:
```
def cubeVolume(sideLength) :
    volume = sideLength ** 3
    return volume
```

Testing a function

In the preceding section, you saw how to write a function. If you run a program containing just the function definition, then nothing happens. After all, nobody is calling the function.

In order to test the function, your program should contain:

- The definition of the function.
- Statements that call the function and print the result.

Here is such a program:

In [100…
```
def cubeVolume(sideLength) :
    volume = sideLength ** 3
    return volume

result1 = cubeVolume(2)
result2 = cubeVolume(10)

print("A cube with side length 2 has volume", result1)
print("A cube with side length 10 has volume", result2)
```

```
A cube with side length 2 has volume 8
A cube with side length 10 has volume 1000
```

Note that the function returns different results when it is called with different arguments. Consider the call `cubeVolume(2)`. The argument 2 corresponds to the `sideLength` parameter variable. Therefore, in this call, sideLength is 2. The function computes `sideLength ** 3`, or `2 ** 3`. When the function is called with a different argument, say 10, then the function computes `10 ** 3`.

Programs that contain functions

When you write a program that contains one or more functions, you need to pay attention to the order of the function definitions and statements in the program. Have another look at the program of the preceding section. Note that it contains

- The definition of the `cubeVolume` function.
- Several statements, two of which call that function.

As the Python interpreter reads the source code, it reads each function definition and each statement. The statements in a function definition are not executed until the function is called. Any statement not in a function definition, on the other hand, is executed as it is encountered. Therefore, it is important that you define each function before you call it. For example, the following will produce a compile-time error:

In [101…
```
print(cubeVolume(10))

def cubeVolume(sideLength) :
    volume = sideLength ** 3
    return volume
```

```
1000
```

The compiler does not know that the cubeVolume function will be defined later in the program.

However, a function can be called from within another function before the former has been defined. For example, the following is perfectly legal:

In [102…
```
def main() :
    result = cubeVolume(2)
    print("A cube with side length 2 has volume", result)

def cubeVolume(sideLength) :
    volume = sideLength ** 3
    return volume

main()
```

```
A cube with side length 2 has volume 8
```

Parameter passing

When a function is called, variables are created for receiving the function's arguments. These variables are called **parameter variables**. (Another commonly used term is **formal parameters**.) The values that are supplied to the function when it is called are the **arguments** of the call. (These values are also commonly called the **actual parameters**.) Each parameter variable is initialized with the corresponding argument.

Consider the function call:

In [103…
```
result1 = cubeVolume(2)
```

- The parameter variable sideLength of the cubeVolume function is created when the function is called.
- The parameter variable is initialized with the value of the argument that was passed in the call. In our case, `sideLength` is set to 2.
- The function computes the expression `sideLength ** 3`, which has the value 8. That value is stored in the variable `volume`.
- The function returns. All of its variables are removed. The return value is transferred to the *caller*, that is, the function calling the `cubeVolume` function.

The caller puts the return value in the `result1` variable.

Now consider what happens in a subsequent call, `cubeVolume(10)`. A new parameter variable is created. (Recall that the previous parameter variable was removed when the first call to `cubeVolume` returned.) It is initialized with 10, and the process repeats. After the second function call is complete, its variables are again removed.

Return values

You use the `return` statement to specify the result of a function. In the preceding examples, each `return` statement returned a variable. However, the `return` statement can return the value of any expression. Instead of saving the return value in a variable and returning the variable, it is often possible to eliminate the variable and return the value of a more complex expression:

In [104…
```
def cubeVolume(sideLength) :
    return sideLength ** 3
```

When the return statement is processed, the function exits *immediately*.

1.9 Lambda functions

[go back to the top]

Python has support for so-called *anonymous* or *lambda functions*, which are a way of writing functions consisting of a single statement, the result of which is the return value. They are defined with the **lambda** keyword, which has no meaning other than

"we are declaring an anonymous function":

Lambda functions have the form:

lambda par1, par2, ...: expression

where the expression is the value to be returned. One useful feature of lambda expressions is that they make use of variables from the function in which they are coded.

> Since lambda expressions are functions without a name, they are often referred to as *anonymous functions*.

The following are some examples of regular functions versus lambda functions:

In [105…
```
def short_function(x):
    return x * 2

equiv_anon = lambda x: x * 2
```

In the above code, note how the **lambda** expression replaces the ordinary function. Note also that there are no `return` statements in lambda functions.

Another example of a lamda function as applied on lists:

In [106…
```
def apply_to_list(some_list, f):
    return [f(x) for x in some_list]

ints = [4, 0, 1, 5, 6]
apply_to_list(ints, lambda x: x * 2)
```

Out[106… `[8, 0, 2, 10, 12]`

As another example, suppose you wanted to sort a collection of strings by the number of distinct letters in each string:

In [107…
```
strings = ['foo', 'card', 'bar', 'aaaa', 'abab']
```

Here we could pass a lambda function to the list's sort method:

In [108…
```
strings.sort(key=lambda x: len(set(list(x))))
strings
```

Out[108… `['aaaa', 'foo', 'abab', 'bar', 'card']`

The following program sorts names by their surnames. The second line sorts the list of names, and the last two lines display the contents of the sorted list.

