Department of Computer Science and Engineering
PES University, Bangalore, India

Python For Computational Problem Solving (UE22CS151A)

**Mr. Prakash C O**
**Asst. Professor,**
**Dept. of CSE, PESU**
**coprakasha@pes.edu**

---

## Sets in Python - set()

- A set is a collection of distinct elements.
- Set is unordered, unindexed, but mutable.
- In Python sets are written with curly brackets and set elements are separated by a comma.

**Examples:**

```
fruitset = {“apple”, “orange”, “kiwi”, “banana”, “cherry”}
set1 = {(1, 'a', 'hi'), 2, 'hello', 5.56, 3+5j, True}
```

A set is a data structure with the following attributes.

- Set elements are unique – does not support repeated elements
- Set elements should be hashable (set can only include hashable objects)

### What is hashing or hash function?

Hashing is the process of generating a value from a text or a list of numbers using a mathematical function known as a hash function.

A Hash Function is a function that converts a given numeric or alphanumeric key to an integer value.

The mapped integer value is used as an index in the hash table. In simple terms, a hash function maps a significant number or string to an integer that can be used as an index in the hash table.

### Hashable object:

o An object is hashable if it has a hash value which never changes during its lifetime (it needs a **hash**() method), and can be compared to other objects (it needs an **eq**() method). Hashable objects which compare equal must have the same hash value.

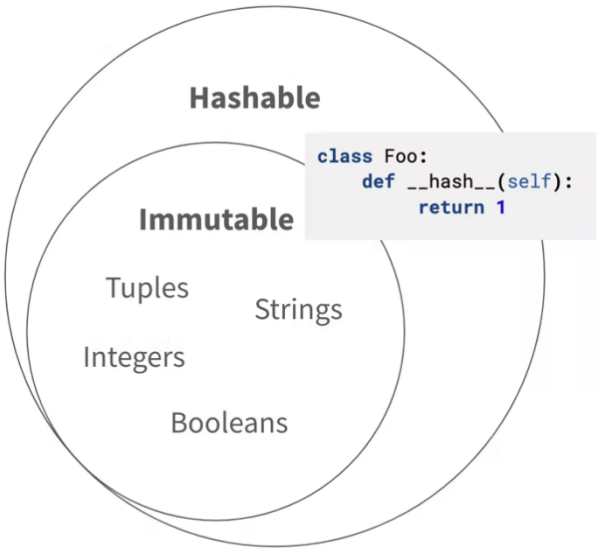
o Hashability makes an object usable as a dictionary key and a set member, because these data structures use the hash value internally.

o All of Python’s immutable built-in objects are hashable, while no mutable containers (such as lists or dictionaries) are.

Objects which are instances of user-defined classes are hashable by default; they all compare unequal, and their hash value is their id().

## Immutable vs. Hashable

- Immutable - A type of object that cannot be modified after it was created.
- Hashable - A type of object that you can call **hash( )** on.
- All immutable objects are hashable, but not all hashable objects are immutable.
- Python Sets can only include hashable objects.




**Examples of hashable objects:**

- int, float, complex, bool, string, tuple, range, frozenset, bytes, decimal

**Examples of Unhashable objects:**

- list, dict, set, bytearray

**Examples for hashable objects and unhashable objects**

In [ ]:
```
# Hashable python objects
print('hash({}) = {}'.format(55,hash(55)))
print('hash({}) = {}'.format(5.5,hash(5.5)))
print('hash({}) = {}'.format(True,hash(True)))
print('hash({}) = {}'.format(False,hash(False)))
print('hash({}) = {}'.format('Hi',hash('Hi')))
print('hash({}) = {}'.format('Hello',hash('Hello')))
my_tuple=(1,3,5)
print('hash({}) = {}'.format(my_tuple,hash(my_tuple)))
print('hash({}) = {}'.format('range(5)',hash(range(5))))
```

