NAME: - SANYAM JAIN

*Assignment 1*

1. List Methods & Functions

List of list methods and functions.

List Methods

|  |  |  |
| --- | --- | --- |
| Method | Description | Examples |
|  |  |  |
| append(x) | Adds an item (*x*) to the end of the list. This is equivalent to a[len(a):] = [x]. | a = ["bee", "moth"]  print(a)  a.append("ant")  print(a)  RLT  ['bee', 'moth']  ['bee', 'moth', 'ant'] |
| extend(*iterable*) | Extends the list by appending all the items from the iterable. This allows you to join two lists together. This method is equivalent to a[len(a):] = iterable. | a = ["bee", "moth"]  print(a)  a.extend(["ant", "fly"])  print(a)  RESULT  ['bee', 'moth']  ['bee', 'moth', 'ant', 'fly'] |
| insert(*i*, *x*) | Inserts an item at a given position. The first argument is the index of the element before which to insert. For example, a.insert(0, x) inserts at the front of the list. | a = ["bee", "moth"]  print(a)  a.insert(0, "ant")  print(a)  a.insert(2, "fly")  print(a)  RESULT  ['bee', 'moth']  ['ant', 'bee', 'moth']  ['ant', 'bee', 'fly', 'moth'] |
| remove(*x*) | Removes the first item from the list that has a value of *x*. Returns an error if there is no such item. | a = ["bee", "moth", "ant"]  print(a)  a.remove("moth")  print(a)  RESULT  ['bee', 'moth', 'ant']  ['bee', 'ant'] |
| pop([*i*]) | Removes the item at the given position in the list, and returns it. If no index is specified, pop() removes and returns the last item in the list. | a = ["bee", "moth", "ant"]  print(a)  a.pop()  print(a)  ​  # Example 2: Index specified  a = ["bee", "moth", "ant"]  print(a)  a.pop(1)  print(a)  RESULT  ['bee', 'moth', 'ant']  ['bee', 'moth']  ['bee', 'moth', 'ant']  ['bee', 'ant'] |
| clear() | Removes all items from the list. Equivalent to del a[:]. | a = ["bee", "moth", "ant"]  print(a)  a.clear()  print(a)  RESULT  ['bee', 'moth', 'ant']  [] |
| index(*x*[, *start*[, *end*]]) | Returns the position of the first list item that has a value of x. Raises a [ValueError](http://www.python-ds.com/python-3-built-in-exceptions" \l "ValueError) if there is no such item.  The optional arguments start and end are interpreted as in the slice notation and are used to limit the search to a particular subsequence of the list. The returned index is computed relative to the beginning of the full sequence rather than the start argument. | a = ["bee", "ant", "moth", "ant"]  print(a.index("ant"))  print(a.index("ant", 2))  RESULT  1  3 |
| count(*x*) | Returns the number of times *x* appears in the list. | a = ["bee", "ant", "moth", "ant"]  print(a.count("bee"))  print(a.count("ant"))  print(a.count(""))  RESULT  1  2  0 |
| sort(key=None, reverse=False) | Sorts the items of the list in place. The arguments can be used to customize the operation.  key  Specifies a function of one argument that is used to extract a comparison key from each list element. The default value is None (compares the elements directly).  reverse  Boolean value. If set to True, then the list elements are sorted as if each comparison were reversed. | a = [3,6,5,2,4,1]  a.sort()  print(a)  ​  a = [3,6,5,2,4,1]  a.sort(reverse=True)  print(a)  ​  a = ["bee", "wasp", "moth", "ant"]  a.sort()  print(a)  ​  a = ["bee", "wasp", "butterfly"]  a.sort(key=len)  print(a)  ​  a = ["bee", "wasp", "butterfly"]  a.sort(key=len, reverse=True)  print(a)  RESULT  [1, 2, 3, 4, 5, 6]  [6, 5, 4, 3, 2, 1]  ['ant', 'bee', 'moth', 'wasp']  ['bee', 'wasp', 'butterfly']  ['butterfly', 'wasp', 'bee'] |
| reverse() | Reverses the elements of the list in place. | a = [3,6,5,2,4,1]  a.sort()  print(a)  ​  a = [3,6,5,2,4,1]  a.sort(reverse=True)  print(a)  ​  a = ["bee", "wasp", "moth", "ant"]  a.sort()  print(a)  ​  a = ["bee", "wasp", "butterfly"]  a.sort(key=len)  print(a)  ​  a = ["bee", "wasp", "butterfly"]  a.sort(key=len, reverse=True)  print(a)  RESULT  [1, 2, 3, 4, 5, 6]  [6, 5, 4, 3, 2, 1]  ['ant', 'bee', 'moth', 'wasp']  ['bee', 'wasp', 'butterfly']  ['butterfly', 'wasp', 'bee'] |
| copy() | Returns a shallow copy of the list. Equivalent to a[:].  Use the copy() method when you need to update the copy without affecting the original list. If you don't use this method (eg, if you do something like list2 = list1), then any updates you do to list2 will also affect list1.  The example at the side demonstrates this. | # WITHOUT copy()  a = ["bee", "wasp", "moth"]  b = a  b.append("ant")  print(a)  print(b)  ​  # WITH copy()  a = ["bee", "wasp", "moth"]  b = a.copy()  b.append("ant")  print(a)  print(b)  RESULT  ['bee', 'wasp', 'moth', 'ant']  ['bee', 'wasp', 'moth', 'ant']  ['bee', 'wasp', 'moth']  ['bee', 'wasp', 'moth', 'ant'] |

List Functions

The following Python functions can be used on lists.

