Sequences in Python with Types and Examples

When it comes to ease of writing code, Python is the most preferred language due to its large community, extensive library support, and data structures.

Sequences in Python

Sequences are containers with items stored in a deterministic ordering. Each sequence data type comes with its unique capabilities.

There are many types of sequences in Python. Let’s learn them with Examples.

Types of Sequences in Python

Python sequences are of **six types,** namely:

1. Strings
2. Lists
3. Tuples
4. Bytes Sequences
5. Bytes Arrays
6. range() objects

Strings in Python

In python, the string is a sequence of Unicode characters written inside a single or double-quote. Python does not have any **char** type as in other languages (C, C++), therefore, a single character inside the quotes will be of type **str** only.

1. To declare an **empty string**, use **str()** or it can be defined using empty string inside quotes.

Example of Empty String in Python

```
name = "PythonGeeks"
print(name)
```

Output

```
PythonGeeks
```

2. Strings are **immutable** data types, therefore once declared, we can’t alter the string. Though, we can reassign it to a new string.

Code

```
name = "PythonGeeks"
print(name[6]) # outputs 'G'
name[6] = 'g' # throws error
print(name)
```

Output

```
G
Traceback (most recent call last):
  File
“/home/apoorve/Documents/PythonGeeks/python_sequences/main.py
”, line 3, in <module>
    name[7] = ‘g’
TypeError: ‘str’ object does not support item assignment
```

Lists in Python

Lists are a single storage unit to store multiple data items together. It’s a mutable data structure, therefore, once declared, it can still be altered.

A list can hold **strings, numbers, lists, tuples, dictionaries, etc.**

1. To declare a list, either use **list()** or square brackets [], containing comma-separated values.

Example of Lists in Python

```
list_1 = ["PythonGeeks", "Sequences", "Tutorial"] # [all string list]
print(f'List 1: {list_1}')
list_2 = list() # [empty list]
print(f'List 2: {list_2}')
list_3 = [2021, ['hello', 2020], 2.0] # [integer, list, float]
print(f'List 3: {list_3}')
list_4 = [{'language': 'Python'}, (1,2)] # [dictionary, tuple]
print(f'List 4: {list_4}')
```

Output

```
List 1: [‘PythonGeeks’, ‘Sequences’, ‘Tutorial’]
List 2: []
List 3: [2021, [‘hello’, 2020], 2.0]
List 4: [{‘language’: ‘Python’}, (1, 2)]
```

2. Let’s check the mutability of lists, now.

Code

```
list_1 = ["PythonGeeks", "Sequences", "Tutorial"] # [all string list]
list_1[2] = "Blog"
print(list_1)
```

Output

```
[‘PythonGeeks’, ‘Sequences’, ‘Blog’]
```

Tuples in Python

Just like Lists, Tuples can store multiple data items of different data types. The only difference is that they are **immutable** and are stored inside the parenthesis **()**.

1. To declare a tuple, either use **tuple()** or parenthesis, containing comma-separated values.

Example of Tuple in Python:

```
tuple_1 = ("PythonGeeks", "Sequences", "Tutorial") # [all string tuple]
print(f'tuple 1: {tuple_1}')
tuple_2 = tuple() # [empty tuple]
print(f'tuple 2: {tuple_2}')
tuple_3 = [2021, ('hello', 2020), 2.0] # [integer, tuple, float]
print(f'tuple 3: {tuple_3}')
tuple_4 = [{'language': 'Python'}, [1,2]] # [dictionary, list]
print(f'tuple 4: {tuple_4}')
```

Output:

```
tuple 1: (‘PythonGeeks’, ‘Sequences’, ‘Tutorial’)
tuple 2: ()
tuple 3: [2021, (‘hello’, 2020), 2.0]
tuple 4: [{‘language’: ‘Python’}, [1, 2]]
```

2. Let’s test the immutability of tuples now.

Code:

```
tuple_1 = ("PythonGeeks", "Sequences", "Tutorial") # [all string tuple]
tuple_1[2] = "Blog"
print(tuple_1)
```

Output:

```
Traceback (most recent call last):
File
“/home/apoorve/Documents/PythonGeeks/python_sequences/main.py
”, line 2, in <module>
tuple_1[3] = “Blog”
TypeError: ‘tuple’ object does not support item assignment
```

Byte Sequences in Python

The **bytes()** function returns an **immutable** bytes sequence in between quotes, preceded by a ‘B’ or ‘b’.

1. To declare an empty bytes object, use **bytes(size),** where size is the number of empty bytes we want to generate.

Example of Byte Sequences in Python:

```
size = 10
b = bytes(size)
print(b)
```

Output:

```
b’\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00′
```

2. Alternatively, we can pass a list of numbers in bytes() function as follows:

Code:

```
b = bytes([5,8])
print(b)
```

Output:

```
B’\x05\x08’
```

3. In the case of strings, we need to pass a second parameter as the type of encoding.

Code:

```
b = bytes("PythonGeeks", 'utf-8')
print(b)
```

Output:

```
b’PythonGeeks’
```

Byte Arrays in Python

Just like bytes sequence, bytes arrays also return a bytes object. The only difference is that they are **mutable**.
1. Let’s see how to use them.
Example of Byte Arrays in Python:

```
print(bytearray(10))
print(bytearray([5,8]))
print(bytearray("PythonGeeks", 'utf-8'))
```

Output:

```
bytearray(b’\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00′)
bytearray(b’\x05\x08′)
bytearray(b’PythonGeeks’)
```

2. Try **mutating a byte** in the array.
Code:

```
b = bytearray([10,20,60,40,50])
print(f'Before: {b}')
b[2] = 30
print(f'Later: {b}')
```

Output:

```
Before: bytearray(b’\n\x14<(2′)
Later: bytearray(b’\n\x14\x1e(2′)
```

range() object in Python

It returns a **sequence of integers** in the specified range. By default, it starts the sequence from 0 (if not specified).
Code:

```
sequence = range(10)
print(sequence)
```

Output:

```
range(0,10)
```

Code:

```
for i in range(2,5):
print(i)
```

Output:

```
2
3
4
```

As a third parameter, we can specify the **increment count** (here, 2). By default it is set to 1.

Code:

```
sequence= range(1, 10, 2)
for i in sequence:
print(i)
```

Output:

```
1
3
5
7
9
```

Operations on Sequences in Python

We now know the types of sequences, but now, it’s time to see what all operations can we perform on them. Here, we will focus on the most commonly used operations.

1. Concatenation Operator in Python

(+) operator is used to join two sequences of the same type.

Code:

```
print(["Python"] + ["Geeks"])
```

Output:

```
[“Python”, “Geeks”]
```

2. Repeat Operator in Python

(*) operator repeats a sequence specified number of times.

Code:

```
print(("PythonGeeks", 1) * 2)
```

Output:

```
('PythonGeeks', 1, 'PythonGeeks', 1)
```

3. **Membership Operator in Python**

These are used to check whether a value is present in a sequence or not using the **"in"** and **"not in"** operators.
Code:

```
dict = {
"lang" : "Python",
"platform": "PythonGeeks"
}
print("lang" in dict)
print("code" not in dict)
```

Output:

```
True
True
```

4. Slicing Operator in Python

[:] operator returns a part of the sequence between a given range.
Code:

```
list_1 = [3,2,5,6,7]
print(list_1[:5])
print(list_1[2:4])
print(list_1[-1:])
```

Output:

```
[3, 2, 5, 6, 7]
[5, 6]
[7]
```

Python Sequence Functions and Methods

some important functions and methods used for sequences of all the types.
1. len(sequence) : Returns length of a sequence.
2. index(index): Returns index of the first occurrence of an element in a sequence.

3. min(sequence): Returns the minimum value of a sequence.
4. max(sequence): Returns maximum value of a sequence.
5. count(): Returns the count of a number of occurrences of an element in a sequence.
6. append(value): Adds the **value** at the end of the sequence.
7. clear(): Clears all the contents of the sequence.
8. insert(value, index): Inserts the **value** at the index "**index**" of the sequence.
9. pop(index): Returns and deletes elements at index "**index**". By default, the last element is deleted from the sequence.
10. remove(value): Removes the first occurrence of **value** from the sequence.
11. reverse(): Reverse the sequence
Example of Sequence Function in Python

```
test = [2, 4, 6, 8, 10, 10]
print(len(test))
print(test.index(6))
print(min(test))
print(max(test))
print(test.count(10))
test.append(11)
print(test)
test.clear()
print(test)
test = [2, 4, 6, 8, 10, 10]
test.insert(9,4)
print(test)
print(test.pop())
test.remove(10)
print(test)
test.reverse()
print(test)
```

Output:

```
6
2
2
10
2
[2, 4, 6, 8, 10, 10, 11]
[]
[2, 4, 6, 8, 10, 10, 4]
4
[2, 4, 6, 8, 10]
[10, 8, 6, 4, 2]
```

Python Mapping Types

The mapping objects are used to map hash table values to arbitrary objects. In python there is mapping type called **dictionary**. It is mutable.

The keys of the dictionary are arbitrary. As the value, we can use different kind of elements like lists, integers or any other mutable type objects.

Some dictionary related methods and operations are −

Method len(d)

The len() method returns the number of elements in the dictionary.

Operation d[k]

It will return the item of d with the key 'k'. It may raise **KeyError** if the key is not mapped.