In [ ]:
```
# Unhashable python objects
l=[1,2,3]
#print(hash(l))    # TypeError: unhashable type: 'list'
s={1, 5.5, 'hello'}
#print(hash(s))    # TypeError: unhashable type: 'set'
d={1:'hi', 2:'hello', 3:'how r u'}
#print(hash(d))    # TypeError: unhashable type: 'dict'
t=(10,'hi',[1,2,3])
print(hash(t))    # TypeError: unhashable type: 'list'
```

**Invalid examples for sets**

In [4]:
```
set1={ [1,'a','hi'], 2, 'hello', {3, 4.5, 'how r u' } } #TypeError: unhashable type: 'li
set2={ 1, 'hello', {1:'hi', 2:'hello', 3:'how r u'} }   #TypeError: unhashable type: 'di
```

Note: Sets cannot be nested

In [ ]:
```
set3={ 2, 'hello', {3, 4.5, 'how r u' } } #TypeError: unhashable type: 'set'
```

```
print(set3)
```

If a tuple contains any mutable object either directly or indirectly(is not hashable), it cannot be used as a set element.

#### Valid examples for sets

- Only hashable object can be used in a set.

In [5]:
```
set1={ (1, 'hi'), 2, 'hello', 5.56, 3+5j, True }
print(set1)
```

```
{True, 2, (3+5j), 5.56, (1, 'hi'), 'hello'}
```

In [21]:
```
set2={1,2,(1,2,3),(1,2,3)}  # Set does not support repeated elements,
print(set2)                 # hence set removes duplicates.
```

```
{1, 2, (1, 2, 3)}
```

In [22]:
```
set3={1,2,(1,2,3),(1,3,2)}
print(set3)
```

```
{1, 2, (1, 2, 3), (1, 3, 2)}
```

In [36]:
```
t1=(1,2,3)
t2=t1
set1={1,2,t1,t2} # Set does not support repeated elements, hence set removes duplicates
print(set1)
```

```
{1, 2, (1, 2, 3)}
```

#### Set is mutable, mutable objects can change their values

In [15]:
```
set4={( 1, 5.5, 'hi' ), 2, 'hello', 5.56, 3+5j}
print('set4 =',set4)
set4.add(True)
print('set4 =',set4)

print(set4.pop())
```

```
set4 = {2, 'hello', (3+5j), 5.56, (1, 5.5, 'hi')}
set4 = {True, 2, 'hello', (3+5j), 5.56, (1, 5.5, 'hi')}
True
```

#### Set is unordered – we cannot assume the order of elements in a set.

#### Set is an iterable – eager and not lazy

In [35]:
```
set4={ ( 1, 5.5, 'hi' ), 2, 'hello', 5.56, 3+5j}
for ele in set4:
    print(ele,end=' ')
```

```
hello 2 (3+5j) 5.56 (1, 5.5, 'hi')
```

We can check for membership using the in operator. This would be faster in case of a set compared to a list, a tuple or a string.

In [13]:
```
my_set = { 11, 22, 33, 44, 55}
print(11 in my_set) # True
print(77 in my_set) # False
```

```
True
False
```

#### Note: To create an empty set, we must use the set constructor set() and not { }. The latter would become a dict.

In [14]:
```
s1=set()     # empty set
print(type(s1))
s2={}        # empty dictionary, not an empty set
print(type(s2))
```

```
<class 'set'>
<class 'dict'>
```

**We use sets for**

- Deduplication : removal of repeated elements
- Finding unique elements
- Comparing two iterables for common elements or differences.

In [3]:
```
# deduplication : removed repeated elements
l = [11, 33, 22, 11, 33, 11, 11, 44, 22]
s = set(l)
l = list(s)
print(l)
```

```
[33, 11, 44, 22]
```

In [4]:
```
# deduplication : removed repeated elements
l = [11, 33, 22, 11, 33, 11, 11, 44, 22]
l = list(set(l))
print(l)
```

```
[33, 11, 44, 22]
```

In [6]:
```
s5 = set("mississippi")
print(s5)       # {'s', 'i', 'm', 'p'}
```

```
{'m', 'i', 'p', 's'}
```