|  |  |  |  |
| --- | --- | --- | --- |
| Method |  | Description |  |
| len(*s*) |  | Returns the number of items in the list.  The len() function can be used on any sequence (such as a string, bytes, tuple, list, or range) or collection (such as a dictionary, set, or frozen set). | a = ["bee", "moth", "ant"]  print(len(a))  RESULT  3 |
| list([*iterable*]) |  | The list() constructor returns a mutable sequence list of elements. The *iterable* argument is optional. You can provide any sequence or collection (such as a string, list, tuple, set, dictionary, etc). If no argument is supplied, an empty list is returned.  Strictly speaking, list([*iterable*]) is actually a mutable sequence type. | print(list())  print(list([]))  print(list(["bee", "moth", "ant"]))  print(list([["bee", "moth"], ["ant"]]))  ​  a = "bee"  print(list(a))  ​  a = ("I", "am", "a", "tuple")  print(list(a))  ​  a = {"I", "am", "a", "set"}  print(list(a))  RESULT  []  []  ['bee', 'moth', 'ant']  [['bee', 'moth'], ['ant']]  ['b', 'e', 'e']  ['I', 'am', 'a', 'tuple']  ['am', 'I', 'a', 'set'] |
| max(*iterable*, \*[, *key*, *default*])  or  max(*arg1*, *arg2*, \**args*[, *key*]) |  | Returns the largest item in an iterable (eg, list) or the largest of two or more arguments.  The *key* argument specifies a one-argument ordering function like that used for sort().  The *default* argument specifies an object to return if the provided iterable is empty. If the iterable is empty and *default* is not provided, a [ValueError](http://www.python-ds.com/python-3-built-in-exceptions" \l "ValueError) is raised.  If more than one item shares the maximum value, only the first one encountered is returned. | a = ["bee", "moth", "ant"]  print(max(a))  ​  a = ["bee", "moth", "wasp"]  print(max(a))  ​  a = [1, 2, 3, 4, 5]  b = [1, 2, 3, 4]  print(max(a, b))  RESULT  moth  wasp  [1, 2, 3, 4, 5] |
| min(*iterable*, \*[, *key*, *default*])  or  min(*arg1*, *arg2*, \**args*[, *key*]) |  | Returns the smallest item in an iterable (eg, list) or the smallest of two or more arguments.  The *key* argument specifies a one-argument ordering function like that used for sort().  The *default* argument specifies an object to return if the provided iterable is empty. If the iterable is empty and *default* is not provided, a [ValueError](http://www.python-ds.com/python-3-built-in-exceptions" \l "ValueError) is raised.  If more than one item shares the minimum value, only the first one encountered is returned. | a = ["bee", "moth", "wasp"]  print(min(a))  ​  a = ["bee", "moth", "ant"]  print(min(a))  ​  a = [1, 2, 3, 4, 5]  b = [1, 2, 3, 4]  print(min(a, b))  RESULT  bee  ant  [1, 2, 3, 4] |
| range(*stop*)  or  range(*start*, *stop*[, *step*]) |  | Represents an immutable sequence of numbers and is commonly used for looping a specific number of times in [for](http://www.python-ds.com/python-for-loops) loops.  It can be used along with list() to return a list of items between a given range.  Strictly speaking, range() is actually a mutable sequence type. | print(list(range(10)))  print(list(range(1,11)))  print(list(range(51,56)))  print(list(range(1,11,2)))  RESULT  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  [51, 52, 53, 54, 55]  [1, 3, 5, 7, |

2.Dictionary methods and its functions.

Bottom of Form

# Python Dictionary

#### **In this tutorial, you'll learn everything about Python dictionaries; how they are created, accessing, adding, removing elements from them and various built-in methods.**

Python dictionary is an unordered collection of items. Each item of a dictionary has a key/value pair.

Dictionaries are optimized to retrieve values when the key is known.

## Creating Python Dictionary

Creating a dictionary is as simple as placing items inside curly braces {} separated by commas.

An item has a key and a corresponding value that is expressed as a pair (**key: value**).

While the values can be of any data type and can repeat, keys must be of immutable type ([string](https://www.programiz.com/python-programming/string), [number](https://www.programiz.com/python-programming/numbers) or [tuple](https://www.programiz.com/python-programming/tuple) with immutable elements) and must be unique.

# empty dictionary

my\_dict = {}

# dictionary with integer keys

my\_dict = {1: 'apple', 2: 'ball'}

# dictionary with mixed keys

my\_dict = {'name': 'John', 1: [2, 4, 3]}

# using dict()

my\_dict = dict({1:'apple', 2:'ball'})

# from sequence having each item as a pair

my\_dict = dict([(1,'apple'), (2,'ball')])

As you can see from above, we can also create a dictionary using the built-in dict() function.

## Accessing Elements from Dictionary

While indexing is used with other data types to access values, a dictionary uses keys. Keys can be used either inside square brackets [] or with the get() method.

If we use the square brackets [], KeyError is raised in case a key is not found in the dictionary. On the other hand, the get() method returns None if the key is not found.

# get vs [] for retrieving elements

my\_dict = {'name': 'Jack', 'age': 26}

# Output: Jack

print(my\_dict['name'])

# Output: 26

print(my\_dict.get('age'))

# Trying to access keys which doesn't exist throws error

# Output None

print(my\_dict.get('address'))

# KeyError

print(my\_dict['address'])

**Output**

Jack

26

None

Traceback (most recent call last):

File "<string>", line 15, in <module>

print(my\_dict['address'])

KeyError: 'address'

## Changing and Adding Dictionary elements

Dictionaries are mutable. We can add new items or change the value of existing items using an assignment operator.

If the key is already present, then the existing value gets updated. In case the key is not present, a new (**key: value**) pair is added to the dictionary.

# Changing and adding Dictionary Elements

my\_dict = {'name': 'Jack', 'age': 26}

# update value

my\_dict['age'] = 27

#Output: {'age': 27, 'name': 'Jack'}

print(my\_dict)

# add item

my\_dict['address'] = 'Downtown'

# Output: {'address': 'Downtown', 'age': 27, 'name': 'Jack'}

print(my\_dict)

**Output**

{'name': 'Jack', 'age': 27}

{'name': 'Jack', 'age': 27, 'address': 'Downtown'}

## Removing elements from Dictionary

We can remove a particular item in a dictionary by using the pop() method. This method removes an item with the provided key and returns the value.

The popitem() method can be used to remove and return an arbitrary (key, value) item pair from the dictionary. All the items can be removed at once, using the clear() method.

We can also use the del keyword to remove individual items or the entire dictionary itself.

