Collection Datatypes

More on Strings

String Manipulation

In the second lesson "*Basic Data Types*" in Python, we learned how to define strings:

"**Objects that contain sequences of characters** "

Most applications would need to manipulate strings; at least to some extent.

Python provides a rich set of *operators, functions, and methods* for working with strings and manipulating them.

Operators used with Strings

The “+” operator; concatenates strings, It returns a string consisting of the operands joined together, as shown here:

```
>>> s = 'foo'
>>> t = 'bar'
>>> u = 'baz'

>>> s + t
'foobar'
>>> s + t + u
'foobarbaz'

>>> print('Go team' + '!!!')
Go team!!!
```

```
s = "foo"
t = "bar"
u = "baz"
s+t+u

print('Go Team ' + '!!!')
```

Operators used with Strings

The * operator creates multiple copies of a string.

If s is a string and n is an integer; either of the following expressions will return a string consisting of n concatenated copies of s:

s * n
n * s
Here are examples of both forms:

```
s = "foo."

s * 4

4 * s
```

Python

```
>>> s = 'foo.'

>>> s * 4
'foo.foo.foo.foo.'
>>> 4 * s
'foo.foo.foo.foo.'
```

String Operators

The multiplier operand, n must be an integer. You'd think it would be required to be a positive integer;

but amusingly, it can be zero or negative, in which case, the result will be an empty string:

```
s = "foo "
print(s *-8)
#-> ''
```

```
>>> 'foo' * -8
''
```

If you were to create a string variable and initialize it to the empty string by assigning it the value *'foo' * -8*, anyone would rightly think you were a bit daft; But it would work.

Built-in String Functions

As you saw in the last lesson on "*Basic Data Types in Python*".

Python provides many functions that are built-in to the interpreter and always available.

Here are a few that work with strings:

Function	Description
`chr()`	Converts an integer to a character
`ord()`	Converts a character to an integer
`len()`	Returns the length of a string
`str()`	Returns a string representation of an object

Built-in String Functions

len(s)

Returns the length of a string.

len(s) returns the number of characters in s:

```
s = "I am a string."
len(s)
#-> 14
```

str(obj)

Returns a string representation of an object.

Virtually any object in Python can be rendered as a string.

str(obj) returns the string representation of object obj

```
str(49.2)
#->'49.2'

str(3+4j)
#->'3+3j'

str(3+29)
#->'32'

str('foo')
#->'foo'
```

Built-in String Functions

ord(ch)

The ord() function returns the number representing the Unicode code of a specified character.

```
ord("a")
#->97

chr(97)
#->'a'
```

chr(i)

The chr() method returns a character (a string) from an integer (represents Unicode code point of the character).

chr() method takes a single parameter, an integer.

String Indexing

Often in programming languages, an individual item in an ordered *set* of data can be accessed directly using a numeric index or key value. This process is referred to as indexing.

In Python, strings are ordered sequences of character data, and thus can be indexed in this way.

Individual characters in a string can be accessed by specifying the string name followed by a number in square brackets ([]).

String Indexing

String indexing in Python is zero-based: The first character in the string has index 0, the next has index 1, and so on.

The index of the last character will be the length of the string minus one.

For example; a schematic diagram of the indices of the string '*foobar*' would look like this:

f	o	o	b	a	r
0	1	2	3	4	5

```
s = "foobar"
s[0]
#->'f'

s[1]
#->'o'

s[3]
#->'o'
```

String Indexing

String indices can also be specified with negative numbers, in which case indexing occurs from the end of the string backward:

-1 refers to the last character,

-2 the second-to-last character

and so on ...

Here is the same diagram showing both the positive and negative indices in the string 'foobar'

-6	-5	-4	-3	-2	-1
f	o	o	b	a	r
0	1	2	3	4	5

```
s = "foobar"
s[-1]
#->'r'

s[1]
#-> 'a'
```

String Slicing

Python also allows a form of indexing syntax that extracts **subStrings** from a String, this is known as *string slicing*.

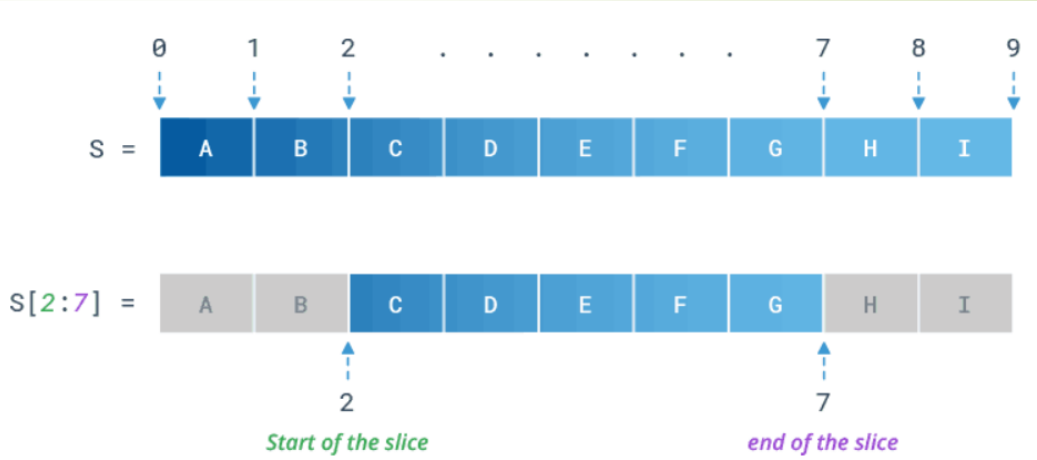
If **s** is a string, an expression of the form **s[m:n]** returns the part of **s** starting with position **m**, and up to but **not** including position **n**.

This means characters from **m** to **n-1** indexes are returned.

Example

Here is a basic example of string slicing.

```
s="ABCDEFGHI"
s[2:7]
#->'CDEFG'
```

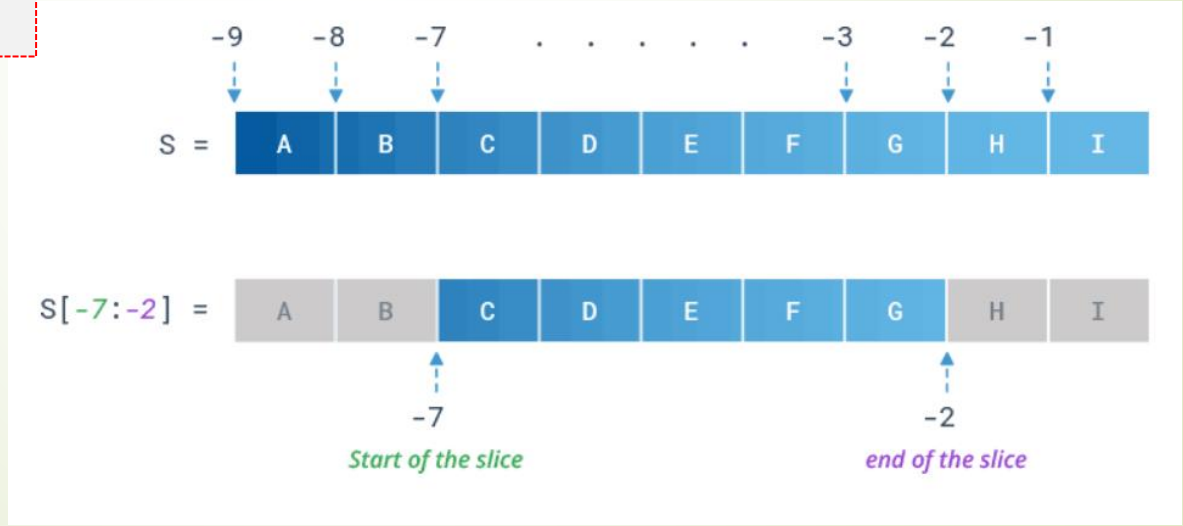


Note that the item at index 7 `'H'` is not included.

Slice with Negative Indices

You can also specify negative indices while slicing a string.

```
s="ABCDEFGHI"
s[-7:-2]
#->'CDEFG'
```



Slice with Positive & Negative Indices

You can specify both positive and negative indices at the same time.

```
s="ABCDEFGHI"
s[2:-5]
#->'CD'
```

String Slicing

If you omit the first index, the slice starts at the beginning of the string.

Thus, s[:m] and s[0:m] are equivalent:

```
s = "foobar!"

s[:3]
#->'foob'

s[0:3]
#->'foob'
```

String Slicing

Similarly, if you omit the second index as in s[n:], the slice extends from the first index through the end of the string.

This is a nice, concise alternative to the more cumbersome s[n:len(s)]:

```
s = "foobar!"

s[2:]
#->'obar'

s[2:len(s)]
#->'obar'
```

String Slicing

Omitting both indices returns the original string, in its entirety.

It’s not a copy, it’s a reference to the original string.

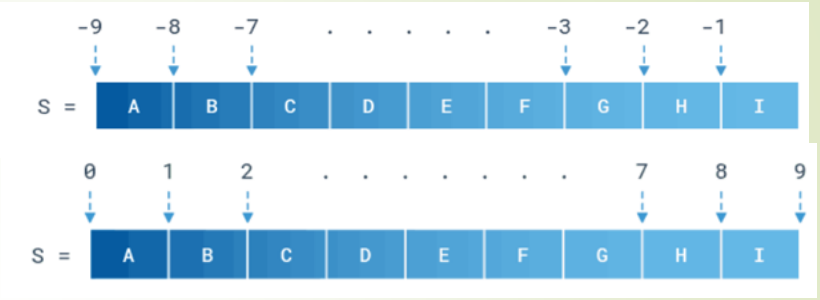
```
s = "foobar"
s[:4] + s[4:]
#->'foobar'

s[:4] + s[4:]== s
#->True
```

```
s = "foobar"
t=s[:]

id(s)
#-> 49186304

id(t)
#-> 49186304

s is t
True
```

String Slicing

If the first index in a slice is greater than or equal to the second index, Python returns an empty string.

This is yet another way to generate an empty string.




```
>>> s='ABCDEFGHI'
>>> s[2:7]
'CDEFG'
>>> s[7:2]
''
>>> s[2:2]
''
>>> s[-7:-2]
'CDEFG'
>>> s[-2:-7]
''
>>> s[-2:-2]
''
```

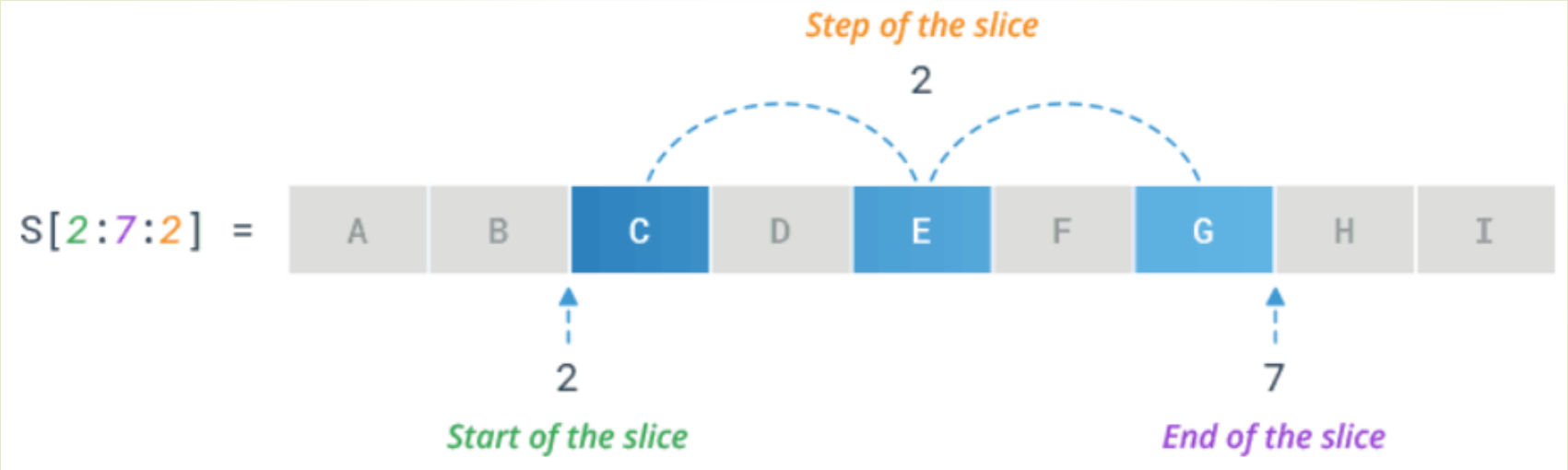
Specifying a Stride in a String Slice

There is one more variant of the slicing syntax to discuss.