**Note: Sets are unordered, so you cannot be sure in which order the items will appear.**

In [2]:
```
s6 = { 10, 30, 10, 40, 20, 50, 30 }
print(s6)      # elements are unique; there is no particular order

# set :
   #    not sequence
   #    no concept of an element in a particular position
   #    represents a finite set of math
```

```
{50, 20, 40, 10, 30}
```

In [7]:
```
# set is an iterable
s6 = { 10, 30, 10, 40, 20, 50, 30 }
print(s6)

for i in s6 :
    print(i, end = " ")
```

```
{50, 20, 40, 10, 30}
50 20 40 10 30
```

## Sets support many mathematical operations.

In [ ]:
```
o       Membership : in
o       Union : |
o       Intersection : &
o       Set difference :  -
o       Symmetric difference : ^
o       Equality and inequality:  ==   !=
o       Subset and superset :  <  <=  >  >=
```

In [19]:
```
# set operations
s1 = {1, 2, 3, 4, 5}
s2 = {1, 3, 5, 7, 9}

# Union A∪B : the set of all elements that belong to set A and set B.
print(s1 | s2)
```

```
{1, 2, 3, 4, 5, 7, 9}
```

In [20]:
```
# set operations
s1 = {1, 2, 3, 4, 5}
s2 = {1, 3, 5, 7, 9}
# Intersection A∩B : the set of common elements that belong to set A and B.
print(s1 & s2) # {1, 3, 5}
```

```
{1, 3, 5}
```

In [15]:
```
# set operations
s1 = {1, 2, 3, 4, 5}
s2 = {1, 3, 5, 7, 9}
# set difference A-B : the set of all the elements that are in set A but not in set B
print(s1 - s2) # {2, 4}
print(s2 - s1) # {9, 7}
# A-B : set of elements that are only in A but not in B.
```

```
{2, 4}
{9, 7}
```

In [22]:
```
# set operations
s1 = {1, 2, 3, 4, 5}
s2 = {1, 3, 5, 7, 9}
# Symmetric difference
# A ^ B : set of elements in A and B but not in both(excluding the intersection).
print(s1 ^ s2) # {2, 4, 7, 9}
```

```
{2, 4, 7, 9}
```

In [ ]:
```
print("what : ", s1[0]) # TypeError: 'set' object does not support indexing
```

In [49]:
```
# Equality and inequality:  ==   !=
s1={1,2,3,4}
s2={4,1,3,2}
if (s1 == s2):
    print('sets are equal')
else:
    print('sets are not equal')
```

```
sets are equal
```

In [1]:
```
s1={1,2,3}
s2={1,2,3}  # or s2=set(s1)
if (s1 is s2): # Compares references
    print('sets having same references')
else:
    print('sets not having same references')
```

```
sets not having same references
```

In [2]:
```
s1={1,2,3}
s2=s1
if (s1 is s2):
    print('sets having same references')
else:
    print('sets not having same references')
```

```
sets having same references
```

If A is a subset of B, then A is contained in B. It implies that B contains A, or in other words, B is a superset of A.

If A ⊂ B and A ≠ B, this means that A is a proper subset of B. And B is known as the superset of set A.

In [31]:
```
#Subset and superset :  <  <=  >  >=
s1={1,2,3}
s2={1,2,3,4}
if (s1 < s2):
    print('s1 is proper subset of s2')
else:
    print('s1 is not a proper subset of s2')
```

```
s1 is proper subset of s2
```

**Finding unique elements(characters) in a string**

In [5]:
```
s1='Hello How are you?'
print(set(s1))
```

```
{'l', 'r', 'y', '?', 'o', 'u', 'w', 'a', 'e', ' ', 'H'}
```

**Finding unique words in a string str1; words separated by white space**

In [3]:
```
str1='Hello How are you? How are you doing?'
l=str1.split()
print(l)
print(set(l))
#print(set(str1.split()))
```

```
['Hello', 'How', 'are', 'you?', 'How', 'are', 'you', 'doing?']
{'How', 'you?', 'are', 'Hello', 'you', 'doing?'}
```