# Removing elements from a dictionary

# create a dictionary

squares = {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

# remove a particular item, returns its value

# Output: 16

print(squares.pop(4))

# Output: {1: 1, 2: 4, 3: 9, 5: 25}

print(squares)

# remove an arbitrary item, return (key,value)

# Output: (5, 25)

print(squares.popitem())

# Output: {1: 1, 2: 4, 3: 9}

print(squares)

# remove all items

squares.clear()

# Output: {}

print(squares)

# delete the dictionary itself

del squares

# Throws Error

print(squares)

**Output**

16

{1: 1, 2: 4, 3: 9, 5: 25}

(5, 25)

{1: 1, 2: 4, 3: 9}

{}

Traceback (most recent call last):

File "<string>", line 30, in <module>

print(squares)

NameError: name 'squares' is not defined

## Python Dictionary Methods

Methods that are available with a dictionary are tabulated below. Some of them have already been used in the above examples.

|  |  |
| --- | --- |
| Method | Description |
| [clear()](https://www.programiz.com/python-programming/methods/dictionary/clear) | Removes all items from the dictionary. |
| [copy()](https://www.programiz.com/python-programming/methods/dictionary/copy) | Returns a shallow copy of the dictionary. |
| [fromkeys(seq[, v])](https://www.programiz.com/python-programming/methods/dictionary/fromkeys) | Returns a new dictionary with keys from seq and value equal to v (defaults to None). |
| [get(key[,d])](https://www.programiz.com/python-programming/methods/dictionary/get) | Returns the value of the key. If the key does not exist, returns d (defaults to None). |
| [items()](https://www.programiz.com/python-programming/methods/dictionary/items) | Return a new object of the dictionary's items in (key, value) format. |
| [keys()](https://www.programiz.com/python-programming/methods/dictionary/keys) | Returns a new object of the dictionary's keys. |
| [pop(key[,d])](https://www.programiz.com/python-programming/methods/dictionary/pop) | Removes the item with the key and returns its value or d if key is not found. If d is not provided and the key is not found, it raises KeyError. |
| [popitem()](https://www.programiz.com/python-programming/methods/dictionary/popitem) | Removes and returns an arbitrary item (**key, value**). Raises KeyError if the dictionary is empty. |
| [setdefault(key[,d])](https://www.programiz.com/python-programming/methods/dictionary/setdefault) | Returns the corresponding value if the key is in the dictionary. If not, inserts the key with a value of d and returns d (defaults to None). |
| [update([other])](https://www.programiz.com/python-programming/methods/dictionary/update) | Updates the dictionary with the key/value pairs from other, overwriting existing keys. |
| [values()](https://www.programiz.com/python-programming/methods/dictionary/values) | Returns a new object of the dictionary's values |

Here are a few example use cases of these methods.

# Dictionary Methods

marks = {}.fromkeys(['Math', 'English', 'Science'], 0)

# Output: {'English': 0, 'Math': 0, 'Science': 0}

print(marks)

for item in marks.items():

print(item)

# Output: ['English', 'Math', 'Science']

print(list(sorted(marks.keys())))

**Output**

{'Math': 0, 'English': 0, 'Science': 0}

('Math', 0)

('English', 0)

('Science', 0)

['English', 'Math', 'Science']

## Python Dictionary Comprehension

Dictionary comprehension is an elegant and concise way to create a new dictionary from an iterable in Python.

Dictionary comprehension consists of an expression pair (**key: value**) followed by a for statement inside curly braces {}.

Here is an example to make a dictionary with each item being a pair of a number and its square.

# Dictionary Comprehension

squares = {x: x\*x for x in range(6)}

print(squares)

**Output**

{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

This code is equivalent to

squares = {}

for x in range(6):

squares[x] = x\*x

print(squares)

**Output**

{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

A dictionary comprehension can optionally contain more [for](https://www.programiz.com/python-programming/for-loop) or [if](https://www.programiz.com/python-programming/if-elif-else) statements.

An optional if statement can filter out items to form the new dictionary.

Here are some examples to make a dictionary with only odd items.

# Dictionary Comprehension with if conditional

odd\_squares = {x: x\*x for x in range(11) if x % 2 == 1}

print(odd\_squares)

**Output**

{1: 1, 3: 9, 5: 25, 7: 49, 9: 81}

To learn more dictionary comprehensions, visit [Python Dictionary Comprehension](https://www.programiz.com/python-programming/dictionary-comprehension).

## Other Dictionary Operations

### Dictionary Membership Test

We can test if a key is in a dictionary or not using the keyword in. Notice that the membership test is only for the keys and not for the values.

# Membership Test for Dictionary Keys

squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}

# Output: True

print(1 in squares)

# Output: True

print(2 not in squares)

# membership tests for key only not value

# Output: False

print(49 in squares)

**Output**

True

True

False

### Iterating Through a Dictionary

We can iterate through each key in a dictionary using a for loop.

# Iterating through a Dictionary

squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}

for i in squares:

print(squares[i])

**Output**

1

9

25

49

81

### Dictionary Built-in Functions

Built-in functions like all(), any(), len(), cmp(), sorted(), etc. are commonly used with dictionaries to perform different tasks.

|  |  |
| --- | --- |
| Function | Description |
| [all()](https://www.programiz.com/python-programming/methods/built-in/all) | Return True if all keys of the dictionary are True (or if the dictionary is empty). |
| [any()](https://www.programiz.com/python-programming/methods/built-in/any) | Return True if any key of the dictionary is true. If the dictionary is empty, return False. |
| [len()](https://www.programiz.com/python-programming/methods/built-in/len) | Return the length (the number of items) in the dictionary. |
| cmp() | Compares items of two dictionaries. (Not available in Python 3) |
| [sorted()](https://www.programiz.com/python-programming/methods/built-in/sorted) | Return a new sorted list of keys in the dictionary. |

Here are some examples that use built-in functions to work with a dictionary.

# Dictionary Built-in Functions

squares = {0: 0, 1: 1, 3: 9, 5: 25, 7: 49, 9: 81}

# Output: False

print(all(squares))

# Output: True

print(any(squares))

# Output: 6

print(len(squares))

# Output: [0, 1, 3, 5, 7, 9]

print(sorted(squares))

**Output**

False

True

6

[0, 1, 3, 5, 7, 9]

Top of Form

3. Sets and its default functions..

Python Sets

#### **In this tutorial, you'll learn everything about Python sets; how they are created, adding or removing elements from them, and all operations performed on sets in Python.**

A set is an unordered collection of items. Every set element is unique (no duplicates) and must be immutable (cannot be changed).

However, a set itself is mutable. We can add or remove items from it.

Sets can also be used to perform mathematical set operations like union, intersection, symmetric difference, etc.

## Creating Python Sets

A set is created by placing all the items (elements) inside curly braces {}, separated by comma, or by using the built-in set() function.

It can have any number of items and they may be of different types (integer, float, tuple, string etc.). But a set cannot have mutable elements like [lists](https://www.programiz.com/python-programming/list), sets or [dictionaries](https://www.programiz.com/python-programming/dictionary) as its elements.