A third index outlines a stride (also called a step), which indicates how many characters to jump after retrieving each character in the slice. You can specify the step of the slicing using step parameter.

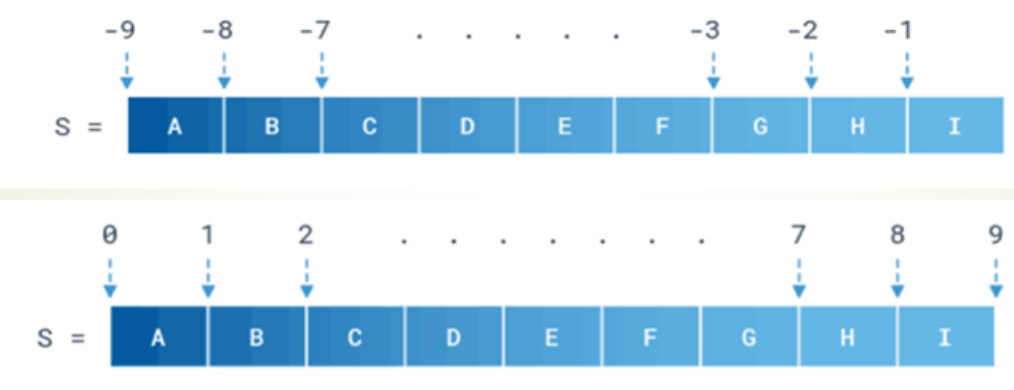
The step parameter is optional and by default 1.

```
#Retuns every 2nd item between position 2 to 7

s="ABCDEFGHI"
s[2:7:2]
#->'CEG'
```



Negative Step Size

You can even specify a negative step size.




```
#Retuns every 2nd item between position 6 to 1 in reverse order

s="ABCDEFGHI"
s[6:1:-2]
#->'CEG'
```

Specifying a Stride in a String Slice

As with any slicing, the first and second parameters can be omitted, and default to the first and last characters respectively:

```
s ="12345"*5
s
#->'1234512345123451234512345'

s[::5]
#->'11111'

s[4::5]
#->'55555'
```

Behaviour of Slice Operator

```
# Forward direction
#(left to right)

s="ABCDEFGHIJ"
s[0:5:2]

#->'ACE'
```

```
# Backward direction
#(right to left)

s="ABCDEFGHIJ"
s[5:0:-2]

#->'FDB'
```

```
S[start:stop:step]
```

S =	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
	"a	b	c	d	e	f	g	h	i	j"
	0	1	2	3	4	5	6	7	8	9

Collection Datatypes

What Are Collections In Python?

Collections in python are basically container data types, namely

Lists, Sets, Tuples & Dictionary

They have different characteristics based on the declaration and the usage.

A **list** is declared in square brackets [] , it is **mutable**, stores duplicate values and elements can be accessed using indexes.

A **tuple** is declared in round brackets () , it is **immutable**, stores duplicate values and elements can be accessed using indexes.

A **set** is declared in curly brackets {} , it is **immutable**, and does not have duplicate entries, & can't be accessed using indexes.

What Are Collections In Python?

A **dictionary** is a collection which is ordered*, changeable and does not allow duplicates.. Dictionaries are written with curly brackets {} and have keys and values.

*As of Python version 3.7, dictionaries are ordered. In Python 3.6 and earlier, dictionaries are unordered.

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}

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

Lists

What are lists?

Lists represent a group of values as a single entity where

1) Insertion Order is preserved
2) Unlike Objects are allowed
3) Duplicates are allowed
4) Lists are **Mutable**
5) Dynamic in nature
6) Values should be enclosed within square brackets.

How to create a list?

A list is created by placing all the items (elements) inside square brackets [], separated by commas.

It can have any number of items and they may be of different types (integer, float, string etc.).

```
#empty list
my_list = []

# list of intergers
my_list= [1,2,3]

#list with mixed data types
my_list = [1, "hello", 3.4]

#A list can also hava another list as an item
#This is called an nested list.

#nested list
my_list = ["moues"m [8,4,6], ['a'] ]
```

How to access elements from a list?

List Index

We can use the index operator [] to access an item in a list. In Python, indices start at 0. So, a list having 5 elements will have an index from 0 to 4.

Trying to access indexes other than these will raise an *IndexError*. The index must be an integer.

We can't use float or other types; this will result in TypeError.

Nested lists are accessed using nested indexing “[][]”.

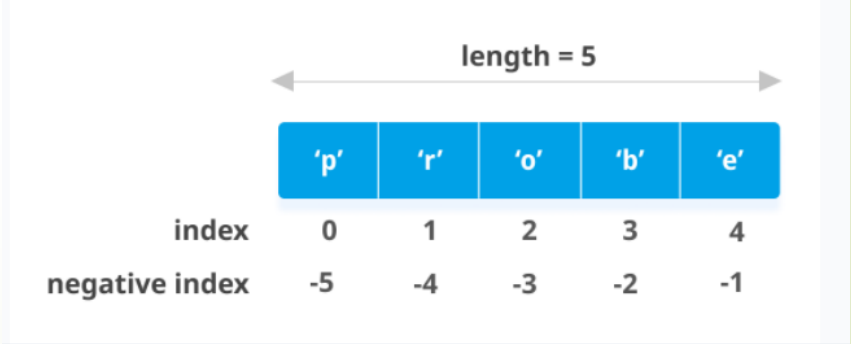
```
# List indexing

my_list = ['p','r','o','b','e']

my_list[0]
#->'p'

my_list[2]
#->'o'

my_list[4]
#->'e'

my_list[4.0]
#-> TypeError: list indices must be integers
or slices, not float
```

```
#Accessing Nestes Lists

list= ["Happy",[2,0,1,5]]

list[0]
#->"Happy"

list[0][3]
#->"p"

list[1]
#->[2,0,1,5]

list[1][2]
#->[1]


list[3]
#->IndexError: list index out of range
```

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 in lists

my_list = ['p','r','o','b','e']

my_list[-1]
#->'e'

my_list[-5]
#->'p'
```



List indexing in Python

How to slice lists in Python?

We can access a range of items

in a list by using the slicing operator “ : ” (colon).

```
# List slicing Python

my_list = ['p','r','o','g','r','a','m','i','z’]

#element 3rd to 5th
my_list[2:5]
#->  ['o', 'g', 'r’]

#element beginning to 4th
my_list[:-5]
#->['p', 'r', 'o', 'g']

#element 6th  to end
my_list[5:]
#->['a', 'm', 'i', 'z']

#element beginning to end
my_list[:]
#->['p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z']
```

Lists are Mutable

Lists are mutable, meaning their elements can be changed unlike string or tuple.

```
#Defining a list
z= [3,7,4,2]


#Update the item at index 1 with the string
"fish"
z[1] = "fish"
z
```

Lists are Mutable

We can also use the assignment operator (=) to assign to a range of items.

```
#Correcting mistake values in a list
odd= [2,4,6,8]


#change the 1st item
odd[0]=1


odd
#-> [1, 4, 6, 8]

#change 2nd to 4th items
odd[1:4] = [3,5,7]

odd
#-> [1, 3, 5, 7]
```

Using the arithmetic operators on List

We can also use + operator to combine two lists. This is also called concatenation.

The * operator repeats a list for the given number of times.

Concatenation Operator (+):

We can use + to concatenate 2 lists into a single list

```
a = [10, 20, 30]
b = [40, 50, 60]
c = a+b
c
#-> [10, 20, 30, 40, 50, 60]
```

Note: To use + operator compulsory both arguments should be list objects, otherwise we will get TypeError.

```
a = [10, 20, 30]

c = a+40
#-> TypeError: can only concatenate list (not "int") to list
```

Concatenation operator

Concatenation can happen only with lists and not with any other datatype.

```
>>> l1 = [1,2,3]
>>> l1 = l1+l1
>>> l1
[1, 2, 3, 1, 2, 3]
>>> l1 = l1 + 3
Traceback (most recent call last):
  File "<pyshell#15>", line 1, in <module>
    l1 = l1 + 3
TypeError: can only concatenate list (not "int") to list
>>> l1 = l1 +[4,5]
>>> l1
[1, 2, 3, 1, 2, 3, 4, 5]
```

Repetition Operator (*):

We can use repetition operator * to repeat elements of list specified number of times.

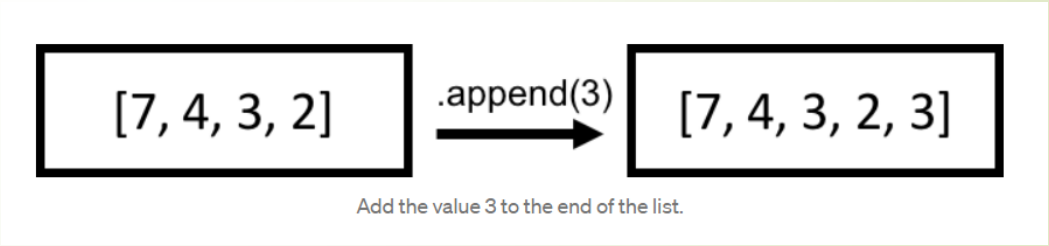
```
x = [10, 20, 30]
y = x*3
y
#-> [10, 20, 30, 10, 20, 30, 10, 20, 30]
```

Manipulating elements of a list

1) append() :

We can use the append() function to add item at the end of the list.

```
z = [7,4,3,2]
z.append(3)
z

#-> [7,4,3,2,3]
```



Appending more elements to a list

Append can be used to append only one element to a list

```
z = [7,4,3,2]
z.append(5,6)
#-> TypeError: append() takes exactly one argument (2 given)

z.append("Hello")
z
#-> [7, 4, 3, 2, 'Hello']


z.append([5,6])
z
#-> [7, 4, 3, 2, 'Hello', [5, 6]]
```

Manipulating elements of a list

2.) insert()

```
list.insert(i, elem)
```

Here, `elem` is inserted to the list at the `i^th` index. All the elements after `elem` are shifted to the right.

```
z = [7,4,3,2]

z.insert(4,1)
z
#->[7, 4, 3, 2, 1]

z.insert(2,5.6)
z
#-> [7, 4, 5.6, 3, 2, 1]
```

```
z = [7,4,3,2]
z.insert(4,[1,2])
z
#->[7, 4, 5.6, 3, [1, 2], 2, 1]

z.insert(2,"Happy")
z
#-> [7, 4, 5.6, 3, [1, 2], 2, 1]

z.append(2,"s",40)
#-> TypeError: append() takes exactly one
argument (3 given)
```

Manipulating elements of a list

Note: If the specified index is greater than max index then element will be inserted at last position. If the specified index is smaller than min index, then element will be inserted at first position.