In [6]:
```
str1=input('Enter multiword string:  ')
#l=str1.split()
#print(set(l))
print(set(str1.split()))
```

```
Enter multiword string:Hello How are you? How are you doing
{'you?', 'doing', 'How', 'you', 'are', 'Hello'}
```

## Create a set of numbers from 2 to n.

**Method 1:**

Use a for loop; iterate on the range object range(2, n + 1); add each element to the set

In [7]:
```
n=10
my_set=set()
for ele in range(2, n + 1):
    my_set.add(ele)
print(my_set)
```

```
{2, 3, 4, 5, 6, 7, 8, 9, 10}
```

**Method 2:**

Pass the range object as an argument to the set constructor. This method is clean and elegant. s = set(range(2, n + 1))

In [8]:
```
n=10
my_set=set(range(2, n + 1))
print(my_set)
```

```
{2, 3, 4, 5, 6, 7, 8, 9, 10}
```

#### Check whether a set is empty

In [4]:
```
# Approach 1:
s1={1,2,3}
if len(s1) == 0:
    print ("set is empty")
else:
    print ("set is not empty")
```

```
set is not empty
```

In [6]:
```
# Approach 2:
s1={1,2,3}
if s1== set():   # Comparing with Another Empty Set
    print ("set is empty")
else:
    print ("set is not empty")
```

```
set is not empty
```

In [ ]:
```
# Approach 3:
s1={1,2,3}
if s1 :
    print("set is not empty")
else:
    print("set is empty")

# empty data structure => False
# non-empty data structure => True
```

## Remove elements from a set

Given a set s3={1, 2, 3, 4, 5, 6, 7, 8, 9}, remove even numbers.

#### Method 1:

Iterate through a range or a list object containing 2, 4, 6, 8. Call remove for each element on the set.

In [9]:
```
s3={1, 2, 3, 4, 5, 6, 7, 8, 9}
for i in [2,4,6,8]:
    s3.remove(i)
print(s3)
```

```
{1, 3, 5, 7, 9}
```

#### Method 2:

Create a set of the elements to be removed – use set difference to remove the elements. This is preferred. s3 = s3 - set(range(2, 10, 2))

In [10]:
```
s3={1, 2, 3, 4, 5, 6, 7, 8, 9}
s3 = s3 - set(range(2, 10, 2))
print(s3)
```

```
{1, 3, 5, 7, 9}
```

## Set Methods

Python has a set of built-in methods that you can use on sets.

| Method | Description |
|---|---|
| add() | Adds an element to the set |
| clear() | Removes all the elements from the set |
| copy() | Returns a copy of the set |
| difference() | Returns a set containing the difference between two or more sets |
| difference_update() | Removes the items in this set that are also included in another, specified set |
| discard() | Remove the specified item |
| intersection() | Returns a set, that is the intersection of two other sets |
| intersection_update() | Removes the items in this set that are not present in other, specified set(s) |
| isdisjoint() | Returns whether two sets have a intersection or not |
| issubset() | Returns whether another set contains this set or not |
| issuperset() | Returns whether this set contains another set or not |
| pop() | Removes an element from the set<br>**Note:** Sets are unordered, so when using the pop() method, you will not know which item that gets removed. |
| remove() | Removes the specified element |
| symmetric_difference() | Returns a set with the symmetric differences of two sets |
| symmetric_difference_update() | inserts the symmetric differences from this set and another |
| union() | Return a set containing the union of sets |
| update() | Update the set with the union of this set and others |

add(obj)

- Adds an element to a set.
- s.add(obj) adds obj, which must be an hashable object, to set s.

In [9]:
```
s1={ (1, 'hi'), 2, 5.56 }
s1.add(True)
s1.add('hello')
#s1.add([1,2])  #TypeError: unhashable type: 'list'
print(s1)
```

```
{True, 2, 5.56, 'hello', (1, 'hi')}
```

pop()

- Removes a random element from a set.
- s.pop() removes and returns an arbitrarily chosen element from s. If s is empty, s.pop() raises an exception.