Method iter(d)

This method will return an iterator over the keys of dictionary. We can also perform this taks by using **iter(d.keys())**.

Method get(key[, default])

The get() method will return the value from the key. The second argument is optional. If the key is not present, it will return the default value.

Method items()

It will return the items using (key, value) pairs format.

Method keys()

Return the list of different keys in the dictionary.

Method values()

Return the list of different values from the dictionary.

Method update(elem)

Modify the element elem in the dictionary.

Example Code

Live Demo

```
myDict = {'ten' : 10, 'twenty' : 20, 'thirty' : 30,
'forty' : 40}
print(myDict)
print(list(myDict.keys()))
print(list(myDict.values()))

#create items from the key-value pairs
print(list(myDict.items()))

myDict.update({'fifty' : 50})
print(myDict)
```

Output

```
{'ten': 10, 'twenty': 20, 'thirty': 30, 'forty': 40}
['ten', 'twenty', 'thirty', 'forty']
[10, 20, 30, 40]
[('ten', 10), ('twenty', 20), ('thirty', 30), ('forty', 40)]
{'ten': 10, 'twenty': 20, 'thirty': 30, 'forty': 40, 'fifty': 50}
```

Python – Mapping Matrix with Dictionary

When it is required to map the matrix to a dictionary, a simple iteration is used.

Example

Below is a demonstration of the same −

```
my_list = [[2, 4, 3], [4, 1, 3], [2, 1, 3, 4]]

print("The list :")
print(my_list)

map_dict = {2 : "Python", 1: "fun", 3 : "to", 4 : "learn"}

my_result = []
for index in my_list:
  temp = []
  for element in index:
    temp.append(map_dict[element])
  my_result.append(temp)

print("The result is :")
print(my_result)
```

Output

```
The list :
[[2, 4, 3], [4, 1, 3], [2, 1, 3, 4]]
The result is :
[['Python', 'learn', 'to'], ['learn', 'fun', 'to'], ['Python', 'fun', 'to', 'learn']]
```

Python - Sets

A set is one of the built-in data types in Python. In mathematics, set is a collection of distinct objects. Set data type is Python's implementation of a set. Objects in a set can be of any data type.

Set in Python also a collection data type such as list or tuple. However, it is not an ordered collection, i.e., items in a set or not accessible by its positional index. A set object is a collection of one or more immutable objects enclosed within curly brackets {}.

Example 1

Some examples of set objects are given below −

```
s1 = {"Rohan", "Physics", 21, 69.75}
s2 = {1, 2, 3, 4, 5}
s3 = {"a", "b", "c", "d"}
s4 = {25.50, True, -55, 1+2j}
print (s1)
print (s2)
print (s3)
print (s4)
```

It will produce the following **output** −

```
{'Physics', 21, 'Rohan', 69.75}
{1, 2, 3, 4, 5}
{'a', 'd', 'c', 'b'}
{25.5, -55, True, (1+2j)}
```

The above result shows that the order of objects in the assignment is not necessarily retained in the set object. This is because Python optimizes the structure of set for set operations.

In addition to the literal representation of set (keeping the items inside curly brackets), Python's built-in set() function also constructs set object.

set() Function

set() is one of the built-in functions. It takes any sequence object (list, tuple or string) as argument and returns a set object

Syntax

```
Obj = set(sequence)
```

Parameters

- **sequence** − An object of list, tuple or str type

Return value

The set() function returns a set object from the sequence, discarding the repeated elements in it.

Example 2

```
L1 = ["Rohan", "Physics", 21, 69.75]
s1 = set(L1)
T1 = (1, 2, 3, 4, 5)
s2 = set(T1)
string = "TutorialsPoint"
s3 = set(string)

print (s1)
print (s2)
print (s3)
```

It will produce the following **output** −

```
{'Rohan', 69.75, 21, 'Physics'}
{1, 2, 3, 4, 5}
{'u', 'a', 'o', 'n', 'r', 's', 'T', 'P', 'i', 't', 'l'}
```

Example 3

Set is a collection of distinct objects. Even if you repeat an object in the collection, only one copy is retained in it.

```
s2 = {1, 2, 3, 4, 5, 3,0, 1, 9}
s3 = {"a", "b", "c", "d", "b", "e", "a"}
print (s2)
print (s3)
```

It will produce the following **output** −

```
{0, 1, 2, 3, 4, 5, 9}
{'a', 'b', 'd', 'c', 'e'}
```

Example 4

Only immutable objects can be used to form a set object. Any number type, string and tuple is allowed, but you cannot put a list or a dictionary in a set.

```
s1 = {1, 2, [3, 4, 5], 3,0, 1, 9}
print (s1)
s2 = {"Rohan", {"phy":50}}
print (s2)
```

It will produce the following **output** −

```
  s1 = {1, 2, [3, 4, 5], 3,0, 1, 9}
       ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
TypeError: unhashable type: 'list'
  s2 = {"Rohan", {"phy":50}}
       ^^^^^^^^^^^^^^^^^^^^^
TypeError: unhashable type: 'dict'
```

Python raises TypeError with a message unhashable types 'list' or 'dict'. Hashing generates a unique number for an immutable item that enables quick search inside computer's memory. Python has built-in hash() function. This function is not supported by list or dictionary.

Even though mutable objects are not stored in a set, set itself is a mutable object. Python has a special operators to work with sets, and there are different methods in set class to perform add, remove, update operations on elements of a set object.

Python - Access Set Items

Previous

Next

Since set is not a sequence data type, its items cannot be accessed individually as they do not have a positional index (as in list or tuple). Set items do not have a key either (as in dictionary) to access. You can only traverse the set items using a **for** loop.

Example 1

```
langs = {"C", "C++", "Java", "Python"}
for lang in langs:
  print (lang)
```

It will produce the following **output** −

```
Python
C
C++
Java
```

Example 2

Python's membership operators let you check if a certain item is available in the set. Take a look at the following example −

```
langs = {"C", "C++", "Java", "Python"}
print ("PHP" in langs)
print ("Java" in langs)
```

It will produce the following **output** −

```
False
True
```

Python - Add Set Items

Previous

Next

Even if a set holds together only immutable objects, set itself is mutable. We can add new items in it with any of the following ways −

add() Method

The add() method in set class adds a new element. If the element is already present in the set, there is no change in the set.

Syntax

```
set.add(obj)
```

Parameters

- **obj** − an object of any immutable type.

Example

Take a look at the following example −

```
lang1 = {"C", "C++", "Java", "Python"}
lang1.add("Golang")
print (lang1)
```

It will produce the following **output** −

```
{'Python', 'C', 'Golang', 'C++', 'Java'}
```

update() Method

The update() method of set class includes the items of the set given as argument. If elements in the other set has one or more items that are already existing, they will not be included.

Syntax

```
set.update(obj)
```

Parameters

- **obj** − a set or a sequence object (list, tuple, string)

Example

The following example shows how the update() method works −

```
lang1 = {"C", "C++", "Java", "Python"}
lang2 = {"PHP", "C#", "Perl"}
lang1.update(lang2)
print (lang1)
```

It will produce the following **output** −

```
{'Python', 'Java', 'C', 'C#', 'PHP', 'Perl', 'C++'}
```

Example

The update() method also accepts any sequence object as argument. Here, a tuple is the argument for update() method.

```
lang1 = {"C", "C++", "Java", "Python"}
lang2 = ("PHP", "C#", "Perl")
lang1.update(lang2)
print (lang1)
```

It will produce the following **output** −

```
{'Java', 'Perl', 'Python', 'C++', 'C#', 'C', 'PHP'}
```

Example

In this example, a set is constructed from a string, and another string is used as argument for update() method.

```
set1 = set("Hello")
set1.update("World")
print (set1)
```

It will produce the following **output** −

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

union() Method

The union() method of set class also combines the unique items from two sets, but it returns a new set object.

Syntax

```
set.union(obj)
```

Parameters

- **obj** − a set or a sequence object (list, tuple, string)

Return value

The union() method returns a set object

Example

The following example shows how the union() method works −

```
lang1 = {"C", "C++", "Java", "Python"}
lang2 = {"PHP", "C#", "Perl"}
lang3 = lang1.union(lang2)
print (lang3)
```

It will produce the following **output** −

```
{'C#', 'Java', 'Perl', 'C++', 'PHP', 'Python', 'C'}
```

Example

If a sequence object is given as argument to union() method, Python automatically converts it to a set first and then performs union.

```
lang1 = {"C", "C++", "Java", "Python"}
lang2 = ["PHP", "C#", "Perl"]
lang3 = lang1.union(lang2)
print (lang3)
```

It will produce the following **output** −

```
{'PHP', 'C#', 'Python', 'C', 'Java', 'C++', 'Perl'}
```

Example

In this example, a set is constructed from a string, and another string is used as argument for union() method.

```
set1 = set("Hello")
set2 = set1.union("World")
print (set2)
```

It will produce the following **output** −

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

Python - Loop Sets

Previous

Next

A set in Python is not a sequence, nor is it a mapping type class. Hence, the objects in a set cannot be traversed with index or key. However, you can traverse each item in a set using a **for** loop.