```
n=[1,2,3,4,5]
n.insert(10,777)
n.insert(-10,999)
n
#->[999, 1, 2, 3, 4, 5, 777]
```

Manipulating elements of a list

3.extend() :

```
list1.extend(iterable)
```

Here, all the elements of `iterable` are added to the end of `list1`.

The iterable can be a list, tuple, string etc.,

```
z=[7,3,3]
z.extend([4,5])
z
#->[7, 3, 3, 4, 5]

z.extend("Happy")
z
#->[7, 3, 3, 4, 5, 'H', 'a', 'p', 'p', 'y']
```

extend()

Arguments to extend should be iterable;

It takes only one argument

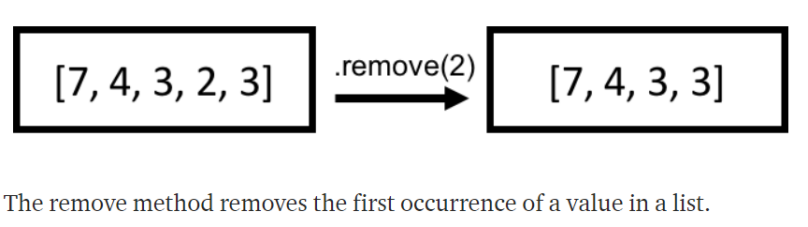
```
>>> l =[1,2,3]
>>> l.extend(3,4)
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    l.extend(3,4)
TypeError: list.extend() takes exactly one argument (2 given)
>>> l.extend(3)
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    l.extend(3)
TypeError: 'int' object is not iterable
>>> l.extend([3,4],"Happy")
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    l.extend([3,4],"Happy")
TypeError: list.extend() takes exactly one argument (2 given)
>>>
```

Manipulating elements of a list

4.remove():

We can use this function to remove specified item from the list.

If the item is present multiple times, then only first occurrence will be removed.



The remove method removes the first occurrence of a value in a list.

```
z=[7,4,3,2,3]
z
#-> [7, 4, 3, 2, 3]

z.remove(2)
z
#-> [7, 4, 3, 3]
```

Manipulating elements of a list

- If the specified item not present in list, then we will get ValueError.

Output:

```
n=[10,20,10,30]
n.remove(40)
n
#-> ValueError: list.remove(x): x not in list
```

`ValueError: list.remove(x): x not in list`

Note: Hence before using remove() method first we have to check if the specified element present **in** the list or not, by using **in** operator.

Manipulating elements of a list

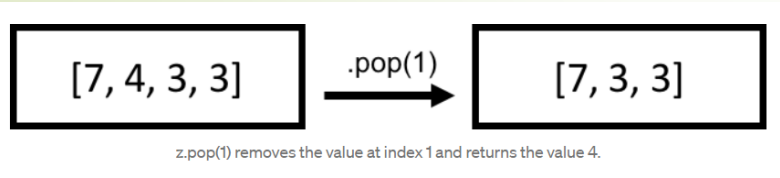
5.pop()

The pop method removes an item at the index you provide.

This method will also return the item you removed from the list.

If you don't provide an index, it will by default remove the item at the last index.

```
z= [7,4,3,3]
z
#->[7, 3, 3]

z.pop(1)
#->4

z
#->[7, 3, 3]

z.pop()
#->3

z
#->[7, 3]
```



z.pop(1) removes the value at index 1 and returns the value 4.

Manipulating elements of a list

Popping from empty list

```
>>> z = [7,4,5]
>>> print(z.pop())
5
>>> z
[7, 4]
>>> print(z.pop())
4
>>> z
[7]
>>> print(z.pop())
7
>>> z
[]
>>> print(z.pop())
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    print(z.pop())
IndexError: pop from empty list
>>>
```

How to delete or remove elements from a list?

We can delete one or more items from a list using the keyword **del**.

It can even delete the list entirely.

```
#Deleting list items
my_list = ['p','r','o','b','l','e','m']
#->['p', 'r', 'o', 'b', 'l', 'e', 'm’]

#delete multiple items
del my_list [1:5]

my_list
#->['p', 'e', 'm’]

# dlelte entire list
del my_list

my_list
#->Error: List not defined
```

Delete items in a list

Finally, we can also delete items in a list by assigning an empty list to a slice of elements.

```
my_list = ['p','r','o','b','l','e','m']

my_list [2:3]=[]
my_list
#->['p', 'r', 'b', 'l', 'e', 'm']


my_list [2:5]=[]
my_list
#->['p', 'r', 'm']
```

Clear items from list

clear()

The method clear() is used to remove all the items from a given Python List. This method does not delete the list but it makes the list empty.

clear() Parameters

The method clear() does not take any arguments as no data is needed to be supplied explicitly to delete all of the list items.

clear() Return Value

The clear() method only performs the action of removing all the list elements and hence it has nothing to return i.e. it returns None.

Example

clear()

```
items = ["Copy", "Pen",10, 1.5,True, ('A','B')]

items
#-> ['Copy', 'Pen', 10, 1.5, True, ('A', 'B')]

items.clear()
items
#-> []
```

Ordering Elements of List

1) reverse():

We can use to reverse() order of elements of list.

```
n=[10,20,30,40]
n
#->[10,20,30,40]


n.reverse()
n
#-> [40, 30, 20, 10]
```

Ordering Elements of List

2) sort():

In list by default insertion order is preserved.

If want to sort the elements of list according to default natural sorting order then we should go for sort() method.

For numbers - Default Natural sorting Order is Ascending Order .

For Strings - Default Natural sorting order is Alphabetical Order.

```
n = [20,5,15,10,0]

n
#-> [20,5,15,10,0]

n.sort()
n
#->[0,5,10,15,20]

s = ["Dog","Banana","Cat","Apple"]
s
#-> ["Dog","Banana","Cat","Apple"]

s.sort()
s
#-> ['Apple','Banana','Cat','Dog']
```

Ordering Elements of List

Note: To use sort() function, list should contain only homogeneous elements. Otherwise, we will get TypeError

```
>>> n = [20,10,"A","B"]
>>> n.sort()
Traceback (most recent call last):
  File "<pyshell#15>", line 1, in <module>
    n.sort()
TypeError: '<' not supported between instances of 'str' and 'int'
>>>
```

Ordering Elements of List

To Sort in Reverse of Default Natural Sorting Order:

We can sort according to reverse of default natural sorting order by using ***reverse=True*** argument.

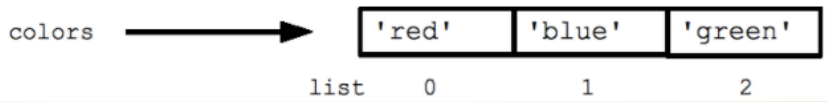
```
n = [40,10,30,20]
n
# -> [40,10,30,20]

n.sort()
n
# -> [10,20,30,40]
n.sort(reverse = True)

n
# ->[40,30,20,10]


n.sort(reverse = False)
n
# ->[10,20,30,40]
```

Aliasing a List

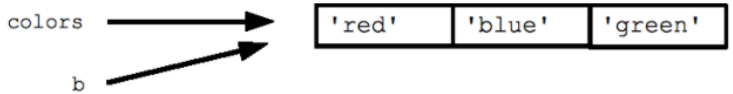
The process of giving another reference variable to the existing list is called aliasing.

Assignment with an " **=** " on lists does not make a copy. Instead, assignment makes the two variables point to the one list in memory.

```
colors = ['red', 'blue', 'green']
print colors[0]    ## red
print colors[2]    ## green
print len(colors)  ## 3
```




```
b = colors   ## Does not copy the list
```



Cloning a list

The problem in this approach is by using one reference variable if we are changing content, then those changes will be reflected to the other reference variable.

To overcome this problem, we should go for cloning.

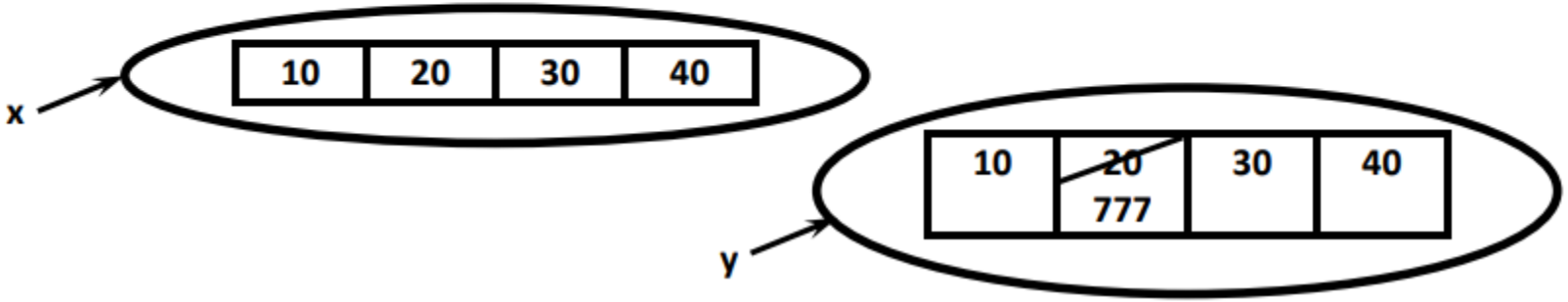
The process of creating exactly duplicate independent object is called cloning.

We can implement cloning by using ***slice*** *operator* or by using ***copy()*** function

Cloning a list

1) By using Slice Operator:

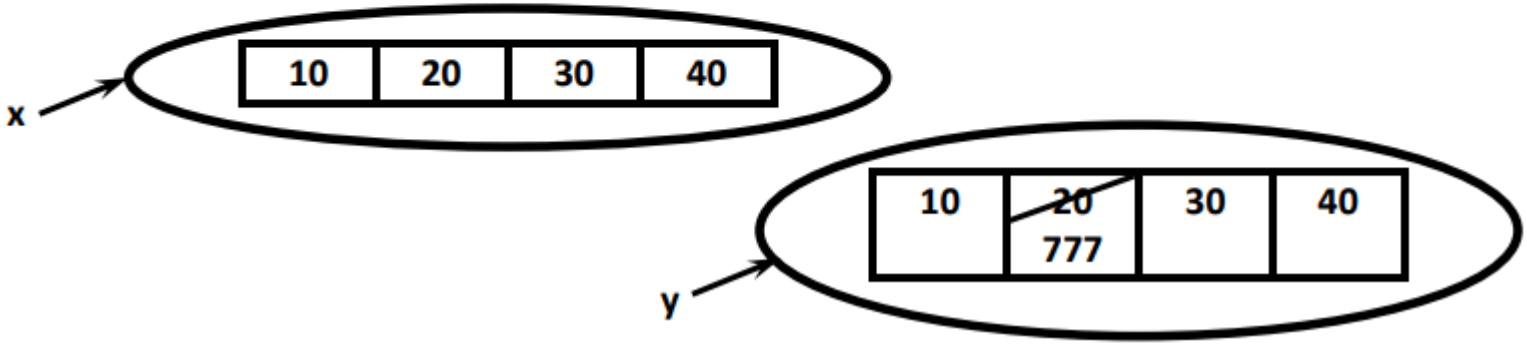
```
1) x = [10,20,30,40]
2) y = x[:]
3) y[1] = 777
4) print(x) → [10, 20, 30, 40]
5) print(y) → [10, 777, 30, 40]
```



Cloning a list

2) By using copy() Function:

```
1) x = [10,20,30,40]
2) y = x.copy()
3) y[1] = 777
4) print(x) → [10, 20, 30, 40]
5) print(y) → [10, 777, 30, 40]
```



Comparing List Objects

We can use comparison operators for List objects.

When we use comparison operators (==, !=) for List objects then the following should be considered:

1) The Number of Elements,
2) The Order of Elements,
3) The Content of Elements (Case Sensitive).

Comparing List Objects

```
x = [50, 20, 30]
y = [40, 50, 60, 100, 200]
print(x>y)  # True
print(x>=y) # True
print(x<y)  # False
print(x<=y) # False
```

```
x = ["Dog", "Cat", "Rat"]
y = ["Rat", "Cat", "Dog"]
print(x>y)  # False
print(x>=y) # False
print(x<y)  # True
print(x<=y) # True
```

```
x = ["Dog", "Cat", "Rat"]
y = ["Dog", "Cat", "Rat"]
z = ["DOG", "CAT", "RAT"]

x == y # True
x == z # False
x != z # True
```

Membership Operators on Lists

We can check whether element is a member of the list or not by using membership operators.

1) in Operator
2) not in Operator

```
n=[10,20,30,40]

print (10 in n)
#True

print (10 not in n)
#False

print (50 in n)
#False

print (50 not in n)
#True
```

Some more Functions of List:

I. To get Information about List:

1) len(): Returns the number of elements present in the list

```
n = [10, 20, 30, 40]

print(len(n))
#-> 4
```

Some more Functions of List:

2) count():

It returns the number of occurrences of specified item in the list

```
n = [1,2,2,2,2,3,3]

n.count(1)
#1

n.count(2)
#2

n.count(3)
#2

n.count(4)
#0
```

Some more functions of List

3) index():

Returns the index of first occurrence of the specified item.

```
n = [1, 2, 2, 2, 2, 3, 3]

print(n.index(1))   # 0
print(n.index(2))   # 1
print(n.index(3))   # 5
print(n.index(4))   # ValueError: 4 is not in list
```

Note: If the specified element not present in the list then we will get ValueError. Hence before index() method we have to check whether item present in the list or not by using in operator

Tuple

Tuple

1) Tuple is exactly same as List except that it is **immutable**. i.e., once a Tuple object is created, we cannot perform any changes on that object. Hence Tuple is a Read only version of List.

2) Insertion Order is preserved.

3) Duplicates are allowed.

4) unlike objects are allowed.

5) Tuples are static.

Tuple Data Type

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.

```
>>> t=10,20,30,40
>>> print(t)
(10, 20, 30, 40)
>>> print(type(t))
<class 'tuple'>
>>> t=(10,20,30,3.5,"Hi")
>>> print(t)
(10, 20, 30, 3.5, 'Hi')
>>> print(type(t))
<class 'tuple'>
>>>
```

Creating using eval() to create list and tuples

```
>>> t2 = input("ENter values")
ENter values(1,2,3)
>>> t2
'(1,2,3)'
>>> type(t2)
<class 'str'>
>>> t2 = eval(input("ENter values"))
ENter values1,2,3
>>> t2
(1, 2, 3)
>>> type(t2)
<class 'tuple'>
>>>
```

```
>>> t2 = eval(input("ENter values"))
ENter values[1,2,3]
>>> t2
[1, 2, 3]
>>> type(t2)
<class 'list'>
>>>
```

Creating tuple with tuple()

It is also possible to use the tuple() constructor to make a tuple.

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

```
>>> t = tuple()
>>> type(t)
<class 'tuple'>
>>> l = list()
>>> type(l)
<class 'list'>
>>> t=tuple('a','b','c')
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    t=tuple('a','b','c')
TypeError: tuple expected at most 1 argument, got 3
>>> t=tuple(('a','b','c'))
>>> t
('a', 'b', 'c')
>>>
```

Creating a tuple with one element

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'>
```

Different Types of elements

A tuple can have any number of items and they may be of different types (integer, float, list, 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))
```

Access Tuple Elements

Insertion order is preserved in Tuples. Hence the elements of the tuple can be accessed through indexing and slicing like lists.

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 in the next slide.

```
# 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
```

Access Tuple Elements

Negative Indexing

Tuple 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
```

Access Tuple Elements

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')
my_tuple[1:4]

#elemrnts beinning to 2nd
# Output: ('p', 'r')
my_tuple[:-7]

#elements 8th to end
my_tuple[7:]
# ('i', 'z')

# elements beginning to end
my_tuple[:]
#('p', 'r', 'o', 'g', 'r', 'a', 'm', 'i', 'z')
```

Changing a Tuple

Unlike lists, tuples are immutable.

This means that elements of a tuple cannot be changed once they have been assigned.

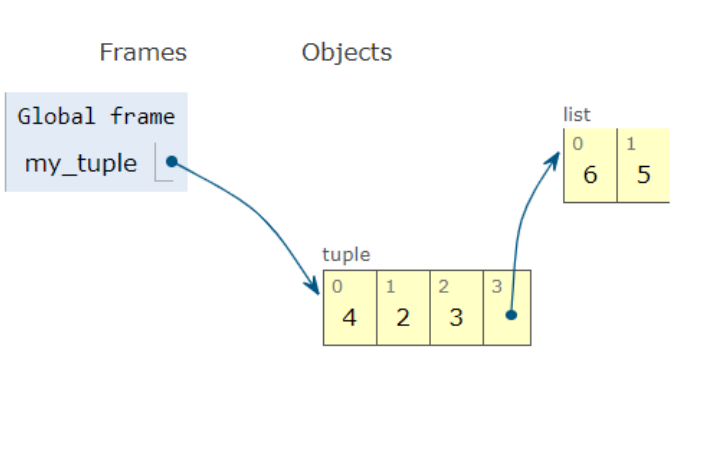
But, if the element is itself a mutable data type like list, its nested items can be changed.

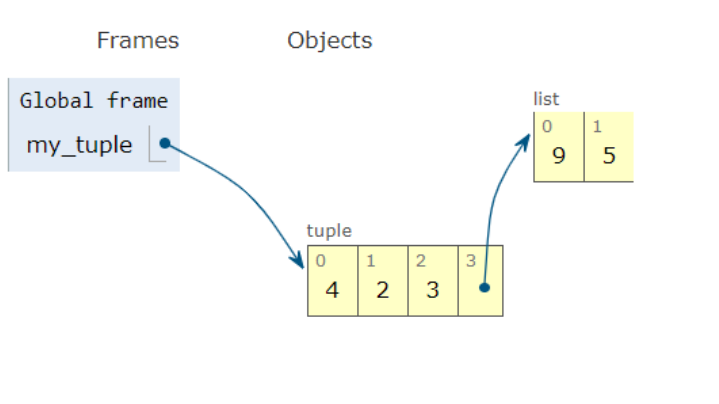
We can also assign a tuple to different values (reassignment).

```
# 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')
```





Concatenation and Repetition

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.

```
#Concatention
print((1,2,3)+(4,5,6))
#-> (1, 2, 3, 4, 5, 6)

# Repeat
print(("Repeat",)*3)
#->('Repeat', 'Repeat', 'Repeat')
```

Using the comparison operators and membership operators

The standard comparisons (<, <=, >, >=, ==, !=, in , not in) work the same among tuples as they do among lists.

The tuples are compared element by element. If the corresponding elements are the same type, ordinary comparison rules are used.

If the corresponding elements are different types, the type names are compared, since there is almost no other rational basis for comparison.

```
(3,6,2) == (3,6,2)
#->True

(3,6,2)== (3,7,0)
#->False


(3,6,2) < (3,6,2)
#->False

(3,6,2) <= (3,6,2)
#->True

(3,6,2) < (3,7,0)
#->True

(3,6,2) >= (3,7,0)
#->False
```

Compare tuples with heterogeneous items

Tuples comparison for == equality operator works for heterogeneous items. But 'less than' and 'greater than' operators does not work with different datatypes.

```
tuple1= (1,2,3)
tuple2= (1,2,"6") #3 will be compared to 6

tuple1 == tuple2
#-> False

tuple1 < tuple2
#-> TypeError: '<' not supported between instances of 'int' and 's
```

Tuple Methods

Methods that add items or remove items are not available with tuple, as tuples are immutable.

Some methods available to be used with Tuple.

1. len()
2. count()
3. index()

Tuple Methods

1) len()

To return number of elements present in the tuple.

```
t = (10,20,30,40)
len(t)
# 4
```

2) count()
To return number of occurrences of given element in the tuple

```
t = (10, 20, 10, 10, 20)
t.count(10)
# 3
```

Tuple methods

3) index()

Returns index of first occurrence of the given element.

If the specified element is not available, then we will get ValueError.

```
t = (10, 20, 10, 10, 20)

t.index(10))
# 0

t.index(30))
# ValueError: tuple.index(x): x not in tuple
```

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.

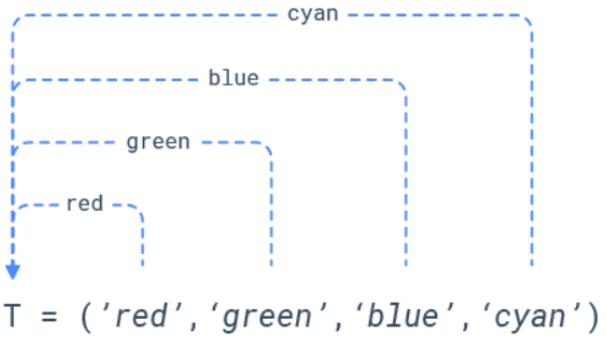
```
# Deleting a Tuples

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_tuplr[3]

# Can delete an entire tuple


del my_tuple

my_tuple
#->NameError: name 'my_tuple' is not
defined
```

Tuple packing

When a tuple is created, the items in the tuple are packed together into the object.

```
T= ('red','green', 'blue', 'cyan’)
print(T)
#->('red', 'green', 'blue', 'cyan')
```

In above example, the values ‘red’, ‘green’, ‘blue’ and ‘cyan’ are packed together in a tuple.

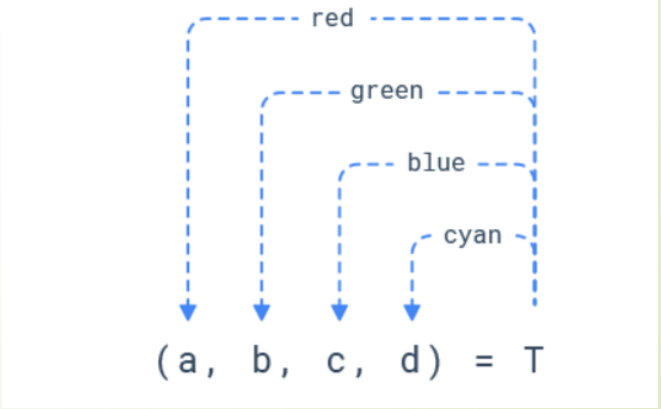


Tuple Unpacking

When a packed tuple is assigned to a new tuple, the individual items are unpacked (assigned to the items of a new tuple).

```
# common errors in tuple unpacking
T= ('red','green', 'blue', 'cyan')
(a,b,c,d)= T

a
#->'red'

b
#->'green’

c
#->'blue'

d
#->'cyan'
```

the tuple `T` is unpacked into a, b, c and d variables.



Common errors in Tuple unpacking

When unpacking, the number of variables on the left must match the number of items in the tuple.