In [ ]:
```
s1=set()
print(s.pop()) # KeyError: 'pop from an empty set'
```

In [8]:
```
s2={True, 4, 5.56, 'hello', (1, 'hi')}
```

```
print(s2.pop())
print(s2)
print(s2.pop())
print(s2)
```

```
True
{4, 5.56, (1, 'hi'), 'hello'}
4
{5.56, (1, 'hi'), 'hello'}
```

**remove(obj) and discard(obj) methods**

- If the item to remove does not exist, remove() will raise an error.
- If the item to remove does not exist, discard() will NOT raise an error.

In [4]:
```
s4 = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
print(s4.remove(1))
print(s4.discard(2))
print(s4)
```

```
None
None
{3, 4, 5, 6, 7, 8, 9, 10}
```

In [6]:
```
s4 = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
#print(s4.remove(12)) # KeyError: 12
print(s4.discard(12))
```

```
None
```

**x1.isdisjoint(x2)**

- Determines whether or not two sets have any elements in common.
- returns True if x1 and x2 have no elements in common
- There is no operator that corresponds to the .isdisjoint() method.

In [15]:
```
x1={1,2,3,4,5}
x2={6,7,8,9,10}
print(x1.isdisjoint(x2))
```

```
True
```

**x1.issubset(x2)**

- is same as x1 <= x2
- Determine whether one set is a subset of the other.
- return True if x1 is a subset of x2
- A set x1 is considered a subset of another set x2 if every element of x1 is in x2.

In [17]:
```
x1={1,2,3,4,5}
x2={1,2,3,4,5,6,7,8,9,10}
print('x1.issubset(x2) =',x1.issubset(x2))
print('x1.issubset(x1) =',x1.issubset(x1))
```

```
x1.issubset(x2) = True
x1.issubset(x1) = True
```

In [18]:
```
x1 = {'foo', 'bar', 'baz'}
x2 = {'baz', 'qux', 'quux'}
x3 = {'foo', 'bar', 'baz', 'qux', 'quux'}
print('x1.issubset(x3) =',x1.issubset(x3))
print('x1 <= x3 =',x1 <= x3)
print('x1 <= x2 =',x1 <= x2)
```

```
x1.issubset(x3) = True
x1 <= x3 = True
x1 <= x2 = False
```

While a set is considered a subset of itself, if it is not a proper subset of itself

**x1 < x2 returns True if x1 is a proper subset of x2**

In [3]:
```
x1 = {'foo', 'bar'}
x2 = {'foo', 'bar', 'baz'}
print(x1 < x2)

x1 = {'foo', 'bar', 'baz'}
x2 = {'foo', 'bar', 'baz'}
print(x1 < x2)
```

```
True
False
```

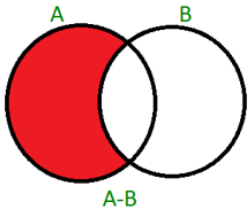
**x1.issuperset(x2)**

- is same as x1 >= x2
- Determine whether one set is a superset of the other.
- return True if x1 is a superset of x2
- A superset is the reverse of a subset.
- A set x1 is considered a superset of another set x2 if x1 contains every element of x2.

In [20]:
```
x1={1,2,3,4,5}
x2={1,2,3,4,5,6,7,8,9,10}
print('x1.issuperset(x2) =',x1.issuperset(x2))
print('x2.issuperset(x1) =',x2.issuperset(x1))
print('x1.issuperset(x1) =',x1.issuperset(x1))
```

```
x1.issuperset(x2) = False
x2.issuperset(x1) = True
x1.issuperset(x1) = True
```

**Difference (A-B): Set of elements that are only in A but not in B**




**A.difference(B[, C ...]) and A.difference_update(B[, C ...])**

If A and B are two sets.