In [109…
```
names = ["Dennis Ritchie", "Alan Kay", "John Backus", "James Gosling"]
names.sort(key=lambda name: name.split()[-1])
nameString = ", ".join(names)
print(nameString)
```

```
John Backus, James Gosling, Alan Kay, Dennis Ritchie
```

> Lambda functions do not include the `return` statement. They always contain expressions that are always returned.

1.9.1 map function

[go back to the top]

The `map()` function provides an easy way to transform each item into an iterable object. The `map` function is frequently used in lambda functions that returns a list of elements. `map` requires two arguments:

```
r = map(function, sequence)
```

The first argument is the name of the function that is to be applied on the second argument, which is the name of a sequence, e.g., list, set etc. `map` will then retruns a new list of elements of sequence that have been changed after applying the function.

For example, here are efficient, compact ways to perform an operation on a sequence. Note the use of the lambda anonymous function:

In [110…
```
lst = [1, 2, 3, 4, 5, 6]
list(map(lambda x: x ** 3, lst))
```

Out[110… `[1, 8, 27, 64, 125, 216]`

In [111…
```
animals = ['hawk', 'hen', 'hedgehog', 'hyena', 'zebra', 'giraffe']
print(animals)

list(map(lambda animal: len(animal), animals))     # apply len() to every animals item
```

```
['hawk', 'hen', 'hedgehog', 'hyena', 'zebra', 'giraffe']
```

Out[111… `[4, 3, 8, 5, 5, 7]`

In [112…
```
numbers = [5, 6, 7, 8, 9, 10, 100, 1000, 10000]
print(numbers)

list(map(lambda n: n % 10, numbers))
```

```
[5, 6, 7, 8, 9, 10, 100, 1000, 10000]
```

Out[112… `[5, 6, 7, 8, 9, 0, 0, 0, 0]`

In [113…
```
sentence = "Weather is beginning to change"
words = sentence.split()
print(words)

lengths = map(lambda word: len(word), words)
list(words)
```

```
['Weather', 'is', 'beginning', 'to', 'change']
```

Out[113… `['Weather', 'is', 'beginning', 'to', 'change']`

1.9.2 filter function

[go back to the top]

The `filter` function provides an elegant way to apply filter on lists. It accepts two arguments: `filter(function, list)`.

The first argument *function* will be applied on every element in the second argument *list*. The first argument *function* returns a **boolean** value of *True* or *False*. The elements of the *list* will only be included in the resultant list if *function* evaluates to *True*.

For example,

In [114…
```
words = ["Hello", "World", "Python", "Great", "OK"]
resultList1 = filter(lambda s: len(s) > 2, words)

list(resultList1)
```

Out[114… `['Hello', 'World', 'Python', 'Great']`

In [115…
```
sports = ('Cricket', 'Soccer', 'Hockey', 'Baseball')
resultList2 = filter(lambda w: len(w) % 2 == 0, sports)

list(resultList2)
```

Out[115… `['Soccer', 'Hockey', 'Baseball']`

In [116…
```
colors = ('Red', 'Green', 'Blue', 'Black')
resultList3 = filter(lambda b: len(b) < 4  , colors)

list(resultList3)
```

Out[116… `['Red']`

In [117…
```
numbers = [5, 6, 7, 8, 9, 10, 100, 1000, 10000]
print(numbers)

list(map(lambda n: n % 10 == 0, numbers))
```

```
[5, 6, 7, 8, 9, 10, 100, 1000, 10000]
```

Out[117… `[False, False, False, False, False, True, True, True, True]`

1.10 File Input / Output (I/O)

[go back to the top]

This section explains how to perform input and output using keyboard and files. We will begin by introducing taking input from keyboard.

Reading input from keyboard: The input function

The `input` function prompts the user to enter data. A typical input statement is:

In [ ]:
```
city = input("Enter the name of your city: ")
print(city)
```

When Python reaches this statement, the string "Enter the name of your city: " is displayed and the program pauses. After the user types in the name of his or her city and presses the Enter (or return) key, the variable town is assigned the name of the city. (If the variable had not been created previously, it is created at this time.) The general form of an input statement is:

variableName = input(prompt)

where prompt is a string that requests a response from the user.

The `input` function always returns a string. However, a combination of an `input` function and an `int` , `float` , or `eval` function allows numbers to be input into a program. For instance, consider the following three statements:

In [ ]:
```
age = int(input("Enter your age: "))    #only integers are accepted
weight = float(input("Enter your weight: "))
height = eval(input("Enter your height: "))   #both integer and floating-point values are accepted
```

Files input and output

We now discuss the common task of reading and writing files that contain text. Examples of text files include not only files created with a simple text editor, such as Windows Notepad, but also Python source code and HTML files.