```
# common errors in tuple unpacking

T= ('red','green', 'blue', 'cyan’)

(a,b)= T
#-> ValueError: too many values to unpack (expected 2)


T= ('red','green', 'blue')
(a,b,c,d)= T
#-> ValueError: not enough values to unpack (expected 4, got 3)
```

Usage

Tuple unpacking comes handy when you want to swap values of two variables without using a temporary variable.

```
#Swap values of 'a' and 'b'

a= 1
b= 99

id(a)
id(b)

a, b= b,a
id(a)
id(b)

a
#->99

b
#->1
```

Unpacking a Tuple

While unpacking a tuple, the right side can be any kind of sequence (tuple, string or list).

```
#Split an email address into a user name  and domain

addr = 'bob@python.org'

user, domain = addr.split('@')

user
#-> bob

domain
#-> python.org
```

Set

Sets

Sets in mathematics, are simply a collection of **distinct** objects forming a group.

A set can have any group of items, be it a collection of numbers, days of a week, types of vehicles, or the items you wear and so on.

Every item in the set is called an element of the set

What is a set?

A set is an unordered collection of similar items.

Every set element is unique **(no duplicates).**

A set is mutable. We can add or remove items from it.

Since insertion order is not preserved, indexing and slicing concepts are not applicable.

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

Creating an empty set

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() constructor.

```
#Distinguish set and dictionary while
creating empty set

#initialize a with {}
a= {}

#chack data type of a
type(a)
#-> <class 'dict'>

#initalaiz a with set()
a = set()

#chack data type of a
type(a)
#-> <class 'set'>
```

Using set constructor

The set constructor “**set()**” returns a new set initialized with elements of the specified iterable.

Note: that if an element is repeated in the iterable, the resultant set silently discards the duplicate.

```
# Set of item in an iterable
s= set('abc')

print(s)
#-> {'b', 'c', 'a'}

# set of successive integers
s= set(range(0,4))

print(s)
#-> {0, 1, 2, 3}


# Convert list into set
s= set([1,2,3])
print(s)

#->  s= set([1,2,3])
```

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.).

```
# 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)
```

```
#Set cannot have duplicates

my_set= {1,2,3,4,3,2}
print(my_set)
#->{1, 2, 3, 4}

#we can make set from a list
my_set= set([1,2,3,2])


# set cannot have mutable items
# here [3,4] is a mutable list
# this will cause an error.
my_set= {1,2,[3,4]}

#-> TypeError: unhashable type: 'list'
```

```
s= {1, 'abc', ('a','b'),True}
print(s)

s= {[1,2], 'abc', ('a','b'),True}
#-> TypeError: unhashable type: 'list'
```

But a **set cannot have mutable elements** like lists, sets or dictionaries as its elements.

Concatenation and Repetition

Concatenation and Repetition operators will **not** work for Sets.

```
s= {1,2,3,4}

s1= {5,6,7,8}

s2= s +s1
#->TypeError: unsupported operand type(s) for +: 'set' and 'set'

s3= s*3
#->TypeError: unsupported operand type(s) for *: 'set' and 'int'
```

Comparisons on sets

Set supports set to set comparisons. Two sets are equal if and only if every element of each set is contained in the other (each is a subset of the other).

{1,2,3} == {1,2,3}

A set is less than another set if and only if the first set is a proper subset of the second set (is a subset but is not equal).

{1,2,3} < {1,2,3,4 }

A set is greater than another set if and only if the first set is a proper superset of the second set (is a superset but is not equal).

{1,2,3,4} > {1,2,3}

Comparison operators

```
s1= {1,2,3,4}
s2= {1,2,3,4}

print(s1==s2)  #->True
print(s1!=s2)  #->False
print( s1<s2)  #->False
print( s1>s2)  #->False
print( s1<s2)  #->False
print( s1<=s2) #->True
print( s1>=s2) #->True
```

```
s7=
{"James",1,"Python"}
s8=
{"James",1,"Python"}

print(s7==s8)
print(s7!=s8)
print(s7<s8)
print(s7>s8)
print(s7<s8)
print(s7<=s8)
print(s7>=s8)
```

```
s5= {"Apple", "Ball", "Cat"}
s6= {"apple", "Ball", "Cat"}

print(s5==s6)
print(s5 < s6)
print(s5 > s6)
print(s5 != s6)
```

```
s3 = {10,20,30,40}
s4 = {10,20,30}

print(s3==s4)
print(s3<s4)
print(s3>s4)
print(s3<=s4)
print(s3>=s4)
```

Membership operator

To check if a specific item is present in a set, you can use **in** and **not in** operators with if statement.

```
# Check for presence
S= {'red', 'green', 'blue'}

if 'red' in S:
    print("yes")

# Check for absence
S= {'red', 'green', 'blue'}

if 'yellow' not in S:
    print("yes")
```

Accessing a set

Sets are unordered, hence indexing has no meaning.

We cannot access or change an element of a set using indexing or slicing.

But you can loop through the set items using a for loop (we'll see more on this later) or ask if a specified value is present in a set, by using the **in** keyword.

```
#Loop through the set, and print the values:

thisset = {"apple", "banana", "charry"}

for x in thisset:
    print(x)
```

```
#Check if "banana" is present in the set:

thisset = {"apple", "banana", "charry"}

print("banana" in thisset)
```

Accessing a set

USE iter() AND next() TO GET AN ELEMENT FROM A SET

Call iter(collection) with collection as a set to convert it to an iterator object.

Call next(iterator, default) with iterator as the iterator returned in the previous step and default set to None to get the next element, or None if there are no elements remaining.

Accessing a set - Example

```
print(a_set)
```

OUTPUT

```
{1, 2}
```

```
iterator = iter(a_set)
item1 = next(iterator, None)
```

Gets an item from `iterator`

```
print(item1)
```

OUTPUT

```
1
```

```
item2 = next(iterator, None)
```

Get next item from `iterator`

```
print(item2)
```

OUTPUT

```
2
```

```
item3 = next(iterator, None)
```

`iterator` is empty

```
print(item3)
```

OUTPUT

```
None
```

```
a_set= {1,2}

print(a_set)

iterator = iter(a_set)
item1 =next(iterator, None)

print(item1)
item2 =next(iterator, None)

print(item2)
item3 =next(iterator, None)

print(item3)
```

Modifying a set

We can add a single element using the add() method, and multiple elements using the update() method.

The update() method can take tuples, lists, strings or other sets as its argument.

In all cases, duplicates are avoided.

```
print(my_set)

# 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}
```

```
my_set= {1,3}
print(my_set)
#->{1, 2, 3}

#add multiple elements
my_set.add(2)

print(my_set)
#-> {1, 2, 3, 4}

#add list and set
my_set.update([2,3,4])
print(my_set)
#-> {1, 2, 3, 4}

#->add list and set
my_set.update([4,5], {1,6,8})
print(my_set)
#->{1, 2, 3, 4, 5, 6, 8}
```

update()

Update() – can take only iterable(string, list…) as its arguments and not single values

But it can take more than one iterable argument at a time.

```
s={1,2,3,4}

s.update([5,6,7])

print(s)
#->{1, 2, 3, 4, 5, 6, 7}

s.update(8)

s.update([8,9],"String")

print(s)
#->{1, 2, 3, 4, 5, 6, 8}
```

```
>>> s = {1,2,3,4}
>>> s.update([5,6,7])
>>> s
{1, 2, 3, 4, 5, 6, 7}
>>> s.update(8)
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    s.update(8)
TypeError: 'int' object is not iterable
>>> s.update([8,9],"String")
>>> s
{1, 2, 3, 4, 5, 6, 7, 8, 9, 'r', 'i', 'n', 't', 'g', 'S'}
>>>
```

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.

```python
# 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)
```

```python
#Differencr between discard() and remove()

#initialze my_set
my_set = {1,3,4,5,6}

#discard an element
my_set.discard(4)
print(my_set)

#->

#remove an element
my_set.remove(6)
print(my_set)

#->

#discard an element not present in my_set
my_set.discard(2)
print(my_set)


# remove an element not present in my_set you will get an error.
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
```

Removing elements from a set

- 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.
- Also, the **del** keyword can be used to delete an entire set, but not a single item, as indexes cannot be specified for an element.

```
#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())

#clear my_set
my_set.clear()
print(my_set)
```

```
s8= {1,2,3,4}
print(s8)

s9= {1,2,3,4,100,200}
print(s9)

del s8
print(s8)

del s9[100]# (NEW)
del s9[1]

s={1,2}

print(s.pop())
print(s.pop())
print(s.pop())
```

Find Set Size

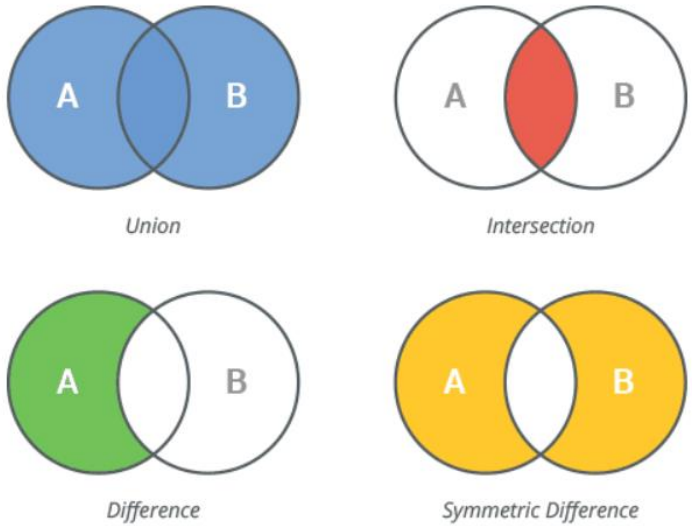
To find how many items a set has, use len() method.

```
S={'red','green','blue'}
print(len(S))
#-> 3
```

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.



Set Union

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

A = {1, 2, 3, 4, 5} ; B = {4, 5, 6, 7, 8}

Union of A and B is a set of all elements from both sets. {1, 2, 3, 4, 5, 6, 7, 8}

Union is performed using **|** operator Or **union()** method.

You can specify as many sets, you want, separated by commas.

It does not have to be a set; it can be any iterable object.

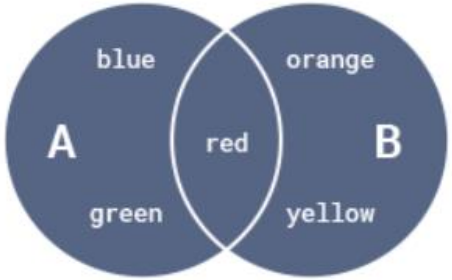
If an item is present in more than one set, the result will contain only one appearance of this item

Syntax

```
set.union(set1, set2...)
```

Parameter Values

Parameter	Description
set1	Required. The iterable to unify with
set2	Optional. The other iterable to unify with. You can compare as many iterables as you like. Separate each iterable with a comma



Union of the sets A and B is the set of all items in either A or B

```
x={"a","b","c"}
y={"c","d","e"}
z={"f","g","c"}

result= x.union(y,z)
print("Union:", result)

result= x|y|z
print("Union (|):", result)
```

```
A = {'red', 'green', 'blue'}
B = {'yellow', 'red', 'orange'}

#by operator
print(A|B)
#->{'blue', 'green', 'orange', 'yellow', 'red'}

#by method
print (A.union(B))
#->{'blue', 'green', 'orange', 'yellow', 'red'}
```

Set Intersection

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

Intersection is performed using & operator or intersection() method.

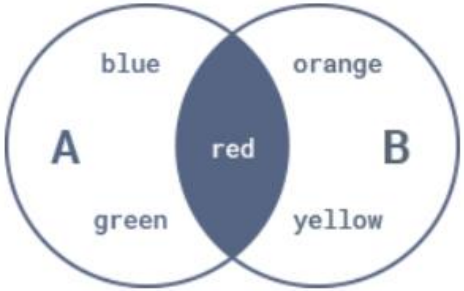
```
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

print(A & B)
print(A.intersection(B))
```

Syntax

```
set.intersection(set1, set2 ... etc)
```

Parameter Values

Parameter	Description
set1	Required. The set to search for equal items in
set2	Optional. The other set to search for equal items in. You can compare as many sets you like. Separate the sets with a comma



Intersection of the sets A and B is the set of items common to both A and B.