Example 1

The following example shows how you can traverse through a set using a **for** loop −

```
langs = {"C", "C++", "Java", "Python"}
for lang in langs:
  print (lang)
```

It will produce the following **output** −

```
C
Python
C++
Java
```

Example 2

The following example shows how you can run a **for** loop over the elements of one set, and use the **add()** method of set class to add in another set.

```
s1={1,2,3,4,5}
s2={4,5,6,7,8}
for x in s2:
  s1.add(x)
print (s1)
```

It will produce the following **output** −

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

Python - Join Sets

Previous
Next

In Python, a Set is an ordered collection of items. The items may be of different types. However, an item in the set must be an immutable object. It means, we can only include numbers, string and tuples in a set and not lists. Python's set class has different provisions to join set objects.

Using the "|" Operator

The "|" symbol (pipe) is defined as the union operator. It performs the A∪B operation and returns a set of items in A, B or both. Set doesn't allow duplicate items.

```
s1={1,2,3,4,5}
s2={4,5,6,7,8}
s3 = s1|s2
print (s3)
```

It will produce the following **output** −

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

Using the union() Method

The set class has union() method that performs the same operation as | operator. It returns a set object that holds all items in both sets, discarding duplicates.

```
s1={1,2,3,4,5}
s2={4,5,6,7,8}
s3 = s1.union(s2)
print (s3)
```

Using the update() Method

The update() method also joins the two sets, as the union() method. However it doen't return a new set object. Instead, the elements of second set are added in first, duplicates not allowed.

```
s1={1,2,3,4,5}
s2={4,5,6,7,8}
s1.update(s2)
print (s1)
```

Using the unpacking Operator

In Python, the "*" symbol is used as unpacking operator. The unpacking operator internally assign each element in a collection to a separate variable.

```
s1={1,2,3,4,5}
s2={4,5,6,7,8}
s3 = {*s1, *s2}
print (s3)
```

Python - Copy Sets

Previous

Next

The copy() method in set class creates a shallow copy of a set object.

Syntax

```
set.copy()
```

Return Value

The copy() method returns a new set which is a shallow copy of existing set.

Example

```
lang1 = {"C", "C++", "Java", "Python"}
print ("lang1: ", lang1, "id(lang1): ", id(lang1))
lang2 = lang1.copy()
print ("lang2: ", lang2, "id(lang2): ", id(lang2))
lang1.add("PHP")
print ("After updating lang1")
print ("lang1: ", lang1, "id(lang1): ", id(lang1))
print ("lang2: ", lang2, "id(lang2): ", id(lang2))
```

Output

```
lang1: {'Python', 'Java', 'C', 'C++'} id(lang1): 2451578196864
lang2: {'Python', 'Java', 'C', 'C++'} id(lang2): 2451578197312
After updating lang1
lang1: {'Python', 'C', 'C++', 'PHP', 'Java'} id(lang1): 2451578196864
lang2: {'Python', 'Java', 'C', 'C++'} id(lang2): 2451578197312
```

Python - Set Operators

Previous

Next

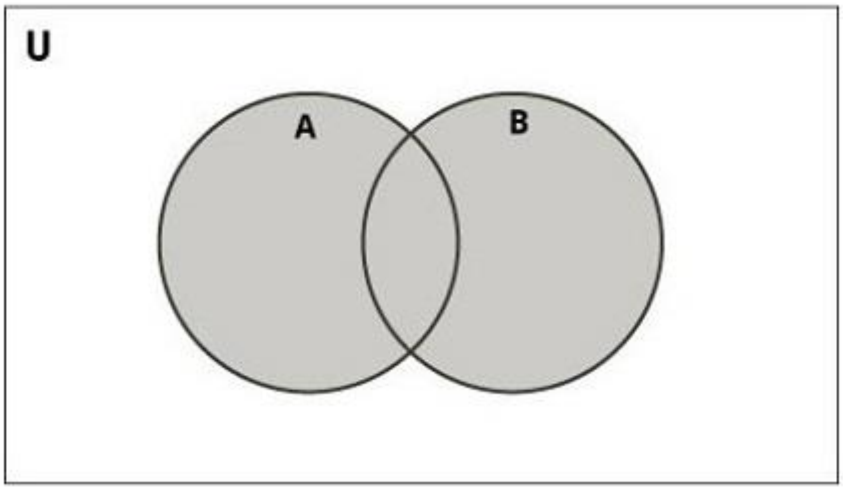
In the Set Theory of Mathematics, the union, intersection, difference and symmetric difference operations are defined. Python implements them with following operators −

Union Operator (|)

The union of two sets is a set containing all elements that are in A or in B or both. For example,

{1,2}∪{2,3}={1,2,3}

The following diagram illustrates the union of two sets.



Python uses the "|" symbol as a union operator. The following example uses the "|" operator and returns the union of two sets.

Example

```
s1 = {1,2,3,4,5}
s2 = {4,5,6,7,8}
s3 = s1 | s2
print ("Union of s1 and s2: ", s3)
```

It will produce the following **output** −

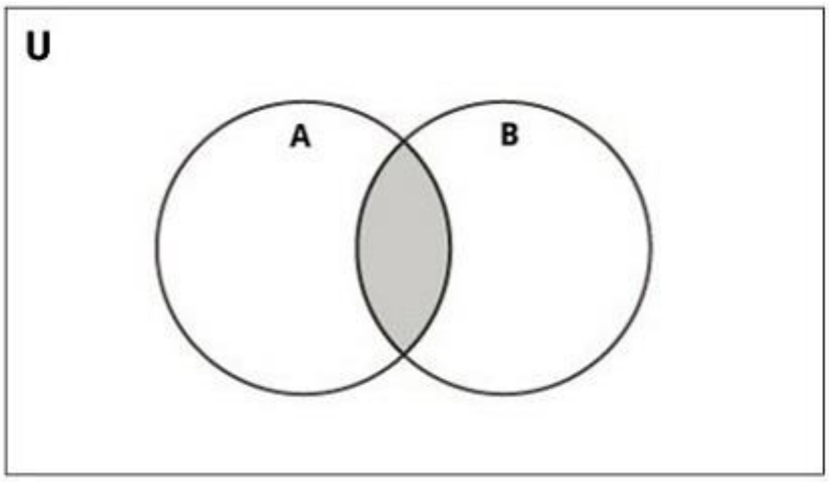
```
Union of s1 and s2: {1, 2, 3, 4, 5, 6, 7, 8}
```

Intersection Operator (&)

The intersection of two sets AA and BB, denoted by A∩B, consists of all elements that are both in A and B. For example,

{1,2}∩{2,3}={2}

The following diagram illustrates intersection of two sets.



Python uses the "&" symbol as an intersection operator. Following example uses & operator and returns intersection of two sets.

```
s1 = {1,2,3,4,5}
s2 = {4,5,6,7,8}
s3 = s1 & s2
print ("Intersection of s1 and s2: ", s3)
```

It will produce the following **output** −

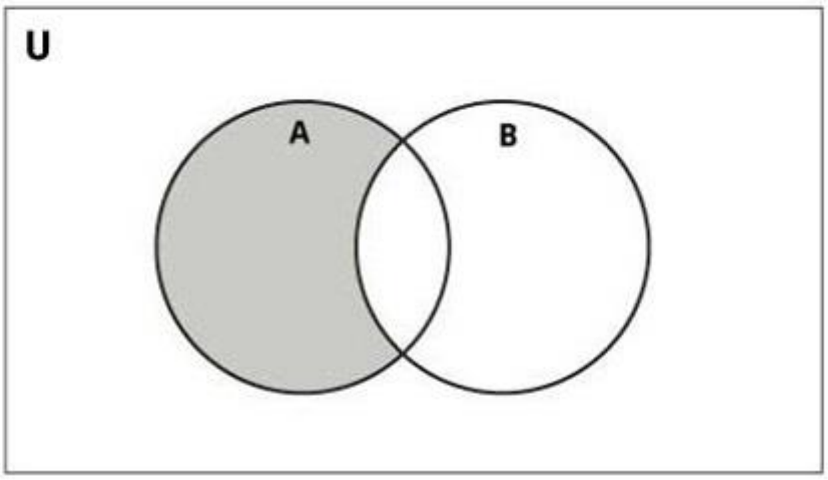
```
Intersection of s1 and s2: {4, 5}
```

Difference Operator (-)

The difference (subtraction) is defined as follows. The set A−B consists of elements that are in A but not in B. For example,

If A={1,2,3} and B={3,5}, then A−B={1,2}

The following diagram illustrates difference of two sets −



Python uses the "-" symbol as a difference operator.

Example

The following example uses the "-" operator and returns difference of two sets.

```
s1 = {1,2,3,4,5}
s2 = {4,5,6,7,8}
s3 = s1 - s2
print ("Difference of s1 - s2: ", s3)
s3 = s2 - s1
print ("Difference of s2 - s1: ", s3)
```

It will produce the following **output** −

```
Difference of s1 - s2: {1, 2, 3}
Difference of s2 - s1: {8, 6, 7}
```

Note that "s1-s2" is not the same as "s2-s1".