# Different types of sets in Python

# set of integers

my\_set = {1, 2, 3}

print(my\_set)

# set of mixed datatypes

my\_set = {1.0, "Hello", (1, 2, 3)}

print(my\_set)

**Output**

{1, 2, 3}

{1.0, (1, 2, 3), 'Hello'}

Try the following examples as well.

# set cannot have duplicates

# Output: {1, 2, 3, 4}

my\_set = {1, 2, 3, 4, 3, 2}

print(my\_set)

# we can make set from a list

# Output: {1, 2, 3}

my\_set = set([1, 2, 3, 2])

print(my\_set)

# set cannot have mutable items

# here [3, 4] is a mutable list

# this will cause an error.

my\_set = {1, 2, [3, 4]}

**Output**

{1, 2, 3, 4}

{1, 2, 3}

Traceback (most recent call last):

File "<string>", line 15, in <module>

my\_set = {1, 2, [3, 4]}

TypeError: unhashable type: 'list'

Creating an empty set is a bit tricky.

Empty curly braces {} will make an empty dictionary in Python. To make a set without any elements, we use the set() function without any argument.

# Distinguish set and dictionary while creating empty set

# initialize a with {}

a = {}

# check data type of a

print(type(a))

# initialize a with set()

a = set()

# check data type of a

print(type(a))

**Output**

<class 'dict'>

<class 'set'>

## Modifying a set in Python

Sets are mutable. However, since they are unordered, indexing has no meaning.

We cannot access or change an element of a set using indexing or slicing. Set data type does not support it.

We can add a single element using the add() method, and multiple elements using the update() method. The update() method can take [tuples](https://www.programiz.com/python-programming/tuple), lists, [strings](https://www.programiz.com/python-programming/string) or other sets as its argument. In all cases, duplicates are avoided.

# initialize my\_set

my\_set = {1, 3}

print(my\_set)

# if you uncomment line 9,

# you will get an error

# TypeError: 'set' object does not support indexing

# my\_set[0]

# add an element

# Output: {1, 2, 3}

my\_set.add(2)

print(my\_set)

# add multiple elements

# Output: {1, 2, 3, 4}

my\_set.update([2, 3, 4])

print(my\_set)

# add list and set

# Output: {1, 2, 3, 4, 5, 6, 8}

my\_set.update([4, 5], {1, 6, 8})

print(my\_set)

**Output**

{1, 3}

{1, 2, 3}

{1, 2, 3, 4}

{1, 2, 3, 4, 5, 6, 8}

## Removing elements from a set

A particular item can be removed from a set using the methods discard() and remove().

The only difference between the two is that the discard() function leaves a set unchanged if the element is not present in the set. On the other hand, the remove() function will raise an error in such a condition (if element is not present in the set).

The following example will illustrate this.

# Difference between discard() and remove()

# initialize my\_set

my\_set = {1, 3, 4, 5, 6}

print(my\_set)

# discard an element

# Output: {1, 3, 5, 6}

my\_set.discard(4)

print(my\_set)

# remove an element

# Output: {1, 3, 5}

my\_set.remove(6)

print(my\_set)

# discard an element

# not present in my\_set

# Output: {1, 3, 5}

my\_set.discard(2)

print(my\_set)

# remove an element

# not present in my\_set

# you will get an error.

# Output: KeyError

my\_set.remove(2)

**Output**

{1, 3, 4, 5, 6}

{1, 3, 5, 6}

{1, 3, 5}

{1, 3, 5}

Traceback (most recent call last):

File "<string>", line 28, in <module>

KeyError: 2

Similarly, we can remove and return an item using the pop() method.

Since set is an unordered data type, there is no way of determining which item will be popped. It is completely arbitrary.

We can also remove all the items from a set using the clear() method.

# initialize my\_set

# Output: set of unique elements

my\_set = set("HelloWorld")

print(my\_set)

# pop an element

# Output: random element

print(my\_set.pop())

# pop another element

my\_set.pop()

print(my\_set)

# clear my\_set

# Output: set()

my\_set.clear()

print(my\_set)

print(my\_set)

**Output**

{'H', 'l', 'r', 'W', 'o', 'd', 'e'}

H

{'r', 'W', 'o', 'd', 'e'}

set()

## Python Set Operations

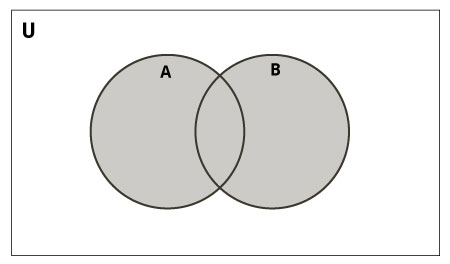
Sets can be used to carry out mathematical set operations like union, intersection, difference and symmetric difference. We can do this with operators or methods.

Let us consider the following two sets for the following operations.

>>> A = {1, 2, 3, 4, 5}

>>> B = {4, 5, 6, 7, 8}

### Set Union

Set Union in Python

Union of A and B is a set of all elements from both sets.

Union is performed using | operator. Same can be accomplished using the union() method.

# Set union method

# initialize A and B

A = {1, 2, 3, 4, 5}

B = {4, 5, 6, 7, 8}

# use | operator

# Output: {1, 2, 3, 4, 5, 6, 7, 8}

print(A | B)

**Output**

{1, 2, 3, 4, 5, 6, 7, 8}

Try the following examples on Python shell.

# use union function

>>> A.union(B)

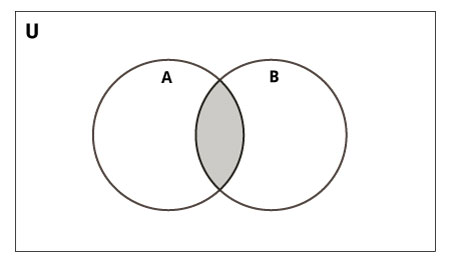
{1, 2, 3, 4, 5, 6, 7, 8}

# use union function on B

>>> B.union(A)

{1, 2, 3, 4, 5, 6, 7, 8}

### Set Intersection

Set Intersection in Python

Intersection of A and B is a set of elements that are common in both the sets.

Intersection is performed using & operator. Same can be accomplished using the intersection() method.

# Intersection of sets

# initialize A and B

A = {1, 2, 3, 4, 5}

B = {4, 5, 6, 7, 8}

# use & operator

# Output: {4, 5}

print(A & B)

**Output**

{4, 5}

Try the following examples on Python shell.