```
A = {'red', 'green', 'blue'}
B = {'yellow', 'red', 'orange'}

#by operator
print(A & B)
#->{'red'}

#by method
print (A.intersection(B))
#->{'red'}
```

Examples

```
# Intersection of sets
# Initialize A and B

A= {1, 2, 3, 4, 5}
B= {4, 5, 6, 7, 8}

# Use & operator
print(A & B)
print(B & A)


# use intersection function on A
print(A.intersection(B))
print( B.intersection(A))
```

```
A = {'red', 'green', 'blue'}
B = {'yellow', 'red', 'orange'}

#by operator
print(A & B)
#->{'red'}

#by method
print (A.intersection(B))
#->{'red'}
```

Set Difference

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 or difference() method.

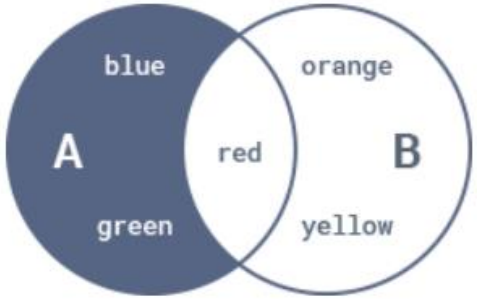
Syntax

```
set.difference(set)
```

Parameter Values

Parameter	Description
set	Required. The set to check for differences in

```
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

#by operator
print(A - B)
print(B - A)
```



Set Difference of A and B is the set of all items that are in A but not in B.

```
A = {'red', 'green', 'blue'}
B = {'yellow', 'red', 'orange'}

#by operator
print(A - B)

#by method
print (A.difference(B))
```

Set Difference

```
# Difference of two sets
# initialize A and B

A= {1, 2, 3, 4, 5}
B= {4, 5, 6, 7, 8}

# Use - operator on A
print(A-B)


#use difference funtion on A
print(A.difference(B))

# Use - operator on B
print(A-B)


#use difference funtion on B
print( B.difference(A))
```

```
x = {"a","b","c"}
y = ("c","d","e")

result= x-y
#->TypeError: unsupported operand type(s) for -: 'set' and 'tuple'

result = x.difference(y)
print(result)
```

Set Symmetric Difference

You can compute symmetric difference between two or more sets using symmetric_difference() method or ^ operator.

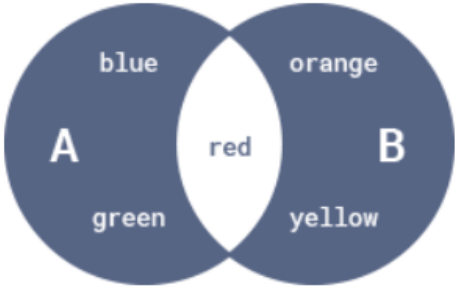
Syntax

```
set.symmetric_difference(set)
```

Parameter Values

Parameter	Description
set	Required. The set to check for matches in

```
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}

#by operator
print(A ^ B)

#by method
print(A.symmetric_difference(B))
```



Symmetric difference of sets A and B is the set of all elements in either A or B, but not both.

```
A = {'red', 'green', 'blue'}
B = {'yellow', 'red', 'orange'}

#by operator
print(A ^ B)
#->

#by method
print (A.symmetric_difference(B))
#->
```

```
x = {"a","b","c"}
y = ("c","d","e")

#result= x^y
#->TypeError: unsupported operand type(s) for ^: 'set' and 'tuple'

result = x. symmetric_difference(y)
print(result)
```

Aliasing and Cloning

Both work very similar to tuples and lists.

```
names= {"Steve", "Rick", "Negan"}
names2 = names

# Adding a new element in the new set
names2.add("Glenn")


# Removing an elemant from the old set
names.romove("Negan")


print("*"* 30)
print("Aliasing")
print("*"* 30)
print("Old set is", names)
print("New set is", names2)

print(id(names))
print(id(names2))

print(id(names) == id(names2))
```

```
**********
Aliasing
**********
Old Set is: {'Glenn', 'Steve', 'Rick'}
New Set is: {'Glenn', 'Steve', 'Rick'}
```

Aliasing and Cloning

```
15
16  # A Set of names
17  names = {"Steve", "Rick", "Negan"}
18
19  # copying using the copy() method
20  names2 = names.copy()
21
22  # adding "Glenn" to the new set
23  names2.add("Glenn")
24
25  # removing "Negan" from the old set
26  names.remove("Negan")
27
28  # displaying both the sets
29
30  print('*' * 30)
31  print("Cloning")
32  print('*' * 30)
33  print("Old Set is:", names)
34  print("New Set is:", names2)
```

```
# A set of names
names= {"Steve", "Rick", "Negan"}

# Copying using the copy() method
names2 = names.copy()

# Adding a new element in the new set
names2.add("Glenn")

# Removing an elemant from the old set
names.romove("Negan")

print("*"* 30)
print("Cloning")
print("*"* 30)
print("Old set is", names)
print("New set is", names2)

print(id(names))
print(id(names2))

print(id(names) == id(names2))
```

```
**********
Cloning
**********
Old Set is: {'Steve', 'Rick'}
New Set is: {'Steve', 'Rick', 'Negan', 'Glenn'}
```

Frozenset

frozenset

Python provides another built-in type called a frozenset. Frozenset is just like set but is immutable (unchangeable).

You can create a frozenset using the **frozenset()** method. It freezes the given sequence and makes it unchangeable.

Syntax: There are two ways of using the constructor:

```
frozenset() -> new empty frozenset

frozenset(iterable) -> new frozenset initialized with elements in iterable
```

frozenset()

You can create an empty frozenset by calling the constructor:

```
>>> frozenset()
frozenset()
```

If you pass an iterable—such as a list, tuple, set, or dictionary—you obtain a new frozenset object with elements obtained from the iterable:

```
s= frozenset({'red','green' ,'blue'})
print(s)

#->frozenset({'red', 'green', 'blue'})
```

```
>>> frozenset([1, 2, 3])
frozenset({1, 2, 3})
```

Operations on frozenset

You cannot Modify a Frozenset Once Created

As stated earlier, a **frozenset** type object has items that do not change during their lifetime. While you can use methods available for Python sets with frozenset type objects, an exception is *raised* if you attempt to use a method that changes the original **frozenset** object itself.

```
l = [1,2,3,4]
new_set = set(l)
new_set.remove(1)
print(new_set)
#->{2, 3, 4}

fset= frozenset(l)
fset.remove(1)
#->AttributeError: 'frozenset' object has no attribute 'remove'
```

Methods that Work with Frozensets

All Python set methods that do not modify items of a set work with frozenset type objects.

So you can compare two frozensets for equalities or inequalities, iterate / loop through them, get common or unique elements between two frozensets, and so on.

Next slide shows an example showing some of the methods that work with frozensets.

```python
# Frozensets
# initifs1lize fs1 and fs2
fs1 = frozenset([1, 2, 3, 4])
fs2 = frozenset([3, 4, 5, 6])

# copying fs1 frozenset
fs1_copy = fs1.copy()  # Output: frozenset({1, 2, 3, 4})
print(fs1_copy)

# union
print(fs1.union(fs2))  # Output: frozenset({1, 2, 3, 4, 5, 6})

# intersection
print(fs1.intersection(fs2))  # Output: frozenset({3, 4})

# difference
print(fs1.difference(fs2))  # Output: frozenset({1, 2})

# symmetric_difference
print(fs1.symmetric_difference(fs2))  # Output: frozenset({1, 2, 5, 6})
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS C:\Users\Angela\Desktop\PythonScripts\Batch-5\Mine> & C:/Users/Angela/AppData/L
39/python.exe c:/Users/Angela/Desktop/PythonScripts/Batch-5/Mine/fset_op.py
frozenset({1, 2, 3, 4})
frozenset({1, 2, 3, 4, 5, 6})
frozenset({3, 4})
frozenset({1, 2})
frozenset({1, 2, 5, 6})
```

```python
# Frozensets
# intifs1lize fs1 and fs3

fs1= frozenset([1,2,3,4])
fs2= frozenset([3,4,5,6])


# Copying fs1 frozenset
fs1_copy =fs1.copy() # Output: frozenset({1,2,3,4})
print(fs1_copy)
#->


#union
print(fs1.union(fs2))
#->

#intersection
print(fs1.intersection(fs2))
#->

#difference
print(fs1.intersection(fs2))
#->


#difference
print(fs1.symmetric_difference(fs2))
#->
```

```python
set_one = (1,2,3,4, 'Five', 5)
fset_one = frozenset(set_one)

set_two = (1,2,3,4, 'Five', 5)
fset_two = frozenset(set_two)

set_three = (1,2,3,4, 'Five')
fset_three = frozenset(set_three)

print(fset_two == fset_one)
print(fset_one < fset_two)  # can be set_one<=set_two
print(fset_one > fset_two)  # can be set_one>=set_two
print(fset_one > fset_three)  # fset_three is a subset of fset_one
print(fset_one < fset_three)  # fset_three is a subset of fset_one
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS C:\Users\Angela\Desktop\PythonScripts\Batch-5\Mine> & C:/Users/Angela/AppDa
39/python.exe c:/Users/Angela/Desktop/PythonScripts/Batch-5/Mine/fset_comp_op.
True
False
False
True
False
```

```python
set_one = (1,2,3,4, "Five", 5)
fset_one = frozenset(set_one)

set_two = (1,2,3,4, "Five", 5)
fset_two  = frozenset(set_two)

set_three = (1,2,3,4, "Five")
fset_three= frozenset(set_three)

print(fset_one==fset_two)
#->

print(fset_one < fset_two ) # can be set to fset_one <= fset_two
#->

print(fset_one > fset_two ) # can be set to fset_one >= fset_two
#->

print(fset_one > fset_three) #fset_three is a subset of fset_one
#->

print(fset_one < fset_three) #fset_three is a subset of fset_one
#->
```

Accessing elements of a frozen set with iter and next()

```
1   set_one = (1,2,3,4, 'Five', 5)
2   fset_one = frozenset(set_one)
3
4   element_fset_one = iter(fset_one)
5   next_element = next(element_fset_one,None)
6   print(next_element)
7   next_element = next(element_fset_one,None)
8   print(next_element)
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS C:\Users\Angela\Desktop\PythonScripts\Batch-5\Mine
39/python.exe c:/Users/Angela/Desktop/PythonScripts/E
1
2
```