Opening a file

To access a file, you must first open it. When you open a file, you give the name of the file, or, if the file is stored in a different directory, the file name preceded by the directory path. You also specify whether the file is to be opened for reading or writing. Suppose you want to read data from a file named `input.txt` , located in the same directory as the program. Then you use the following function call to open the file:

In [ ]:
```
infile = open("input.txt", "r")
```

This statement opens the file for reading (indicated by the string argument "r") and returns a *file object* that is associated with the file named `input.txt` . When opening a file for reading, the file must exist or an exception occurs.

The file object returned by the `open` function must be saved in a variable. All operations for accessing a file are made via the file object. To open a file for writing, you provide the name of the file as the first argument to the open function and the string "w" as the second argument:

In [ ]:
```
outfile = open("output.txt", "w")
```

If the output file already exists, it is emptied before the new data is written into it. If the file does not exist, an empty file is created. When you are done processing a file, be sure to close the file using the **close** method:

In [ ]:
```
infile.close()
outfile.close()
```

After a file has been closed, it cannot be used again until it has been reopened. Attempting to do so will result in an exception.

Reading from a file

To read a line of text from a file, call the `readline` method on the `file` object that was returned when you opened the file:

In [ ]:
```
infile = open("input.txt", "r")
line = infile.readline()    #read the first line from the file

print(line)
infile.close()
```

When a file is opened, an input marker is positioned at the beginning of the file. The `readline` method reads the text, starting at the current position and continuing until the end of the line is encountered. The input marker is then moved to the next line. The `readline` method returns the text that it read, including the newline character that denotes the end of the line.

Reading multiple lines of text from a file is very similar to reading a sequence of values with the input function. You repeatedly read a line of text and process it until the sentinel value is reached:

In [ ]:
```
infile = open("input.txt", "r")
line = infile.readline()

while line != "" :
    # Process the line.
    print(line)
    line = infile.readline()
infile.close()
```

The sentinel value is an empty string, which is returned by the `readline` method after the end of file has been reached.

Writing to a file

You can write text to a file that has been opened for writing. This is done by applying the `write` method to the `file` object. For example, we can write the string "Hello, World! from Python" to our output file using the statement:

In [ ]:
```
outfile = open("output.txt", "w")
outfile.write("Hello, World! from Python\n")
outfile.close()
```

Iterating over the lines of a file

You have seen how to read a file one line at a time. However, there is a simpler way. Python can treat an input file as though it were a container of strings in which each line is an individual string. To read the lines of text from the file, you can iterate over the file object using a for loop.

For example, the following loop reads all lines from a file and prints them:

In [ ]:
```
infile = open("input.txt", "r")

for line in infile :
    print(line)

infile.close()
```

Binary files and random access

In the following section, you will learn how to process files that contain data other than text. You will also see how to read and write data at arbitrary positions in a file.

Reading and writing binary files

There are two fundamentally different ways to store data: in text format or binary format. In text format, data items are represented in human-readable form as a sequence of characters. For example, in text form, the integer 12,345 is stored as the sequence of five characters:

"1" "2" "3" "4" "5"

In binary form, data items are represented in bytes. A byte is composed of 8 bits, each of which can be 0 or 1. A byte can denote one of 256 values. To represent larger values, one uses sequences of bytes. Integers are frequently stored as a sequence of four bytes. For example, the integer 123,456 can be stored as:

64 226 1 0

If you load a binary file into a text editor, you will not be able to view its contents. Processing binary files requires programs written explicitly for reading or writing the binary data.

We have to cover a few technical issues about binary files. To open a binary file for reading, use the following command:

`inFile = open(filename, "rb")`

Remember, the second argument to the open function indicates the mode in which the file will be opened. In this example, the mode string indicates that we are opening a binary file for reading. To open a binary file for writing, you would use the mode string "wb":

`outFile = open(filename, "wb")`

Random access

So far, you've read from a file one string at a time and written to a file one string at a time, without skipping forward or backward. That access pattern is called **sequential access**. In many applications, we would like to access specific items in a file without first having to first read all preceding items. This access pattern is called **random access**. There is nothing "random" about random access—the term means that you can read and modify any item stored at any location in the file.

Each file has a special marker that indicates the current position within the file. This marker is used to determine where the next string is read or written. You can move the file marker to a specific position within the file. To position the marker relative to the beginning of the file, you use the method call:

`inFile.seek(position)`

To determine the current position of the file marker (counted from the beginning of the file), use:

`position = inFile.tell() # Get current position.`

For example,

In [ ]:
```
infile = open("input.txt", "r")

str = infile.read(10)   #read the first 10 characters
print("Read string is: ", str)

pos = infile.tell()    #get the current position
print("Current position is: ", pos)

pos = infile.seek(0, 0)   #reposition the pointer at the beginning of the file
```

```
str = infile.read(30)

print("Read string is: ", str)

infile.close()
```