# use intersection function on A

>>> A.intersection(B)

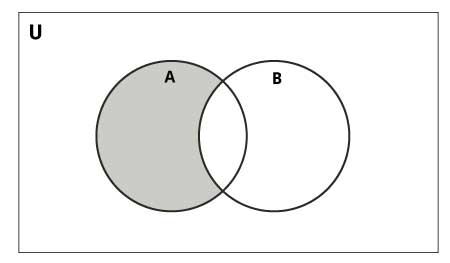
{4, 5}

# use intersection function on B

>>> B.intersection(A)

{4, 5}

### Set Difference

Set Difference in Python

Difference of the set B from set A(A - B) is a set of elements that are only in A but not in B. Similarly, B - A is a set of elements in B but not in A.

Difference is performed using - operator. Same can be accomplished using the difference() method.

# Difference of two sets

# initialize A and B

A = {1, 2, 3, 4, 5}

B = {4, 5, 6, 7, 8}

# use - operator on A

# Output: {1, 2, 3}

print(A - B)

**Output**

{1, 2, 3}

Try the following examples on Python shell.

# use difference function on A

>>> A.difference(B)

{1, 2, 3}

# use - operator on B

>>> B - A

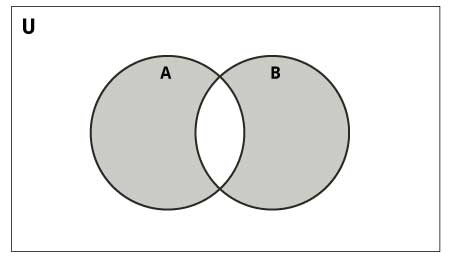
{8, 6, 7}

# use difference function on B

>>> B.difference(A)

{8, 6, 7}

### Set Symmetric Difference

Set Symmetric Difference in Python

Symmetric Difference of A and B is a set of elements in A and B but not in both (excluding the intersection).

Symmetric difference is performed using ^ operator. Same can be accomplished using the method symmetric\_difference().

# Symmetric difference of two sets

# initialize A and B

A = {1, 2, 3, 4, 5}

B = {4, 5, 6, 7, 8}

# use ^ operator

# Output: {1, 2, 3, 6, 7, 8}

print(A ^ B)

**Output**

{1, 2, 3, 6, 7, 8}

Try the following examples on Python shell.

# use symmetric\_difference function on A

>>> A.symmetric\_difference(B)

{1, 2, 3, 6, 7, 8}

# use symmetric\_difference function on B

>>> B.symmetric\_difference(A)

{1, 2, 3, 6, 7, 8}

## Other Python Set Methods

There are many set methods, some of which we have already used above. Here is a list of all the methods that are available with the set objects:

|  |  |
| --- | --- |
| Method | Description |
| [add()](https://www.programiz.com/python-programming/methods/set/add) | Adds an element to the set |
| [clear()](https://www.programiz.com/python-programming/methods/set/clear) | Removes all elements from the set |
| [copy()](https://www.programiz.com/python-programming/methods/set/copy) | Returns a copy of the set |
| [difference()](https://www.programiz.com/python-programming/methods/set/difference) | Returns the difference of two or more sets as a new set |
| [difference\_update()](https://www.programiz.com/python-programming/methods/set/difference_update) | Removes all elements of another set from this set |
| [discard()](https://www.programiz.com/python-programming/methods/set/discard) | Removes an element from the set if it is a member. (Do nothing if the element is not in set) |
| [intersection()](https://www.programiz.com/python-programming/methods/set/intersection) | Returns the intersection of two sets as a new set |
| [intersection\_update()](https://www.programiz.com/python-programming/methods/set/intersection_update) | Updates the set with the intersection of itself and another |
| [isdisjoint()](https://www.programiz.com/python-programming/methods/set/isdisjoint) | Returns True if two sets have a null intersection |
| [issubset()](https://www.programiz.com/python-programming/methods/set/issubset) | Returns True if another set contains this set |
| [issuperset()](https://www.programiz.com/python-programming/methods/set/issuperset) | Returns True if this set contains another set |
| [pop()](https://www.programiz.com/python-programming/methods/set/pop) | Removes and returns an arbitrary set element. Raises KeyError if the set is empty |
| [remove()](https://www.programiz.com/python-programming/methods/set/remove) | Removes an element from the set. If the element is not a member, raises a KeyError |
| [symmetric\_difference()](https://www.programiz.com/python-programming/methods/set/symmetric_difference) | Returns the symmetric difference of two sets as a new set |
| [symmetric\_difference\_update()](https://www.programiz.com/python-programming/methods/set/symmetric_difference_update) | Updates a set with the symmetric difference of itself and another |
| [union()](https://www.programiz.com/python-programming/methods/set/union) | Returns the union of sets in a new set |
| [update()](https://www.programiz.com/python-programming/methods/set/update) | Updates the set with the union of itself and others |

## Other Set Operations

### Set Membership Test

We can test if an item exists in a set or not, using the in keyword.

# in keyword in a set

# initialize my\_set

my\_set = set("apple")

# check if 'a' is present

# Output: True

print('a' in my\_set)

# check if 'p' is present

# Output: False

print('p' not in my\_set)

**Output**

True

False

### Iterating Through a Set

We can iterate through each item in a set using a for loop.

>>> for letter in set("apple"):

... print(letter)

...

a

p

e

l

### Built-in Functions with Set

Built-in functions like all(), any(), enumerate(), len(), max(), min(), sorted(), sum() etc. are commonly used with sets to perform different tasks.

|  |  |
| --- | --- |
| Function | Description |
| [all()](https://www.programiz.com/python-programming/methods/built-in/all) | Returns True if all elements of the set are true (or if the set is empty). |
| [any()](https://www.programiz.com/python-programming/methods/built-in/any) | Returns True if any element of the set is true. If the set is empty, returns False. |
| [enumerate()](https://www.programiz.com/python-programming/methods/built-in/enumerate) | Returns an enumerate object. It contains the index and value for all the items of the set as a pair. |
| [len()](https://www.programiz.com/python-programming/methods/built-in/len) | Returns the length (the number of items) in the set. |
| [max()](https://www.programiz.com/python-programming/methods/built-in/max) | Returns the largest item in the set. |
| [min()](https://www.programiz.com/python-programming/methods/built-in/min) | Returns the smallest item in the set. |
| [sorted()](https://www.programiz.com/python-programming/methods/built-in/sorted) | Returns a new sorted list from elements in the set(does not sort the set itself). |
| [sum()](https://www.programiz.com/python-programming/methods/built-in/sum) | Returns the sum of all elements in the set. |

## Python Frozenset

Frozenset is a new class that has the characteristics of a set, but its elements cannot be changed once assigned. While tuples are immutable lists, frozensets are immutable sets.