```
set_one= (1,2,3,4, 'Five', 5)
fset_one = frozenset(set_one)

element_fset_one = iter(fset_one)

next_element = next(element_fset_one, None)
print(next_element)

next_element = next(element_fset_one,None)
print(next_element)
```

Dictionary

Dictionary

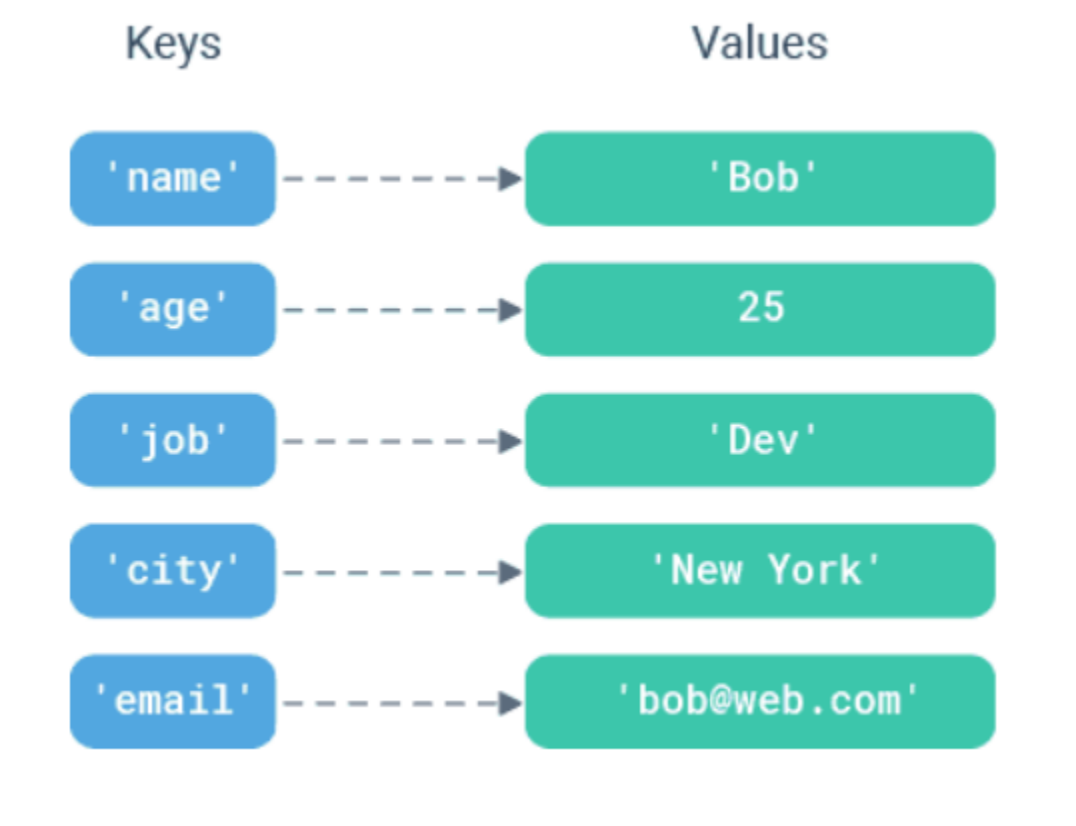
Python dictionary is an unordered collection of items. Each item of a dictionary has a **key/value pair**.

You can think of a dictionary as a mapping between a set of indexes (known as keys) and a set of values.

Each key maps to a value. The association of a key and a value is called a **key:value** pair or sometimes an item.

As an example, we'll build a dictionary that stores employee record.

Example of a Dictionary-Employee Record



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, number or tuple with immutable elements) and must be unique.

```
d = {
    <key>: <value>,
    <key>: <value>,
      .
      .
      .
    <key>: <value>
}
```

Creating a dictionary - Example

```
# Create a dictionary to store employee record

staff= { 'name' : 'Bob',
       'age': 25,
       'job' : 'Dev ',
       'city' : 'New York ',
       'email' : 'bob@web.com'
}

print(staff['name'])
```

The dict() Constructor

You can convert two-valued sequences into a dictionary with Python's dict() constructor.

The first item in each sequence is used as the key and the second as the value.

```
# Create a dictionary with a list of two-item tuples
list = [('name', 'Bob'),
          ('age', 25),
          ('job', 'Dev') ]

staff= dict(list)
print(staff)
```

```
# Create a dictionary with a list of two-item tuples

truple_list = (['name', 'Bob'],
      ['age', 25],
      ['job', 'Dev'])

staff = dict(truple_list )
print(staff)
```

More Examples:

```
>>> d = dict()
>>> d
{}
```

```
# 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')])
```

```
>>> dict(foo=100, bar=200)
{'foo': 100, 'bar': 200}
```

```
d= dict()
print(d)
#-> {}

# empty dictionary
my_dict= {}

#dictonary with integer keys
my_dict ={1:'apple', 2:'ball'}

#dictionry with mixed keys
my_dict = {'name':'john', 1:[2,4,3]}

#using dict()
my_dict = dict({1:'apple', 2:'bell'})

#from sequence having each item as a pair
my_dict = dict([(1,'apple'),(2,'bell')])

dict(foo=100, bar=200)
print(d)
```

Creating a new dictionary with default value

The fromkeys() method creates a new dictionary with default value for all specified keys.

If default value is not specified, all keys are set to None.

Syntax:

```
dict.fromkeys(keys,value)
```

Parameter	Condition	Description
keys	Required	An iterable of keys for the new dictionary
value	Optional	The value for all keys. Default value is None.

Example

```
# Crete a dictionary and set defaulf value 'Developer' for all keys
D = dict.fromkeys(['Bob','Sam'], 'Developer')
print(D)

#->{'Bob': 'Developer', 'Sam': 'Developer’}



#-> if default Value argument is not specified, all keys are to None.
#######################################################################
D = dict.fromkeys(['Bob','Sam'])
print(D)
#-> {'Bob': None, 'Sam': None}

D= dict.fromkeys(['Bob','Sam'], ('Optum','Developer'))
print(D)

#-> {'Bob': ('Optum', 'Developer'), 'Sam': ('Optum', 'Developer')}
```

```
#Python Dictionary fromKeys()
StarWars = ('Luke', 'Vader', 'Ray', 'Yoda')
StarTrek = ('Spock')
universe = dict.fromkeys(StarWars,StarTrek)
print("1.",universe)
#->1. {'Luke': 'Spock', 'Vader': 'Spock', 'Ray': 'Spock', 'Yoda': 'Spock’}
##################################################
StarWars = ('Luke', 'Vader', 'Ray', 'Yoda')
universe = dict.fromkeys(StarWars)
print("2.",universe)
#->2. {'Luke': None, 'Vader': None, 'Ray': None, 'Yoda': None}
##################################################
# Creatr a dictionary from, Python List
StarWars = ['Luke', 'Vader', 'Ray', 'Yoda']
StarTrek = 'Spock'
universe= dict.fromkeys(StarWars, StarTrek)
print("3.",universe)
#->3. {'Luke': 'Spock', 'Vader': 'Spock', 'Ray': 'Spock', 'Yoda': 'Spock’}
##################################################
keys= dict.fromkeys(universe)
print("4.", keys)
#-> {'Luke': None, 'Vader': None, 'Ray': None, 'Yoda': None}
```

Important Properties of a Dictionary

Dictionaries are pretty straightforward, but here are a few points you should be aware of when using them.

Keys must be unique: A key can appear in a dictionary only once.
Even if you specify a key more than once during the creation of a dictionary, the last value for that key becomes the associated value.

Important Properties of a Dictionary

Notice that the first occurrence of ‘name’ is replaced by the second one.

```
# Create a dictionary with a list of two-item tuples

D = {'name': 'Bob',
     'age': 25,
     'name' : 'Jane'}

print(D)
#-> {'name': 'Jane', 'age': 25}
```

Important Properties of a Dictionary

Key must be immutable type:

You can use any object of immutable type as dictionary keys – such as numbers, strings, booleans or tuples.

```
D = {(2,2): 25,
     True: 'a',
     'name': 'Bob'}
```

An exception is raised when mutable object is used as a key.

```
# TypeError: unhashable type: 'list'
D = {[2,2]: 25,
     'name': 'Bob'}
```

```
staff= {(2,2):25,
    True: 'a',
    'name': 'Bob'}


# An exception is raised when mutable object is
used as a key.


staff= {[2,2]:25,
    True: 'a',
    'name': 'Bob'}
#TypeError: unhashable type: 'list'
```

Important Properties of a Dictionary

Value can be of any type:

There are no restrictions on dictionary values.

A dictionary value can be any type of object and can appear in a dictionary multiple times.

```
# values of different datatypes
D= {'a' : [1,2,3],
    'b': {1,2,3}}
print(D)
#-> {'a': [1, 2, 3], 'b': {1, 2, 3}}


# Duplicate values
D= {'a': [1,2],
    'b': [1,2],
    'c': [1,2]}
print(D)
#-> {'a': [1, 2], 'b': [1, 2], 'c': [1, 2]}
```

Access Dictionary Items

The order of key: value pairs is not always the same. In fact, if you write the same example on another PC, you may get a different result. In general, the order of items in a dictionary is unpredictable.

But this is not a problem because the items of a dictionary are not indexed with integer indices.

Instead, you use the keys to access the corresponding values.

Access Dictionary Items

A dictionary is a similar to list; here is a list that contains the number of days in the each month:

```
days = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
```

If we want the number of days in January, use days[0]. December is days[11] or days[-1].

Here is a dictionary of the days in the months:

```
days = {'January':31, 'February':28, 'March':31, 'April':30,
'May':31, 'June':30, 'July':31, 'August':31,
'September':30, 'October':31, 'November':30, 'December':31}
```

To get the number of days in January, we use days['January'].

Access Dictionary Items

You can fetch a value from a dictionary by referring to its key in square brackets [].

```
D = {'name': 'Bob',
     'age': 25,
     'job': 'Dev'}

print(D['name'])
# Prints Bob
```

If you refer to a key that is not in the dictionary, you'll get an exception.

```
print(D['salary'])
# Triggers KeyError: 'salary'
```

```
staff= { 'name' : 'Bob',
                'age': 25,
                'job' : 'Dev' }


print(staff['name'])
#-> Bob


# if you refer to key that  is not in the dictionary, you'll get an exception
print(staff['salary' ])
#->  KeyError: 'salary'
```

Accessing Elements from Dictionary

To avoid such exception, you can use the special dictionary **get()** method.

Syntax:

```
dict.get(key[, value])
```

get() Parameters

get() method takes maximum of two parameters:

- key - key to be searched in the dictionary
- value (optional) - Value to be returned if the key is not found. The default value is None.

Return Value from get()

get() method returns:

the value for the specified key, if key is in dictionary.

None, if the key is not found and value is not specified.

A value, if the key is not found and value is specified.

```
person = {'name': 'Phill', 'age': 22}


print('Name:' , person.get('name')) #Name: Phill
print('Age:' , person.get('age'))   #Age: 22


#Value is not provided
print('Salary:', person.get('Salary')) # Salary: None


#Value is provided
print('Salary:', person.get('salary', 'Not found' ))#
Salary: 0.0
```

Output

```
Name:  Phill
Age:  22
Salary:  None
Salary:  0.0
```

```python
# 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'
```

```python
# get vs [] for retrieving elements
my_dict= {'name': 'Jack', 'age':26}

print(my_dict['name'])
# -> Jack

print(my_dict.get('age'))
# -> 26

print(my_dict.get('address'))
# -> None

#KeyError
print(my_dict['address'])
#Traceback (most recent call last):
#  File "<stdin>", line 2, in <module>
#KeyError: 'address'
```

Insert an Item Into a Dictionary

There is no add(), insert() or append() methods that you can use to add items into your dictionary. Instead, you have to create a new key to store the value in your dictionary.

If the key already exists in the dictionary, then the value will be overwritten.

In case the key is not present, a new (key: value) pair is added to the dictionary.

Changing and Adding Dictionary elements

Dictionaries are mutable.

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

```
#Chaging and adding Dictionary Elements
my_dict= {'name': 'Jack', 'age':26}

#Update value
my_dict['age']=27

print(my_dict)
#->{'name': 'Jack', 'age': 27}

#add item
my_dict['address']= 'Downtown'

print(my_dict)
#->{'name': 'Jack', 'age': 27, 'address': 'Downtown'}
```

Output

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

update()

The update() method updates the dictionary with the key:value pairs from element.

If the key is already present in the dictionary, value gets updated.
If the key is not present in the dictionary, a new **key:value** pair is added to the dictionary.

element can be either another dictionary object or an iterable of **key:value** pairs (like list of tuples).

Passing Different Arguments

Syntax

```
dictionary.update(element)
```

Parameter	Condition	Description
`element`	Optional	A dictionary or an iterable of key:value pairs

`update()` method accepts either another dictionary object or an iterable of `key:value` pairs (like tuples or other iterables of length two).