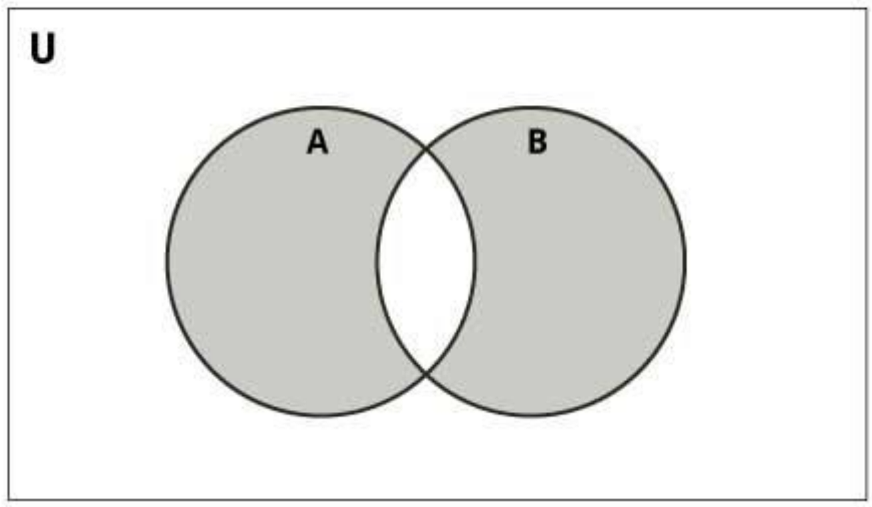
Symmetric Difference Operator

The symmetric difference of A and B is denoted by "A Δ B" and is defined by

A Δ B = (A − B) ⋃ (B − A)

If A = {1, 2, 3, 4, 5, 6, 7, 8} and B = {1, 3, 5, 6, 7, 8, 9}, then A Δ B = {2, 4, 9}.

The following diagram illustrates the symmetric difference between two sets −



Python uses the "^" symbol as a symbolic difference operator.

Example

The following example uses the "^" operator and returns symbolic difference of two sets.

```
s1 = {1,2,3,4,5}
s2 = {4,5,6,7,8}
s3 = s1 - s2
print ("Difference of s1 - s2: ", s3)
s3 = s2 - s1
print ("Difference of s2 - s1: ", s3)
s3 = s1 ^ s2
print ("Symmetric Difference in s1 and s2: ", s3)
```

It will produce the following **output** −

```
Difference of s1 - s2: {1, 2, 3}
Difference of s2 - s1: {8, 6, 7}
Symmetric Difference in s1 and s2: {1, 2, 3, 6, 7, 8}
```

Python - Set Methods

Following methods are defined in Python's set class −

Sr.No.	Methods & Description
1	**add()** Add an element to a set.
2	**clear()** Remove all elements from this set.
3	**copy()** Return a shallow copy of a set.
4	**difference()** Return the difference of two or more sets as a new set.
5	**difference_update()** Remove all elements of another set from this set.
6	**discard()** Remove an element from a set if it is a member.
7	**intersection()** Return the intersection of two sets as a new set.
8	**intersection_update()** Update a set with the intersection of itself and another.
9	**isdisjoint()** Return True if two sets have a null intersection.
10	**issubset()** Return True if another set contains this set.
11	**issuperset()** Return True this set contains another set.
12	**pop()** Remove and return an arbitrary set element
13	**remove()** Remove an element from a set; it must be a member.
14	**symmetric_difference()** Return the symmetric difference of two sets as a new set.
15	**symmetric_difference_update()** Update a set with the symmetric difference of itself and another.

16	**union()** Return the union of sets as a new set.
17	**update()** Update a set with the union of itself and others.

Python - Set Exercises

Example 1

Python program to find common elements in two lists with the help of set operations −

```
l1=[1,2,3,4,5]
l2=[4,5,6,7,8]
s1=set(l1)
s2=set(l2)
commons = s1&s2 # or s1.intersection(s2)
commonlist = list(commons)
print (commonlist)
```

It will produce the following **output** −

```
[4, 5]
```

Example 2

Python program to check if a set is a subset of another −

```
s1={1,2,3,4,5}
s2={4,5}
if s2.issubset(s1):
  print ("s2 is a subset of s1")
else:
  print ("s2 is not a subset of s1")
```

It will produce the following **output** −

```
s2 is a subset of s1
```

Example 3

Python program to obtain a list of unique elements in a list −

```
T1 = (1, 9, 1, 6, 3, 4, 5, 1, 1, 2, 5, 6, 7, 8, 9, 2)
s1 = set(T1)
print (s1)
```

It will produce the following **output** −

```
{1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Dictionaries are the data structures, which include a key value combination. These are widely used in place of JSON – JavaScript Object Notation. Dictionaries are used for API (Application Programming Interface) programming. A dictionary maps a set of objects to another set of objects. Dictionaries are mutable; this means they can be changed as and when needed based on the requirements.

How to implement dictionaries in Python?

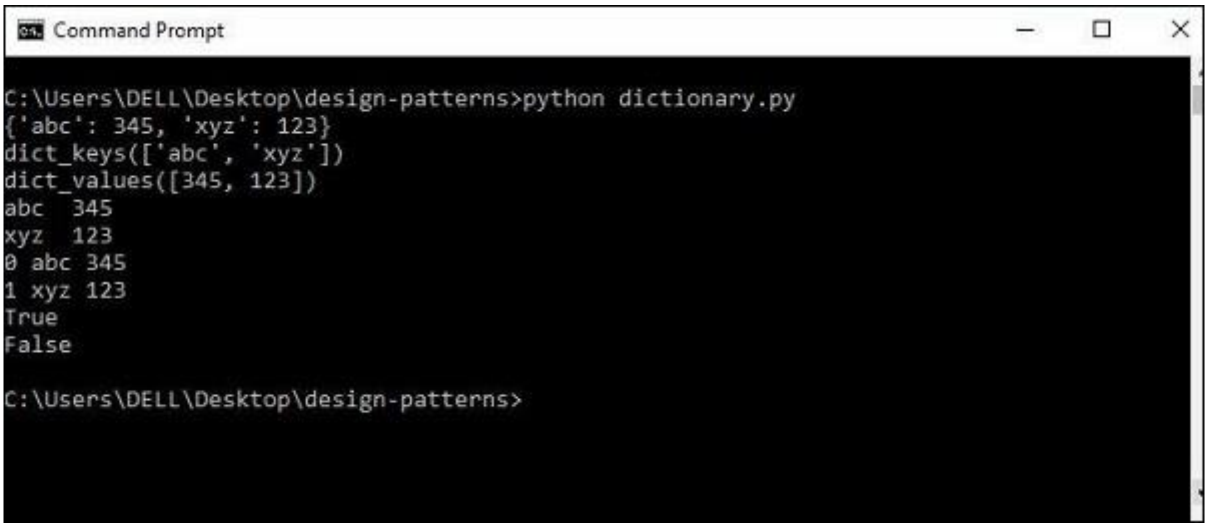
The following program shows the basic implementation of dictionaries in Python starting from its creation to its implementation.

```
# Create a new dictionary
d = dict() # or d = {}

# Add a key - value pairs to dictionary
d['xyz'] = 123
d['abc'] = 345

# print the whole dictionary
print(d)

# print only the keys
print(d.keys())

# print only values
print(d.values())

# iterate over dictionary
for i in d :
   print("%s %d" %(i, d[i]))
```

```
# another method of iteration
for index, value in enumerate(d):
   print (index, value , d[value])

# check if key exist 23. Python Data Structure –print('xyz' in
d)

# delete the key-value pair
del d['xyz']

# check again
print("xyz" in d)
```

Output

The above program generates the following output −



```
C:\Users\DELL\Desktop\design-patterns>python dictionary.py
{'abc': 345, 'xyz': 123}
dict_keys(['abc', 'xyz'])
dict_values([345, 123])
abc   345
xyz   123
0 abc 345
1 xyz 123
True
False

C:\Users\DELL\Desktop\design-patterns>
```

Drawback

Dictionaries do not support the sequence operation of the sequence data types like strings, tuples and lists. These belong to the built-in mapping type.

Python OOPs Concepts

In Python, object-oriented Programming (OOPs) is a programming paradigm that uses objects and classes in programming. It aims to implement real-world entities like inheritance, polymorphisms, encapsulation, etc. in the programming. The main concept of OOPs is to bind the data and the functions that work on that together as a single unit so that no other part of the code can access this data.

OOPs Concepts in Python

- Class
- Objects
- Polymorphism
- Encapsulation
- Inheritance
- Data Abstraction

Python OOPs Concepts

Python Class

A class is a collection of objects. A class contains the blueprints or the prototype from which the objects are being created. It is a logical entity that contains some attributes and methods.

To understand the need for creating a class let's consider an example, let's say you wanted to track the number of dogs that may have different attributes like breed, and age. If a list is used, the first element could be the dog's breed while the second element could represent its age. Let's suppose there are 100 different dogs, then how would you know which element is supposed to be which? What if you wanted to add other properties to these dogs? This lacks organization and it's the exact need for classes.

Some points on Python class:

- Classes are created by keyword class.
- Attributes are the variables that belong to a class.
- Attributes are always public and can be accessed using the dot (.) operator. Eg.: Myclass.Myattribute

Class Definition Syntax:

```
class ClassName:
   # Statement-1
   .
   .
   .
   # Statement-N
```

Creating an Empty Class in Python

In the above example, we have created a class named Dog using the class keyword.