Sets being mutable are unhashable, so they can't be used as dictionary keys. On the other hand, frozensets are hashable and can be used as keys to a dictionary.

Frozensets can be created using the [frozenset()](https://www.programiz.com/python-programming/methods/built-in/frozenset) function.

This data type supports methods like copy(), difference(), intersection(), isdisjoint(), issubset(), issuperset(), symmetric\_difference() and union(). Being immutable, it does not have methods that add or remove elements.

# Frozensets

# initialize A and B

A = frozenset([1, 2, 3, 4])

B = frozenset([3, 4, 5, 6])

Try these examples on Python shell.

>>> A.isdisjoint(B)

False

>>> A.difference(B)

frozenset({1, 2})

>>> A | B

frozenset({1, 2, 3, 4, 5, 6})

>>> A.add(3)

...

AttributeError: 'frozenset' object has no attribute 'add'

4. Tuple and explore default functions.

Python Tuple

#### **In this article, you'll learn everything about Python tuples. More specifically, what are tuples, how to create them, when to use them and various methods you should be familiar with.**

A tuple in Python is similar to a [list](https://www.programiz.com/python-programming/list). The difference between the two is that we cannot change the elements of a tuple once it is assigned whereas we can change the elements of a list.

## Creating a Tuple

A tuple is created by placing all the items (elements) inside parentheses (), separated by commas. The parentheses are optional, however, it is a good practice to use them.

A tuple can have any number of items and they may be of different types (integer, float, list, [string](https://www.programiz.com/python-programming/string), etc.).

# Different types of tuples

# Empty tuple

my\_tuple = ()

print(my\_tuple)

# Tuple having integers

my\_tuple = (1, 2, 3)

print(my\_tuple)

# tuple with mixed datatypes

my\_tuple = (1, "Hello", 3.4)

print(my\_tuple)

# nested tuple

my\_tuple = ("mouse", [8, 4, 6], (1, 2, 3))

print(my\_tuple)

**Output**

()

(1, 2, 3)

(1, 'Hello', 3.4)

('mouse', [8, 4, 6], (1, 2, 3))

A tuple can also be created without using parentheses. This is known as tuple packing.

my\_tuple = 3, 4.6, "dog"

print(my\_tuple)

# tuple unpacking is also possible

a, b, c = my\_tuple

print(a) # 3

print(b) # 4.6

print(c) # dog

**Output**

(3, 4.6, 'dog')

3

4.6

dog

Creating a tuple with one element is a bit tricky.

Having one element within parentheses is not enough. We will need a trailing comma to indicate that it is, in fact, a tuple.

my\_tuple = ("hello")

print(type(my\_tuple)) # <class 'str'>

# Creating a tuple having one element

my\_tuple = ("hello",)

print(type(my\_tuple)) # <class 'tuple'>

# Parentheses is optional

my\_tuple = "hello",

print(type(my\_tuple)) # <class 'tuple'>

**Output**

<class 'str'>

<class 'tuple'>

<class 'tuple'>

## Access Tuple Elements

There are various ways in which we can access the elements of a tuple.

### 1. Indexing

We can use the index operator [] to access an item in a tuple, where the index starts from 0.

So, a tuple having 6 elements will have indices from 0 to 5. Trying to access an index outside of the tuple index range(6,7,... in this example) will raise an IndexError.

The index must be an integer, so we cannot use float or other types. This will result in TypeError.

Likewise, nested tuples are accessed using nested indexing, as shown in the example below.

# Accessing tuple elements using indexing

my\_tuple = ('p','e','r','m','i','t')

print(my\_tuple[0]) # 'p'

print(my\_tuple[5]) # 't'

# IndexError: list index out of range

# print(my\_tuple[6])

# Index must be an integer

# TypeError: list indices must be integers, not float

# my\_tuple[2.0]

# nested tuple

n\_tuple = ("mouse", [8, 4, 6], (1, 2, 3))

# nested index

print(n\_tuple[0][3]) # 's'

print(n\_tuple[1][1]) # 4

**Output**

p

t

s

4

### 2. Negative Indexing

Python allows negative indexing for its sequences.

The index of -1 refers to the last item, -2 to the second last item and so on.

# Negative indexing for accessing tuple elements

my\_tuple = ('p', 'e', 'r', 'm', 'i', 't')

# Output: 't'

print(my\_tuple[-1])

# Output: 'p'

print(my\_tuple[-6])

**Output**

t

p

### 3. Slicing

We can access a range of items in a tuple by using the slicing operator colon :.

# Accessing tuple elements using slicing

my\_tuple = ('p','r','o','g','r','a','m','i','z')

# elements 2nd to 4th

# Output: ('r', 'o', 'g')

print(my\_tuple[1:4])

# elements beginning to 2nd

# Output: ('p', 'r')

print(my\_tuple[:-7])

# elements 8th to end

# Output: ('i', 'z')

print(my\_tuple[7:])

# elements beginning to end

# Output: ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')

print(my\_tuple[:])

**Output**

('r', 'o', 'g')

('p', 'r')

('i', 'z')

('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')

Slicing can be best visualized by considering the index to be between the elements as shown below. So if we want to access a range, we need the index that will slice the portion from the tuple.

Element Slicing in Python

## Changing a Tuple

Unlike lists, tuples are immutable.

This means that elements of a tuple cannot be changed once they have been assigned. But, if the element is itself a mutable data type like list, its nested items can be changed.

We can also assign a tuple to different values (reassignment).

# Changing tuple values

my\_tuple = (4, 2, 3, [6, 5])

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

# my\_tuple[1] = 9

# However, item of mutable element can be changed

my\_tuple[3][0] = 9 # Output: (4, 2, 3, [9, 5])

print(my\_tuple)

# Tuples can be reassigned

my\_tuple = ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')

# Output: ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')

print(my\_tuple)

**Output**

(4, 2, 3, [9, 5])

('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')

We can use + operator to combine two tuples. This is called **concatenation**.

We can also **repeat** the elements in a tuple for a given number of times using the \* operator.

Both + and \* operations result in a new tuple.