```
# Passing a dictionary object
D = {'name': 'Bob'}
D.update({'job': 'Dev', 'age': 25})
print(D)
# Prints {'job': 'Dev', 'age': 25, 'name': 'Bob'}
```

```
# Passing a list of tuples
D = {'name': 'Bob'}
D.update([('job', 'Dev'), ('age', 25)])
print(D)
# Prints {'age': 25, 'job': 'Dev', 'name': 'Bob'}
```

```
# Passing an iterable of length two (nested list)
D = {'name': 'Bob'}
D.update([['job', 'Dev'], ['age', 25]])
print(D)
# Prints {'age': 25, 'job': 'Dev', 'name': 'Bob'}
```

```
# Passing a dictionary object
D= {'name':'Bob'}
D.update({'job':'Dev', 'age':25})
print(D)
# ->{'name': 'Bob', 'job': 'Dev', 'age': 25}

# Passing a list of tuples
D= {'name':'Bob'}
D.update([('job','Dev'), ('age',25)])
print(D)
# -> {'name': 'Bob', 'job': 'Dev', 'age': 25}


# Passing a iterable of length two (nested list)
D= {'name':'Bob'}
D.update([['job','Dev'], ['age',25]])
print(D)
# -> {'name': 'Bob', 'job': 'Dev', 'age': 25}
```

```
staff = {'name':'Bob'}
staff.update([{'job','Dev'},{'age',25}]) # List of sets
print(' staff :', staff)
#->D: {'name': 'Bob', 'Dev': 'job', 25: 'age'}


#Prints D: {'name':'Bob', 'Dev' : 'job', 'age':25}
D= {'name': 'Bob'}
D.update({('job','Dev'),('age',25)}) # set of tuples
print('D:',D)
#->D: {'name': 'Bob', 'job': 'Dev', 'age': 25}
```

```
staff= {'name':'Bob'}


staff.update(job='Dev',age=25)
print(staff)


#->{'name': 'Bob', 'job': 'Dev', 'age': 25}
```

Remove elements in a dictionary

To remove an element in a dictionary, we can use

- `del` **`dict[key]`** keyword
- **`dict.pop(key[, default])`** method.
- **`popitem()`** method

Remove elements in a dictionary

The del dict[key] keyword removes the given element from the dictionary, raising a **KeyError** if key does not exist.

If key exists in the dictionary, the dict.pop(key[, default]) method removes the item with the given key from the dictionary and returns its value.

On the contrary, if key does not exist in the dictionary, the method returns the default value(if one provided). If no default value is provided and key does not exist, the dict.pop() method will raise an exception (**KeyError**).

The dict.popitem() method removes the item that was last inserted into the dictionary.

In versions before 3.7, the popitem() method removes a random item.

Removing elements from Dictionary

All the items can be removed at once, using the **clear()** method.

The **del keyword** can be used 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
print(squares.pop(4))
print(squares)

#remove an abitary item, retuen (key, value)

print(squares.popitem())
print(squares)

#remove all items
squares.clear()
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
```

```
orgi_dict= {
    'shopping': 'Amazon',
     'transport':'Ola',
     'banking':'Paytm',
     'hotel':'oyo rooms'
}

print("orgi_dict (Before removal):", orgi_dict)
removed_item = orgi_dict.pop('shopping')
print ("orgi_dict(After removal-shopping): ",orgi_dict)
print (removed_item)

removed_item = orgi_dict.pop('shopping',None)
print("orig_dict(After removal-shopping): ",orgi_dict)
print (removed_item)

removed-item = orgi_dict.pop('shopping')
print("orgi_dict(After removal-shopping): ",orgi_dict)
print(removed_item)
```

Get All Keys, Values and Key:Value Pairs

There are three dictionary methods that return all of the dictionary's keys, values and key-value pairs: keys(), values(), and items(). These methods are useful in loops that need to step through dictionary entries one by one.

All the three methods return iterable object. If you want a true list from these methods, wrap them in a list() function.

Example

```
staff= {'name':'Bob',
    'age': 25,
    'job':'Dev'}

#Get all keys
print(list(staff.keys()))

#['name', 'age', 'job']

# Get all values
print(list(staff.values()))
#['Bob', 25, 'Dev']



# Get all values
print(list(staff.items()))

#[('name', 'Bob'), ('age', 25), ('job', 'Dev')]
```

```
>>> D.keys()
dict_keys(['name', 'age', 'job'])
>>> D.values()
dict_values(['Angela', 25, 'Dev'])
```

```
>>> D.items()
dict_items([('name', 'Angela'), ('age', 25), ('job', 'Dev')])
>>> list(D.items())
[('name', 'Angela'), ('age', 25), ('job', 'Dev')]
>>> list(D.values())
['Angela', 25, 'Dev']
```

Operators support with Dictionaries

‘+’ and ‘*’ **not** supported for Dictionaries

Equality operators (‘==‘ and ‘!=‘) can be used to check for the equality of the dictionaries.

Relational operators (‘>’,’<‘,<=,>=) are not supported

Membership operators(in and not in) are supported, **But** it checks for keys only and not values.

```
>>> D = {1:'A',2:'B',3:'C'}
>>> type(D)
<class 'dict'>
>>> D1 = {4:'D'}
>>> type(D1)
<class 'dict'>
>>> D+D1
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    D+D1
TypeError: unsupported operand type(s) for +: 'dict' and 'dict'
>>> D*3
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    D*3
TypeError: unsupported operand type(s) for *: 'dict' and 'int'
>>> D2 = {1:'A',2:'B',3:'C'}
>>> D == D2
True
>>> D != D1
True
```

```
D= {1:'A', 2:'B',3:'C'}
print(type(D))
print(D1={4:'D'})
print(D+D1)
D*3


D2= {1:'A', 2:'B',3:'C'}
print( D == D2)
print( D != D1)
print( D < D1)
print( D > D1)
```

```
>>> D < D1
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    D < D1
TypeError: '<' not supported between instances of 'dict' and 'dict'
>>> D > D1
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    D > D1
TypeError: '>' not supported between instances of 'dict' and 'dict'
>>>
```

Check if a Key or Value Exists

If you want to know whether a key exists in a dictionary, use in and not in operators.

```
staff= {'name': 'Bob',
         'age': 25,
         'job':'Dev'}

print('Bob' in staff)
print('salary' in staff)
```

Check if a Key or Value Exists

To check if a certain value exists in a dictionary, you can use the method, values(), which returns the values as a list, and then use the in operator.

```
staff = {'name': 'Bob',
          'age': 25,
         'job':'Dev'}

print('Bob' in staff.values())
print('Sam' in staff.values())
```

Find Dictionary Length

To find how many key: value pairs a dictionary has, use len() method.

```
staff= {'name': 'Bob',
          'age': 25,
          'job':'Dev'}


print(len(staff))
```

setdefault()

Returns the value for key if exists, else inserts it

Usage

The setdefault() method returns the value for key if key is in the dictionary. If not, it inserts key with a value of default and returns default.

setdefault() Parameters

Syntax

```
dictionary.setdefault(key,default)
```

Parameter	Condition	Description
key	Required	Any key you want to return value for
default	Optional	A value to insert if the specified key is not found. Default value is None.

Return Value from setdefault()

setdefault() returns:

value of the key if it is in the dictionary

None, if the key is not in the dictionary and default_value is not specified.

default_value if key is not in the dictionary and default_value is specified.

How setdefault() works when key is in the dictionary?

If key is in the dictionary, the method returns the value for key (no matter what you pass in as default)

```
# without default specified
D = {'name': 'Bob', 'age': 25}
v = D.setdefault('name')
print(v)
# Prints Bob
# with default specified
D = {'name': 'Bob', 'age': 25}
v = D.setdefault('name', 'Max')
print(v)
# Prints Bob
```

```
#without default specified
D= {'name':'Bob', 'age': 25}
V=D.setdefault('name')

print(V)

#with default specified
D={'name':'Bob', 'age':25}
V=D.setdefault('name','Max')
print(V)

V=D.setdefault('salary')
print(V)
D
```

How setdefault() works when key is not in the dictionary?

Key Absent, Default Not Specified

If key is not in the dictionary and default is not specified, the method inserts key with a value None and returns None.

```
D = {'name': 'Bob', 'age': 25}
v = D.setdefault('job')
print(D)
# Prints {'job': None, 'age': 25, 'name': 'Bob'}
print(v)
# Prints None
```

```
D= {'name': 'Bob', 'age': 25}
V= D.setdefault('job')
print(D)
print(V)

V=D.setdefault('salary',25000)
print(D)
print(V)
```

Aliasing and cloning

Aliasing and cloning works the same way as the lists, tuples or set.

Because dictionaries are mutable, you need to be aware of aliasing. Whenever two variables refer to the same object, changes to one affect the other.

If you want to modify a dictionary and keep a copy of the original, use the copy method. For example, opposites is a dictionary that contains pairs of opposites:

```
>>> opposites = {'up': 'down', 'right': 'wrong', 'true': 'false'}
>>> alias = opposites
>>> copy = opposites.copy()
```

Aliasing and cloning

alias and opposites refer to the same object; copy refers to a fresh copy of the same dictionary. If we modify alias, opposites is also changed:

```
>>> alias['right'] = 'left'
>>> opposites['right']
'left'
```

If we modify copy, opposites is unchanged:

```
>>> copy['right'] = 'privilege'
>>> opposites['right']
'left'
```

Dictionaries and lists

Dictionaries and lists share the following characteristics:

Both are mutable.

Both are dynamic. They can grow and shrink as needed.

Both can be nested. A list can contain another list. A dictionary can contain another dictionary. A dictionary can also contain a list, and vice versa.

Dictionaries differ from lists primarily in how elements are accessed.

List elements are accessed by their position in the list, via indexing.

Dictionary elements are accessed via keys.

Python Data Type Cheatsheet

String	List	Tuple	Set	Dictionary
Immutable	Mutable	Immutable	Mutable	Mutable
Ordered/Indexed	Ordered/Indexed	Ordered/Indexed	Unordered	Unordered
Allows Duplicate Members	Allows Duplicate Members	Allows Duplicate Members	Doesn't allow Duplicate Members	Doesn't allow Duplicate keys
Empty string = ""	Empty list = [ ]	Empty tuple = ( )	Empty set = set()	Empty dictionary = { }
String with single element = "H"	List with single item = ["Hello"]	Tuple with single item = ("Hello",)	Set with single item = {"Hello"}	Dictionary with single item = {"Hello": 1}
---	It can store any data type str, list, set, tuple, int and dictionary	It can store any data type str, list, set, tuple, int and dictionary	It can store data types (int, str, tuple) but not (list, set, dictionary)	Inside of dictionary key can be int, str and tuple only values can be of any data type int, str, list, tuple, set and dictionary