```
# Python3 program to
# demonstrate defining
# a class

class Dog:
    pass
```

Python Objects

The object is an entity that has a state and behavior associated with it. It may be any real-world object like a mouse, keyboard, chair, table, pen, etc. Integers, strings, floating-point numbers, even arrays, and dictionaries, are all objects. More specifically, any single integer or any single string is an object. The number 12 is an object, the string "Hello, world" is an object, a list is an object that can hold other objects, and so on. You've been using objects all along and may not even realize it.

An object consists of:

- **State:** It is represented by the attributes of an object. It also reflects the properties of an object.
- **Behavior:** It is represented by the methods of an object. It also reflects the response of an object to other objects.
- **Identity:** It gives a unique name to an object and enables one object to interact with other objects.

To understand the state, behavior, and identity let us take the example of the class dog (explained above).

- The identity can be considered as the name of the dog.
- State or Attributes can be considered as the breed, age, or color of the dog.
- The behavior can be considered as to whether the dog is eating or sleeping.

Creating an Object

This will create an object named obj of the class Dog defined above. Before diving deep into objects and classes let us understand some basic keywords that will we used while working with objects and classes.

```
obj = Dog()
```

The Python self

1. Class methods must have an extra first parameter in the method definition. We do not give a value for this parameter when we call the method, Python provides it
2. If we have a method that takes no arguments, then we still have to have one argument.

3. This is similar to this pointer in C++ and this reference in Java.

When we call a method of this object as myobject.method(arg1, arg2), this is automatically converted by Python into MyClass.method(myobject, arg1, arg2) – this is all the special self is about.

The Python __init__ Method

The __init__ method is similar to constructors in C++ and Java. It is run as soon as an object of a class is instantiated. The method is useful to do any initialization you want to do with your object. Now let us define a class and create some objects using the self and __init__ method.

Creating a class and object with class and instance attributes

```python
class Dog:

    # class attribute
    attr1 = "mammal"

    # Instance attribute
    def __init__(self, name):
        self.name = name

# Driver code
# Object instantiation
Rodger = Dog("Rodger")
Tommy = Dog("Tommy")

# Accessing class attributes
print("Rodger is a {}".format(Rodger.__class__.attr1))
print("Tommy is also a {}".format(Tommy.__class__.attr1))

# Accessing instance attributes
print("My name is {}".format(Rodger.name))
print("My name is {}".format(Tommy.name))
```

Output

```
Rodger is a mammal

Tommy is also a mammal

My name is Rodger

My name is Tommy
```

Creating Classes and objects with methods
Here, The Dog class is defined with two attributes:

- attr1 is a class attribute set to the value “mammal”. Class attributes are shared by all instances of the class.
- __init__ is a special method (constructor) that initializes an instance of the Dog class. It takes two parameters: self (referring to the instance being created) and name (representing the name of the dog). The name parameter is used to assign a name attribute to each instance of Dog. The speak method is defined within the Dog class. This method prints a string that includes the name of the dog instance.

The driver code starts by creating two instances of the Dog class: Rodger and Tommy. The __init__ method is called for each instance to initialize their name attributes with the provided names. The speak method is called in both instances (Rodger.speak() and Tommy.speak()), causing each dog to print a statement with its name.

- Python3

```python
class Dog:

    # class attribute
    attr1 = "mammal"

    # Instance attribute
    def __init__(self, name):
        self.name = name

    def speak(self):
        print("My name is {}".format(self.name))

# Driver code
# Object instantiation
Rodger = Dog("Rodger")
Tommy = Dog("Tommy")

# Accessing class methods
Rodger.speak()
Tommy.speak()
```

Output

```
My name is Rodger

My name is Tommy
```

Python Inheritance

Inheritance is the capability of one class to derive or inherit the properties from another class. The class that derives properties is called the derived class or child class and the class from which the properties are being derived is called the base class or parent class. The benefits of inheritance are:

- It represents real-world relationships well.
- It provides the reusability of a code. We don’t have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
- It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

Types of Inheritance

- **Single Inheritance**: Single-level inheritance enables a derived class to inherit characteristics from a single-parent class.

- **Multilevel Inheritance:** Multi-level inheritance enables a derived class to inherit properties from an immediate parent class which in turn inherits properties from his parent class.
- **Hierarchical Inheritance:** Hierarchical-level inheritance enables more than one derived class to inherit properties from a parent class.
- **Multiple Inheritance:** Multiple-level inheritance enables one derived class to inherit properties from more than one base class.

Inheritance in Python

In the above article, we have created two classes i.e. Person (parent class) and Employee (Child Class). The Employee class inherits from the Person class. We can use the methods of the person class through the employee class as seen in the display function in the above code. A child class can also modify the behavior of the parent class as seen through the details() method.

- Python3

```
# Python code to demonstrate how parent constructors
# are called.

# parent class
class Person(object):

    # __init__ is known as the constructor
    def __init__(self, name, idnumber):
        self.name = name
        self.idnumber = idnumber

    def display(self):
        print(self.name)
        print(self.idnumber)

    def details(self):
        print("My name is {}".format(self.name))
        print("IdNumber: {}".format(self.idnumber))

# child class
class Employee(Person):
    def __init__(self, name, idnumber, salary, post):
        self.salary = salary
        self.post = post

        # invoking the __init__ of the parent class
        Person.__init__(self, name, idnumber)

    def details(self):
        print("My name is {}".format(self.name))
        print("IdNumber: {}".format(self.idnumber))
        print("Post: {}".format(self.post))


# creation of an object variable or an instance
a = Employee('Rahul', 886012, 200000, "Intern")

# calling a function of the class Person using
# its instance
a.display()
a.details()
```

Output

```
Rahul

886012

My name is Rahul

IdNumber: 886012

Post: Intern
```

Python Polymorphism
Polymorphism simply means having many forms. For example, we need to determine if the given species of birds fly or not, using polymorphism we can do this using a single function.
Polymorphism in Python
This code demonstrates the concept of inheritance and method overriding in Python classes. It shows how subclasses can override methods defined in their parent class to provide specific behavior while still inheriting other methods from the parent class.

```
class Bird:

    def intro(self):
        print("There are many types of birds.")

    def flight(self):
        print("Most of the birds can fly but some cannot.")

class sparrow(Bird):

    def flight(self):
        print("Sparrows can fly.")

class ostrich(Bird):

    def flight(self):
        print("Ostriches cannot fly.")

obj_bird = Bird()
obj_spr = sparrow()
obj_ost = ostrich()

obj_bird.intro()
obj_bird.flight()

obj_spr.intro()
obj_spr.flight()
```

```
obj_ost.intro()
obj_ost.flight()
```

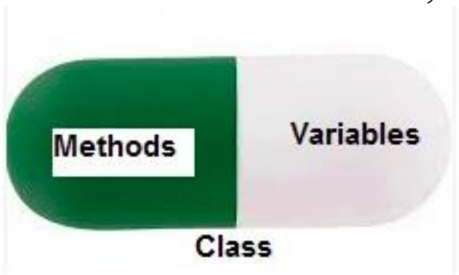
Output

```
There are many types of birds.

Most of the birds can fly but some cannot.

There are many types of birds.

Sparrows can fly.

There are many types of birds.

Ostriches cannot fly.
```

Python Encapsulation

Encapsulation is one of the fundamental concepts in object-oriented programming (OOP). It describes the idea of wrapping data and the methods that work on data within one unit. This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data. To prevent accidental change, an object's variable can only be changed by an object's method. Those types of variables are known as private variables.

A class is an example of encapsulation as it encapsulates all the data that is member functions, variables, etc.



Encapsulation in Python

In the above example, we have created the c variable as the private attribute. We cannot even access this attribute directly and can't even change its value.

```
# Python program to
# demonstrate private members
# "__" double underscore represents private attribute.
# Private attributes start with "__".

# Creating a Base class
class Base:
    def __init__(self):
        self.a = "GeeksforGeeks"
        self.__c = "GeeksforGeeks"

# Creating a derived class
class Derived(Base):
    def __init__(self):

        # Calling constructor of
        # Base class
        Base.__init__(self)
        print("Calling private member of base class: ")
        print(self.__c)


# Driver code
obj1 = Base()
print(obj1.a)

# Uncommenting print(obj1.c) will
# raise an AttributeError

# Uncommenting obj2 = Derived() will
# also raise an AtrributeError as
# private member of base class
# is called inside derived class
```

Output

```
GeeksforGeeks
```

Data Abstraction
It hides unnecessary code details from the user. Also, when we do not want to give out sensitive parts of our code implementation and this is where data abstraction came.
Data Abstraction in Python can be achieved by creating abstract classes.

Python Exception Handling

How to handle exceptions in Python using try, except, and finally statements with the help of proper examples.
Error in Python can be of two types i.e. Syntax errors and Exceptions. Errors are problems in a program due to which the program will stop the execution. On the other hand, exceptions are raised when some internal events occur which change the normal flow of the program.

Different types of exceptions in python:

In Python, there are several built-in Python exceptions that can be raised when an error occurs during the execution of a program. Here are some of the most common types of exceptions in Python:

- **SyntaxError:** This exception is raised when the interpreter encounters a syntax error in the code, such as a misspelled keyword, a missing colon, or an unbalanced parenthesis.
- **TypeError**: This exception is raised when an operation or function is applied to an object of the wrong type, such as adding a string to an integer.
- **NameError**: This exception is raised when a variable or function name is not found in the current scope.
- **IndexError**: This exception is raised when an index is out of range for a list, tuple, or other sequence types.
- **KeyError**: This exception is raised when a key is not found in a dictionary.
- **ValueError**: This exception is raised when a function or method is called with an invalid argument or input, such as trying to convert a string to an integer when the string does not represent a valid integer.

- **AttributeError**: This exception is raised when an attribute or method is not found on an object, such as trying to access a non-existent attribute of a class instance.
- **IOError**: This exception is raised when an I/O operation, such as reading or writing a file, fails due to an input/output error.
- **ZeroDivisionError**: This exception is raised when an attempt is made to divide a number by zero.
- **ImportError**: This exception is raised when an import statement fails to find or load a module

These are just a few examples of the many types of exceptions that can occur in Python. It's important to handle exceptions properly in your code using try-except blocks or other error-handling techniques, in order to gracefully handle errors and prevent the program from crashing.

Difference between Syntax Error and Exceptions

Syntax Error: As the name suggests this error is caused by the wrong syntax in the code. It leads to the termination of the program.

Example:

There is a syntax error in the code . The **'if'** statement should be followed by a colon (:), and the **'print'** statement should be indented to be inside the **'if'** block.