# Concatenation

# Output: (1, 2, 3, 4, 5, 6)

print((1, 2, 3) + (4, 5, 6))

# Repeat

# Output: ('Repeat', 'Repeat', 'Repeat')

print(("Repeat",) \* 3)

**Output**

(1, 2, 3, 4, 5, 6)

('Repeat', 'Repeat', 'Repeat')

## Deleting a Tuple

As discussed above, we cannot change the elements in a tuple. It means that we cannot delete or remove items from a tuple.

Deleting a tuple entirely, however, is possible using the keyword [del](https://www.programiz.com/python-programming/keyword-list#del).

# Deleting tuples

my\_tuple = ('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')

# can't delete items

# TypeError: 'tuple' object doesn't support item deletion

# del my\_tuple[3]

# Can delete an entire tuple

del my\_tuple

# NameError: name 'my\_tuple' is not defined

print(my\_tuple)

**Output**

Traceback (most recent call last):

File "<string>", line 12, in <module>

NameError: name 'my\_tuple' is not defined

## Tuple Methods

Methods that add items or remove items are not available with tuple. Only the following two methods are available.

Some examples of Python tuple methods:

my\_tuple = ('a', 'p', 'p', 'l', 'e',)

print(my\_tuple.count('p')) # Output: 2

print(my\_tuple.index('l')) # Output: 3

**Output**

2

3

## Other Tuple Operations

### 1. Tuple Membership Test

We can test if an item exists in a tuple or not, using the keyword in.

# Membership test in tuple

my\_tuple = ('a', 'p', 'p', 'l', 'e',)

# In operation

print('a' in my\_tuple)

print('b' in my\_tuple)

# Not in operation

print('g' not in my\_tuple)

**Output**

True

False

True

### 2. Iterating Through a Tuple

We can use a for loop to iterate through each item in a tuple.

# Using a for loop to iterate through a tuple

for name in ('John', 'Kate'):

print("Hello", name)

**Output**

Hello John

Hello Kate

### Advantages of Tuple over List

Since tuples are quite similar to lists, both of them are used in similar situations. However, there are certain advantages of implementing a tuple over a list. Below listed are some of the main advantages:

* We generally use tuples for heterogeneous (different) data types and lists for homogeneous (similar) data types.
* Since tuples are immutable, iterating through a tuple is faster than with list. So there is a slight performance boost.
* Tuples that contain immutable elements can be used as a key for a dictionary. With lists, this is not possible.
* If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected.

5. String and explore default.

# Python Strings

# In this tutorial you will learn to create, format, modify and delete strings in Python. Also, you will be introduced to various string operations and functions.

## What is String in Python?

A string is a sequence of characters.

A character is simply a symbol. For example, the English language has 26 characters.

Computers do not deal with characters, they deal with numbers (binary). Even though you may see characters on your screen, internally it is stored and manipulated as a combination of 0s and 1s.

This conversion of character to a number is called encoding, and the reverse process is decoding. ASCII and Unicode are some of the popular encodings used.

In Python, a string is a sequence of Unicode characters. Unicode was introduced to include every character in all languages and bring uniformity in encoding. You can learn about Unicode from [Python Unicode](http://docs.python.org/3.3/howto/unicode.html).

## How to create a string in Python?

Strings can be created by enclosing characters inside a single quote or double-quotes. Even triple quotes can be used in Python but generally used to represent multiline strings and docstrings.

# defining strings in Python

# all of the following are equivalent

my\_string = 'Hello'

print(my\_string)

my\_string = "Hello"

print(my\_string)

my\_string = '''Hello'''

print(my\_string)

# triple quotes string can extend multiple lines

my\_string = """Hello, welcome to

the world of Python"""

print(my\_string)

When you run the program, the output will be:

Hello

Hello

Hello

Hello, welcome to

the world of Python

## How to access characters in a string?

We can access individual characters using indexing and a range of characters using slicing. Index starts from 0. Trying to access a character out of index range will raise an IndexError. The index must be an integer. We can't use floats or other types, this will result into TypeError.

Python allows negative indexing for its sequences.

The index of -1 refers to the last item, -2 to the second last item and so on. We can access a range of items in a string by using the slicing operator :(colon).

#Accessing string characters in Python

str = 'programiz'

print('str = ', str)

#first character

print('str[0] = ', str[0])

#last character

print('str[-1] = ', str[-1])

#slicing 2nd to 5th character

print('str[1:5] = ', str[1:5])

#slicing 6th to 2nd last character

print('str[5:-2] = ', str[5:-2])

When we run the above program, we get the following output:

str = programiz

str[0] = p

str[-1] = z

str[1:5] = rogr

str[5:-2] = am

If we try to access an index out of the range or use numbers other than an integer, we will get errors.

# index must be in range

>>> my\_string[15]

...

IndexError: string index out of range

# index must be an integer

>>> my\_string[1.5]

...

TypeError: string indices must be integers

Slicing can be best visualized by considering the index to be between the elements as shown below.

If we want to access a range, we need the index that will slice the portion from the string.

String Slicing in Python

## How to change or delete a string?

Strings are immutable. This means that elements of a string cannot be changed once they have been assigned. We can simply reassign different strings to the same name.

>>> my\_string = 'programiz'

>>> my\_string[5] = 'a'

...

TypeError: 'str' object does not support item assignment

>>> my\_string = 'Python'

>>> my\_string

'Python'

We cannot delete or remove characters from a string. But deleting the string entirely is possible using the del keyword.

>>> del my\_string[1]

...

TypeError: 'str' object doesn't support item deletion

>>> del my\_string

>>> my\_string

...

NameError: name 'my\_string' is not defined

## Python String Operations

There are many operations that can be performed with strings which makes it one of the most used data types in Python.

To learn more about the data types available in Python visit: [Python Data Types](https://www.programiz.com/python-programming/variables-datatypes)

### Concatenation of Two or More Strings

Joining of two or more strings into a single one is called concatenation.

The **+** operator does this in Python. Simply writing two string literals together also concatenates them.

The **\*** operator can be used to repeat the string for a given number of times.

# Python String Operations

str1 = 'Hello'

str2 ='World!'

# using +

print('str1 + str2 = ', str1 + str2)

# using \*

print('str1 \* 3 =', str1 \* 3)

When we run the above program, we get the following output:

str1 + str2 = HelloWorld!

str1 \* 3 = HelloHelloHello

Writing two string literals together also concatenates them like **+** operator.

If we want to concatenate strings in different lines, we can use parentheses.