- The set difference() method will get the (A – B) and will return a new set.
- The set difference_update() method modifies the existing set.
  - If (A – B) is performed, then A gets modified into (A – B)
  - if (B – A) is performed, then B gets modified into (B – A)

In [38]:
```
A={1,2,3,4,5,6}
B={4,5}
print(A.difference(B)) # returns a new set
print('A =',A)
```

```
A.difference_update(B) # modifies the existing set. If (A - B) is performed, then A gets
print('A =',A)
```

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

**In Python, why is a tuple hashable but not a list?**

Dictionary and other objects use hashes to store and retrieve items really quickly. The mechanics of this all happens "under the covers" - you as the programmer don't need to do anything and Python handles it all internally. The basic idea is that when you create a dictionary with {key: value}, Python needs to be able to hash whatever you used for key so it can store and look up the value quickly.

Immutable objects, or objects that can't be altered, are hashable. They have a single unique value that never changes, so python can "hash" that value and use it to look up dictionary values efficiently. Objects that fall into this category include strings, tuples, integers and so on. Immutable objects never change, so you can always be sure that when you generate a hash for an immutable object, looking up the object by its hash will always return the same object you started with, and not a modified version.

**Exercise Problems:**

1.Create a set of all even numbers between 1 and 20 that are not divisible by 4.

In [1]:
```
n = 20
s = set(range(2, n + 1, 2)) - set(range(4, n + 1, 4))
print(s)
```

```
{2, 6, 10, 14, 18}
```

2.Create a set of all even numbers between 1 and 100 that are not divisible by 8.

In [13]:
```
# Approach 1
s1=set(range(2,101,2))
print('set(range(2,101,2) =',s1)
print()
s2=set(range(8,101,8))
print('set(range(8,101,8) =',s2)
print()
print('s1-s2 =',s1-s2)
```

```
set(range(2,101,2) = {2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 3
6, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72, 74, 76, 78, 8
0, 82, 84, 86, 88, 90, 92, 94, 96, 98, 100}

set(range(8,101,8) = {32, 64, 96, 8, 40, 72, 16, 48, 80, 24, 56, 88}

s1-s2 = {2, 4, 6, 10, 12, 14, 18, 20, 22, 26, 28, 30, 34, 36, 38, 42, 44, 46, 50, 52, 5
4, 58, 60, 62, 66, 68, 70, 74, 76, 78, 82, 84, 86, 90, 92, 94, 98, 100}
```

In [5]:
```
# Approach 2
print(set(range(2,101,2))-set(range(8,101,8)))
```

```
{2, 4, 6, 10, 12, 14, 18, 20, 22, 26, 28, 30, 34, 36, 38, 42, 44, 46, 50, 52, 54, 58, 6
0, 62, 66, 68, 70, 74, 76, 78, 82, 84, 86, 90, 92, 94, 98, 100}
```

3.Create a set of all even numbers between 1 and 100 that are not divisible by 12.

In [9]:
```
# Approach 1
s=set()
for i in range(1,101):
    if (i%2 == 0) and (i%12 != 0):
```

```
        s.add(i)
print(s)
```

```
{2, 4, 6, 8, 10, 14, 16, 18, 20, 22, 26, 28, 30, 32, 34, 38, 40, 42, 44, 46, 50, 52, 54,
56, 58, 62, 64, 66, 68, 70, 74, 76, 78, 80, 82, 86, 88, 90, 92, 94, 98, 100}
```

In [5]:
```
# Approach 2
l=[]
for i in range(2,100,2):
    if (i%12 != 0):
        l.append(i)
s=set(l)
print(s)
```

```
{2, 4, 6, 8, 10, 14, 16, 18, 20, 22, 26, 28, 30, 32, 34, 38, 40, 42, 44, 46, 50, 52, 54,
56, 58, 62, 64, 66, 68, 70, 74, 76, 78, 80, 82, 86, 88, 90, 92, 94, 98}
```

In [1]:
```
# Approach 3
s = set(range(2, 101, 2)) - set(range(12, 101, 12))
print(s)
```

```
{2, 4, 6, 8, 10, 14, 16, 18, 20, 22, 26, 28, 30, 32, 34, 38, 40, 42, 44, 46, 50, 52, 54,
56, 58, 62, 64, 66, 68, 70, 74, 76, 78, 80, 82, 86, 88, 90, 92, 94, 98, 100}
```