```
amount = 10000
if(amount > 2999)
print("You are eligible to purchase Dsa Self Paced")
```

Output:

```
  File "/home/ac35380186f4ca7978956ff46697139b.py", line 4
    if(amount>2999)
                  ^
SyntaxError: invalid syntax
```

Exceptions: Exceptions are raised when the program is syntactically correct, but the code results in an error. This error does not stop the execution of the program, however, it changes the normal flow of the program.

Example:
Here in this code a s we are dividing the **'marks'** by zero so a error will occur known as **'ZeroDivisionError'**

```
marks = 10000
a = marks / 0
print(a)
```

Output:

```
Traceback (most recent call last):
  File "/home/f3ad05420ab851d4bd106ffb04229907.py", line 4, in <module>
    a=marks/0
ZeroDivisionError: division by zero
```

In the above example raised the ZeroDivisionError as we are trying to divide a number by 0.

Example:
1) TypeError: This exception is raised when an operation or function is applied to an object of the wrong type. Here's an example:
Here a **'TypeError'** is raised as both the datatypes are different which are being added.

```
x = 5
y = "hello"
z = x + y
```

output:

```
Traceback (most recent call last):
  File "7edfa469-9a3c-4e4d-98f3-5544e60bff4e.py", line 4, in <module>
    z = x + y
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

try catch block to resolve it:

The code attempts to add an integer (**'x'**) and a string (**'y'**) together, which is not a valid operation, and it will raise a **'TypeError'**. The code used a **'try'** and **'except'** block to catch this exception and print an error message.

```
x = 5
y = "hello"
try:
    z = x + y
except TypeError:
    print("Error: cannot add an int and a str")
```

Output

```
Error: cannot add an int and a str
```

Try and Except Statement – Catching Exceptions

Try and except statements are used to catch and handle exceptions in Python. Statements that can raise exceptions are kept inside the try clause and the statements that handle the exception are written inside except clause.

Example: Here we are trying to access the array element whose index is out of bound and handle the corresponding exception.

```
a = [1, 2, 3]
try:
    print ("Second element = %d" %(a[1]))

    print ("Fourth element = %d" %(a[3]))

except:
    print ("An error occurred")
```

Output

```
Second element = 2

An error occurred
```

In the above example, the statements that can cause the error are placed inside the try statement (second print statement in our case). The second print statement tries to access the fourth element of the list which is not there and this throws an exception. This exception is then caught by the except statement.

Catching Specific Exception

A try statement can have more than one except clause, to specify handlers for different exceptions. Please note that at most one handler will be executed. For example, we can add IndexError in the above code. The general syntax for adding specific exceptions are –

```
try:
    # statement(s)
except IndexError:
    # statement(s)
except ValueError:
    # statement(s)
```

Example: Catching specific exceptions in the Python
The code defines a function **‘fun(a)'** that calculates b based on the input a. If a is less than 4, it attempts a division by zero, causing a **‘ZeroDivisionError'**. The code calls fun(3) and fun(5) inside a try-except block. It handles the ZeroDivisionError for fun(3) and prints **“ZeroDivisionError Occurred and Handled.”** The **‘NameError'** block is not executed since there are no **‘NameError'** exceptions in the code.

```python
def fun(a):
    if a < 4:

        b = a/(a-3)
    print("Value of b = ", b)

try:
    fun(3)
    fun(5)
except ZeroDivisionError:
    print("ZeroDivisionError Occurred and Handled")
except NameError:
    print("NameError Occurred and Handled")
```

Output

```
ZeroDivisionError Occurred and Handled
```

If you comment on the line fun(3), the output will be

```
NameError Occurred and Handled
```

The output above is so because as soon as python tries to access the value of b, NameError occurs.

Try with Else Clause

In Python, you can also use the else clause on the try-except block which must be present after all the except clauses. The code enters the else block only if the try clause does not raise an exception.

Try with else clause

The code defines a function **AbyB(a, b)** that calculates c as ((a+b) / (a-b)) and handles a potential ZeroDivisionError. It prints the result if there's no division by zero error.

Calling **AbyB(2.0, 3.0)** calculates and prints -5.0, while calling **AbyB(3.0, 3.0)** attempts to divide by zero, resulting in a **ZeroDivisionError**, which is caught and **"a/b results in 0"** is printed.

- Python3

```
def AbyB(a , b):
    try:
        c = ((a+b) / (a-b))
    except ZeroDivisionError:
        print ("a/b result in 0")
    else:
        print (c)
AbyB(2.0, 3.0)
AbyB(3.0, 3.0)
```

Output:

```
-5.0
a/b result in 0
```

Finally Keyword in Python

Python provides a keyword finally, which is always executed after the try and except blocks. The final block always executes after the normal termination of the try block or after the try block terminates due to some exception.

Syntax:

```
try:
    # Some Code....

except:
    # optional block
    # Handling of exception (if required)

else:
    # execute if no exception

finally:
    # Some code .....(always executed)
```

Example:

The code attempts to perform integer division by zero, resulting in a **ZeroDivisionError**. It catches the exception and prints **“Can’t divide by zero.”** Regardless of the exception, the finally block is executed and prints **“This is always executed.”**

```
try:
    k = 5//0
    print(k)

except ZeroDivisionError:
    print("Can't divide by zero")
```

```
finally:
    print('This is always executed')
```

Output:

```
Can't divide by zero
This is always executed
```

Raising Exception

The raise statement allows the programmer to force a specific exception to occur. The sole argument in raise indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from Exception).

This code intentionally raises a NameError with the message **"Hi there"** using the raise statement within a try block. Then, it catches the NameError exception, prints **"An exception,"** and re-raises the same exception using raise. This demonstrates how exceptions can be raised and handled in Python, allowing for custom error messages and further exception propagation.

```
try:
    raise NameError("Hi there")
except NameError:
    print ("An exception")
    raise
```

The output of the above code will simply line printed as "An exception" but a Runtime error will also occur in the last due to the raise statement in the last line. So, the output on your command line will look like

```
Traceback (most recent call last):
  File "/home/d6ec14ca595b97bff8d8034bbf212a9f.py", line 5, in <module>
```

```
    raise NameError("Hi there")  # Raise Error
NameError: Hi there
```
Advantages of Exception Handling:

- **Improved program reliability**: By handling exceptions properly, you can prevent your program from crashing or producing incorrect results due to unexpected errors or input.
- **Simplified error handling**: Exception handling allows you to separate error handling code from the main program logic, making it easier to read and maintain your code.
- **Cleaner code:** With exception handling, you can avoid using complex conditional statements to check for errors, leading to cleaner and more readable code.
- **Easier debugging**: When an exception is raised, the Python interpreter prints a traceback that shows the exact location where the exception occurred, making it easier to debug your code.

Disadvantages of Exception Handling:

- **Performance overhead:** Exception handling can be slower than using conditional statements to check for errors, as the interpreter has to perform additional work to catch and handle the exception.
- **Increased code complexity**: Exception handling can make your code more complex, especially if you have to handle multiple types of exceptions or implement complex error handling logic.
- **Possible security risks:** Improperly handled exceptions can potentially reveal sensitive information or create security vulnerabilities in your code, so it’s important to handle exceptions carefully and avoid exposing too much information about your program.

Regular Expression (RegEx) in Python with Examples

A **Regular Expression or RegEx** is a special sequence of characters that uses a search pattern to find a string or set of strings.
It can detect the presence or absence of a text by matching it with a particular pattern and also can split a pattern into one or more sub-patterns.

Regex Module in Python
Python has a built-in module named "**re**" that is used for regular expressions in Python. We can import this module by using the import statement.
Example: Importing re module in Python

```
# importing re module
import re
```

How to Use RegEx in Python?
You can use RegEx in Python after importing re module.

Example:
This Python code uses regular expressions to search for the word **"portal"** in the given string and then prints the start and end indices of the matched word within the string.

```
import re

s = 'GeeksforGeeks: A computer science portal for geeks'

match = re.search(r'portal', s)

print('Start Index:', match.start())
print('End Index:', match.end())
```

Output

```
Start Index: 34

End Index: 40
```

Note: Here r character (r’ portal’) stands for raw, not regex. The raw string is slightly different from a regular string, it won’t interpret the \ character as an escape character. This is because the regular expression engine uses \ character for its own escaping purpose.
Before starting with the Python regex module let’s see how to actually write regex using metacharacters or special sequences.