>>> # two string literals together

>>> 'Hello ''World!'

'Hello World!'

>>> # using parentheses

>>> s = ('Hello '

... 'World')

>>> s

'Hello World'

### Iterating Through a string

We can iterate through a string using a [for loop](https://www.programiz.com/python-programming/for-loop). Here is an example to count the number of 'l's in a string.

# Iterating through a string

count = 0

for letter in 'Hello World':

if(letter == 'l'):

count += 1

print(count,'letters found')

When we run the above program, we get the following output:

3 letters found

### String Membership Test

We can test if a substring exists within a string or not, using the keyword in.

>>> 'a' in 'program'

True

>>> 'at' not in 'battle'

False

### Built-in functions to Work with Python

Various built-in functions that work with sequence work with strings as well.

Some of the commonly used ones are enumerate() and len(). The enumerate() function returns an enumerate object. It contains the index and value of all the items in the string as pairs. This can be useful for iteration.

Similarly, len() returns the length (number of characters) of the string.

str = 'cold'

# enumerate()

list\_enumerate = list(enumerate(str))

print('list(enumerate(str) = ', list\_enumerate)

#character count

print('len(str) = ', len(str))

When we run the above program, we get the following output:

list(enumerate(str) = [(0, 'c'), (1, 'o'), (2, 'l'), (3, 'd')]

len(str) = 4

## Python String Formatting

### Escape Sequence

If we want to print a text like He said, "What's there?", we can neither use single quotes nor double quotes. This will result in a SyntaxError as the text itself contains both single and double quotes.

>>> print("He said, "What's there?"")

...

SyntaxError: invalid syntax

>>> print('He said, "What's there?"')

...

SyntaxError: invalid syntax

One way to get around this problem is to use triple quotes. Alternatively, we can use escape sequences.

An escape sequence starts with a backslash and is interpreted differently. If we use a single quote to represent a string, all the single quotes inside the string must be escaped. Similar is the case with double quotes. Here is how it can be done to represent the above text.

# using triple quotes

print('''He said, "What's there?"''')

# escaping single quotes

print('He said, "What\'s there?"')

# escaping double quotes

print("He said, \"What's there?\"")

When we run the above program, we get the following output:

He said, "What's there?"

He said, "What's there?"

He said, "What's there?"

Here is a list of all the escape sequences supported by Python.

|  |  |
| --- | --- |
| Escape Sequence | Description |
| \newline | Backslash and newline ignored |
| \\ | Backslash |
| \' | Single quote |
| \" | Double quote |
| \a | ASCII Bell |
| \b | ASCII Backspace |
| \f | ASCII Formfeed |
| \n | ASCII Linefeed |
| \r | ASCII Carriage Return |
| \t | ASCII Horizontal Tab |
| \v | ASCII Vertical Tab |
| \ooo | Character with octal value ooo |
| \xHH | Character with hexadecimal value HH |

Here are some examples

>>> print("C:\\Python32\\Lib")

C:\Python32\Lib

>>> print("This is printed\nin two lines")

This is printed

in two lines

>>> print("This is \x48\x45\x58 representation")

This is HEX representation

### Raw String to ignore escape sequence

Sometimes we may wish to ignore the escape sequences inside a string. To do this we can place r or R in front of the string. This will imply that it is a raw string and any escape sequence inside it will be ignored.

>>> print("This is \x61 \ngood example")

This is a

good example

>>> print(r"This is \x61 \ngood example")

This is \x61 \ngood example

### The format() Method for Formatting Strings

The format() method that is available with the string object is very versatile and powerful in formatting strings. Format strings contain curly braces {} as placeholders or replacement fields which get replaced.

We can use positional arguments or keyword arguments to specify the order.

# Python string format() method

# default(implicit) order

default\_order = "{}, {} and {}".format('John','Bill','Sean')

print('\n--- Default Order ---')

print(default\_order)

# order using positional argument

positional\_order = "{1}, {0} and {2}".format('John','Bill','Sean')

print('\n--- Positional Order ---')

print(positional\_order)

# order using keyword argument

keyword\_order = "{s}, {b} and {j}".format(j='John',b='Bill',s='Sean')

print('\n--- Keyword Order ---')

print(keyword\_order)

When we run the above program, we get the following output:

--- Default Order ---

John, Bill and Sean

--- Positional Order ---

Bill, John and Sean

--- Keyword Order ---

Sean, Bill and John

The format() method can have optional format specifications. They are separated from the field name using colon. For example, we can left-justify <, right-justify > or center ^ a string in the given space.

We can also format integers as binary, hexadecimal, etc. and floats can be rounded or displayed in the exponent format. There are tons of formatting you can use. Visit here for all the [string formatting available with the](https://www.programiz.com/python-programming/methods/string/format) [format()](https://www.programiz.com/python-programming/methods/string/format) method.

>>> # formatting integers

>>> "Binary representation of {0} is {0:b}".format(12)

'Binary representation of 12 is 1100'

>>> # formatting floats

>>> "Exponent representation: {0:e}".format(1566.345)

'Exponent representation: 1.566345e+03'

>>> # round off

>>> "One third is: {0:.3f}".format(1/3)

'One third is: 0.333'

>>> # string alignment

>>> "|{:<10}|{:^10}|{:>10}|".format('butter','bread','ham')

'|butter | bread | ham|'

### Old style formatting

We can even format strings like the old sprintf() style used in C programming language. We use the % operator to accomplish this.

>>> x = 12.3456789

>>> print('The value of x is %3.2f' %x)

The value of x is 12.35

>>> print('The value of x is %3.4f' %x)

The value of x is 12.3457

## Common Python String Methods

There are numerous methods available with the string object. The format() method that we mentioned above is one of them. Some of the commonly used methods are lower(), upper(), join(), split(), find(), replace() etc. Here is a complete list of all the [built-in methods to work with strings in Python](https://www.programiz.com/python-programming/methods/string).

>>> "PrOgRaMiZ".lower()

'programiz'

>>> "PrOgRaMiZ".upper()

'PROGRAMIZ'

>>> "This will split all words into a list".split()

['This', 'will', 'split', 'all', 'words', 'into', 'a', 'list']

>>> ' '.join(['This', 'will', 'join', 'all', 'words', 'into', 'a', 'string'])

'This will join all words into a string'

>>> 'Happy New Year'.find('ew')

7

>>> 'Happy New Year'.replace('Happy','Brilliant')

'Brilliant New Year'