Metacharacters
Metacharacters are the characters with special meaning.

To understand the RE analogy, Metacharacters are useful and important. They will be used in functions of module re. Below is the list of metacharacters.

MetaCharacters	Description
\	Used to drop the special meaning of character following it
[]	Represent a character class
^	Matches the beginning
$	Matches the end
.	Matches any character except newline
|	Means OR (Matches with any of the characters separated by it.
?	Matches zero or one occurrence

MetaCharacters	Description
*	Any number of occurrences (including 0 occurrences)
+	One or more occurrences
{}	Indicate the number of occurrences of a preceding regex to match.
()	Enclose a group of Regex

Let’s discuss each of these metacharacters in detail:

1. \ – Backslash

The backslash (\) makes sure that the character is not treated in a special way. This can be considered a way of escaping metacharacters.

For example, if you want to search for the dot(.) in the string then you will find that dot(.) will be treated as a special character as is one of the metacharacters (as shown in the above table). So for this case, we will use the backslash(\) just before the dot(.) so that it will lose its specialty. See the below example for a better understanding.

Example:
The first search **(re.search(r'.', s))** matches any character, not just the period, while the second search **(re.search(r'\.', s))** specifically looks for and matches the period character.

```
import re

s = 'geeks.forgeeks'
```

```
# without using \
match = re.search(r'.', s)
print(match)

# using \
match = re.search(r'\.', s)
print(match)
```

Output

```
<re.Match object; span=(0, 1), match='g'>
<re.Match object; span=(5, 6), match='.'>
```

2. [] – Square Brackets

Square Brackets ([]) represent a character class consisting of a set of characters that we wish to match. For example, the character class [abc] will match any single a, b, or c.

We can also specify a range of characters using – inside the square brackets. For example,

- [0, 3] is sample as [0123]
- [a-c] is same as [abc]

We can also invert the character class using the caret(^) symbol. For example,

- [^0-3] means any number except 0, 1, 2, or 3
- [^a-c] means any character except a, b, or c

Example:
In this code, you're using regular expressions to find all the characters in the string that fall within the range of 'a' to 'm'.
The **re.findall()** function returns a list of all such characters. In the given string, the characters that match this pattern are: 'c', 'k', 'b', 'f', 'j', 'e', 'h', 'l', 'd', 'g'.

```
import re

string = "The quick brown fox jumps over the lazy dog"
pattern = "[a-m]"
result = re.findall(pattern, string)

print(result)
```

Output

```
['h', 'e', 'i', 'c', 'k', 'b', 'f', 'j', 'm', 'e', 'h', 'e', 'l', 'a', 'd', 'g']
```

3. ^ – Caret

Caret (^) symbol matches the beginning of the string i.e. checks whether the string starts with the given character(s) or not. For example –

- ^g will check if the string starts with g such as geeks, globe, girl, g, etc.
- ^ge will check if the string starts with ge such as geeks, geeksforgeeks, etc.

Example:
This code uses regular expressions to check if a list of strings starts with **"The"**. If a string begins with **"The," it's marked as "Matched"** otherwise, it's labeled as **"Not matched".**

```
import re
regex = r'^The'
strings = ['The quick brown fox', 'The lazy dog', 'A quick brown fox']
for string in strings:
    if re.match(regex, string):
        print(f'Matched: {string}')
    else:
        print(f'Not matched: {string}')
```

Output

```
Matched: The quick brown fox

Matched: The lazy dog

Not matched: A quick brown fox
```

4. $ – Dollar

Dollar($) symbol matches the end of the string i.e checks whether the string ends with the given character(s) or not. For example-

- s$ will check for the string that ends with a such as geeks, ends, s, etc.
- ks$ will check for the string that ends with ks such as geeks, geeksforgeeks, ks, etc.

Example:
This code uses a regular expression to check if the string ends with **“World!”.** If a match is found, it prints **“Match found!”** otherwise, it prints **“Match not found”**.

```python
import re

string = "Hello World!"
pattern = r"World!$"

match = re.search(pattern, string)
if match:
    print("Match found!")
else:
    print("Match not found.")
```

Output

```
Match found!
```

5. . – Dot

Dot(.) symbol matches only a single character except for the newline character (\n). For example –

- a.b will check for the string that contains any character at the place of the dot such as acb, acbd, abbb, etc
- .. will check if the string contains at least 2 characters

Example:
This code uses a regular expression to search for the pattern **"brown.fox"** within the string. The dot (.) in the pattern represents any character. If a match is found, it prints **"Match found!"** otherwise, it prints **"Match not found"**.

```python
import re

string = "The quick brown fox jumps over the lazy dog."
pattern = r"brown.fox"

match = re.search(pattern, string)
if match:
    print("Match found!")
else:
    print("Match not found.")
```

Output

```
Match found!
```

6. | – Or

Or symbol works as the or operator meaning it checks whether the pattern before or after the or symbol is present in the string or not. For example –

- a|b will match any string that contains a or b such as acd, bcd, abcd, etc.

7. ? – Question Mark

The question mark (?) is a quantifier in regular expressions that indicates that the preceding element should be matched zero or one time. It allows you to specify that the element is optional, meaning it may occur once or not at all. For example,

- ab?c will be matched for the string ac, acb, dabc but will not be matched for abbc because there are two b. Similarly, it will not be matched for abdc because b is not followed by c.

8.* – Star

Star (*) symbol matches zero or more occurrences of the regex preceding the * symbol. For example –

- ab*c will be matched for the string ac, abc, abbbc, dabc, etc. but will not be matched for abdc because b is not followed by c.

9. + – Plus

Plus (+) symbol matches one or more occurrences of the regex preceding the + symbol. For example –

- ab+c will be matched for the string abc, abbc, dabc, but will not be matched for ac, abdc, because there is no b in ac and b, is not followed by c in abdc.

10. {m, n} – Braces

Braces match any repetitions preceding regex from m to n both inclusive. For example –

- a{2, 4} will be matched for the string aaab, baaaac, gaad, but will not be matched for strings like abc, bc because there is only one a or no a in both the cases.

11. (<regex>) – Group

Group symbol is used to group sub-patterns. For example –

- (a|b)cd will match for strings like acd, abcd, gacd, etc.

Special Sequences
Special sequences do not match for the actual character in the string instead it tells the specific location in the search string where the match must occur. It makes it easier to write commonly used patterns.

List of special sequences

Special Sequence	Description	Examples	
\A	Matches if the string begins with the given character	\Afor	for geeks
			for the world
\b	Matches if the word begins or ends with the given character. \b(string) will check for the beginning of the word and (string)\b will check for the ending of the word.	\bge	geeks
			get

Special Sequence	Description	Examples	
\B	It is the opposite of the \b i.e. the string should not start or end with the given regex.	\Bge	together
			forge
\d	Matches any decimal digit, this is equivalent to the set class [0-9]	\d	123
			gee1
\D	Matches any non-digit character, this is equivalent to the set class [^0-9]	\D	geeks
			geek1
\s	Matches any whitespace character.	\s	gee ks
			a bc a
\S	Matches any non-whitespace character	\S	a bd
			abcd

Special Sequence	Description	Examples	
\w	Matches any alphanumeric character, this is equivalent to the class [a-zA-Z0-9_].	\w	123
			geeKs4
\W	Matches any non-alphanumeric character.	\W	>$
			gee<>
\Z	Matches if the string ends with the given regex	ab\Z	abcdab
			abababab

RegEx Functions

re module contains many functions that help us to search a string for a match.

Let's see various functions provided by this module to work with regex in Python.

Function	Description
re.findall()	finds and returns all matching occurrences in a list
re.compile()	Regular expressions are compiled into pattern objects

Function	Description
re.split()	Split string by the occurrences of a character or a pattern.
re.sub()	Replaces all occurrences of a character or patter with a replacement string.
re.escape()	Escapes special character
re.search()	Searches for first occurrence of character or pattern

Let's see the working of these RegEx functions with definition and examples:

1. re.findall()
Return all non-overlapping matches of pattern in string, as a list of strings. The string is scanned left-to-right, and matches are returned in the order found.

Finding all occurrences of a pattern
This code uses a regular expression **(\d+)** to find all the sequences of one or more digits in the given string. It searches for numeric values and stores them in a list. In this example, it finds and prints the numbers **"123456789"** and **"987654321"** from the input string.

```
import re
string = """Hello my Number is 123456789 and
            my friend's number is 987654321"""
regex = '\d+'

match = re.findall(regex, string)
print(match)
```

Output

```
['123456789', '987654321']
```

2. re.compile()
Regular expressions are compiled into pattern objects, which have methods for various operations such as searching for pattern matches or performing string substitutions.

Example 1:
The code uses a regular expression pattern **[a-e]** to find and list all lowercase letters from ‘a’ to ‘e’ in the input string **“Aye, said Mr. Gibenson Stark”.** The output will be **['e', 'a', 'd', 'b', 'e'],** which are the matching characters.

```python
import re
p = re.compile('[a-e]')

print(p.findall("Aye, said Mr. Gibenson Stark"))
```

Output

```
['e', 'a', 'd', 'b', 'e', 'a']
```

Understanding the Output:

- First occurrence is ‘e’ in “Aye” and not ‘A’, as it is Case Sensitive.
- Next Occurrence is ‘a’ in “said”, then ‘d’ in “said”, followed by ‘b’ and ‘e’ in “Gibenson”, the Last ‘a’ matches with “Stark”.
- Metacharacter backslash ‘\’ has a very important role as it signals various sequences. If the backslash is to be used without its special meaning as metacharacter, use’\\’

Example 2: Set class [\s,.] will match any whitespace character, ‘,’, or, ‘.’ .
The code uses regular expressions to find and list all single digits and sequences of digits in the given input strings. It finds single digits with **\d** and sequences of digits with **\d+**.

```
import re
p = re.compile('\d')
print(p.findall("I went to him at 11 A.M. on 4th July 1886"))

p = re.compile('\d+')
print(p.findall("I went to him at 11 A.M. on 4th July 1886"))
```

Output

```
['1', '1', '4', '1', '8', '8', '6']

['11', '4', '1886']
```

Example 3:
The code uses regular expressions to find and list word characters, sequences of word characters, and non-word characters in input strings. It provides lists of the matched characters or sequences.

```
import re

p = re.compile('\w')
print(p.findall("He said * in some_lang."))

p = re.compile('\w+')
print(p.findall("I went to him at 11 A.M., he \
said *** in some_language."))

p = re.compile('\W')
print(p.findall("he said *** in some_language."))
```

Output

```
['H', 'e', 's', 'a', 'i', 'd', 'i', 'n', 's', 'o', 'm', 'e', '_', 'l', 'a', 'n', 'g']

['I', 'went', 'to', 'him', 'at', '11', 'A', 'M', 'he', 'said', 'in', 'some_language']

[' ', ' ', '*', '*', '*', ' ...
```

Example 4:
The code uses a regular expression pattern ‘ab*’ to find and list all occurrences of ‘ab’ followed by zero or more ‘b’ characters in the input string “ababbaabbb”. It returns the following list of matches: [‘ab’, ‘abb’, ‘abbb’].

```
import re
p = re.compile('ab*')
print(p.findall("ababbaabbb"))
```

Output

```
['ab', 'abb', 'a', 'abbb']
```

Understanding the Output:

- Our RE is ab*, which ‘a’ accompanied by any no. of ‘b’s, starting from 0.
- Output ‘ab’, is valid because of single ‘a’ accompanied by single ‘b’.
- Output ‘abb’, is valid because of single ‘a’ accompanied by 2 ‘b’.
- Output ‘a’, is valid because of single ‘a’ accompanied by 0 ‘b’.
- Output ‘abbb’, is valid because of single ‘a’ accompanied by 3 ‘b’.

3. re.split()
Split string by the occurrences of a character or a pattern, upon finding that pattern, the remaining characters from the string are returned as part of the resulting list.

Syntax :
re.split(pattern, string, maxsplit=0, flags=0)

The First parameter, pattern denotes the regular expression, string is the given string in which pattern will be searched for and in which splitting occurs, maxsplit if not provided is considered to be zero ‘0’, and if any nonzero value is provided, then at most that many splits occur. If maxsplit = 1, then the string will split once only, resulting in a list of length 2. The flags are very useful and can help to shorten code, they are not necessary parameters, eg: flags = re.IGNORECASE, in this split, the case, i.e. the lowercase or the uppercase will be ignored.

Example 1:
Splits a string using non-word characters and spaces as delimiters, returning words: **['Words', 'words', 'Words']**. Considers apostrophes as non-word characters: **['Word', 's', 'words', 'Words']**. Splits using non-word characters and digits: **['On', '12th', 'Jan', '2016', 'at', '11', '02', 'AM']**. Splits using digits as the delimiter: **['On ', 'th Jan ', ', at ', ':', ' AM']**.

```
from re import split

print(split('\W+', 'Words, words , Words'))
print(split('\W+', "Word's words Words"))
print(split('\W+', 'On 12th Jan 2016, at 11:02 AM'))
print(split('\d+', 'On 12th Jan 2016, at 11:02 AM'))
```

Output

```
['Words', 'words', 'Words']

['Word', 's', 'words', 'Words']

['On', '12th', 'Jan', '2016', 'at', '11', '02', 'AM']

['On ', 'th Jan ', ', at ', ':', ' AM']
```

Example 2:
First statement splits the string at the first occurrence of one or more digits: **['On ', 'th Jan 2016, at 11:02 AM']**. second splits the string using lowercase letters a to f as delimiters, case-insensitive: **['', 'y, ', 'oy oh ', 'oy, ', 'ome here'**]. Third splits the string using lowercase letters a to f as delimiters, case-sensitive: **['', 'ey, Boy oh ', 'oy, ', 'ome here']**.

```
import re
print(re.split('\d+', 'On 12th Jan 2016, at 11:02 AM', 1))
print(re.split('[a-f]+', 'Aey, Boy oh boy, come here', flags=re.IGNORECASE))
print(re.split('[a-f]+', 'Aey, Boy oh boy, come here'))
```

Output

```
['On ', 'th Jan 2016, at 11:02 AM']

['', 'y, ', 'oy oh ', 'oy, ', 'om', ' h', 'r', '']

['A', 'y, Boy oh ', 'oy, ', 'om', ' h', 'r', '']
```

4. re.sub()
The ‘sub’ in the function stands for SubString, a certain regular expression pattern is searched in the given string(3rd parameter), and upon finding the substring pattern is replaced by repl(2nd parameter), count checks and maintains the number of times this occurs.

Syntax:
re.sub(pattern, repl, string, count=0, flags=0)

Example 1:

- First statement replaces all occurrences of ‘ub’ with ‘~*’ (case-insensitive): **'S~*ject has ~*er booked already'**.
- Second statement replaces all occurrences of ‘ub’ with ‘~*’ (case-sensitive): **'S~*ject has Uber booked already'**.
- Third statement replaces the first occurrence of ‘ub’ with ‘~*’ (case-insensitive): **'S~*ject has Uber booked already'**.

- Fourth replaces ‘AND’ with ‘ & ‘ (case-insensitive): **'Baked Beans & Spam'.**

```
import re
print(re.sub('ub', '~*', 'Subject has Uber booked already',
             flags=re.IGNORECASE))
print(re.sub('ub', '~*', 'Subject has Uber booked already'))
print(re.sub('ub', '~*', 'Subject has Uber booked already',
             count=1, flags=re.IGNORECASE))
print(re.sub(r'\sAND\s', ' & ', 'Baked Beans And Spam',
             flags=re.IGNORECASE))
```

Output

```
S~*ject has ~*er booked already

S~*ject has Uber booked already

S~*ject has Uber booked already

Baked Beans & Spam
```

5. re.subn()

subn() is similar to sub() in all ways, except in its way of providing output. It returns a tuple with a count of the total of replacement and the new string rather than just the string.

Syntax:

re.subn(pattern, repl, string, count=0, flags=0)

Example:

re.subn() replaces all occurrences of a pattern in a string and returns a tuple with the modified string and the count of substitutions made. It’s useful for both case-sensitive and case-insensitive substitutions.

```
import re

print(re.subn('ub', '~*', 'Subject has Uber booked already'))
```

```
t = re.subn('ub', '~*', 'Subject has Uber booked already',
            flags=re.IGNORECASE)
print(t)
print(len(t))
print(t[0])
```

Output

```
('S~*ject has Uber booked already', 1)

('S~*ject has ~*er booked already', 2)

2

S~*ject has ~*er booked already
```

6. re.escape()
Returns string with all non-alphanumerics backslashed, this is useful if you want to match an arbitrary literal string that may have regular expression metacharacters in it.

Syntax:
re.escape(string)

Example:
re.escape() is used to escape special characters in a string, making it safe to be used as a pattern in regular expressions. It ensures that any characters with special meanings in regular expressions are treated as literal characters.

- Python

```
import re
print(re.escape("This is Awesome even 1 AM"))
print(re.escape("I Asked what is this [a-9], he said \t ^WoW"))
```

Output

```
This\ is\ Awesome\ even\ 1\ AM

I\ Asked\ what\ is\ this\ \[a\-9\]\,\ he\ said\ \   \ \^WoW
```

7. re.search()

This method either returns None (if the pattern doesn’t match), or a re.MatchObject contains information about the matching part of the string. This method stops after the first match, so this is best suited for testing a regular expression more than extracting data.

Example: Searching for an occurrence of the pattern
This code uses a regular expression to search for a pattern in the given string. If a match is found, it extracts and prints the matched portions of the string.

In this specific example, it searches for a pattern that consists of a month (letters) followed by a day (digits) in the input string “I was born on June 24”. If a match is found, it prints the full match, the month, and the day.

```python
import re
regex = r"([a-zA-Z]+) (\d+)"

match = re.search(regex, "I was born on June 24")
if match != None:
    print ("Match at index %s, %s" % (match.start(), match.end()))
    print ("Full match: %s" % (match.group(0)))
    print ("Month: %s" % (match.group(1)))
    print ("Day: %s" % (match.group(2)))

else:
    print ("The regex pattern does not match.")
```

Output

```
Match at index 14, 21

Full match: June 24

Month: June

Day: 24
```