**VARIABLES:** Variables are containers for storing data values. In Python, creating a variable is as simple as assigning a value to it.

**Ex:** x = 5, y = “John”

Print(x) # 5

Print(y) # John

**Python** has no command for declaring a variable. A variable is created the moment we’ve assigned a value to it.

If we want to specify the type of variable. We can use casting.

**Ex:**

X = str(3) # x will be ‘3’

Y = int(3) # y will be 3

Z = float(3) # z will be 3.0

Print(type(x)) # <class ‘str’>

Print(type(y)) # <class ‘int’>

Print(type(z)) # <class ‘float’>

**Python** variables are case-sensitive

**Ex:**

a = 4

A = ‘Sally’

Print(a) # 4

Print(A) # Sally

* Here A will not overwrite a

**Variable Names** must start with a letter or underscore and cannot start with a number. Variable names must contain alphanumeric characters and underscores. Names are case-sensitive.

**Ex:** myvar = “John”, my\_var = “John”, \_my\_var = “John”, myVar = “John”, MYVAR = “John”, myvar2 = “John” -> All are legal variable names.

2myvar = “John”, my-var = “John”, my var = “John” -> All are illegal variable names

**In Python,** we can assign multiple values to a variable

**Ex:** x, y, z = “John”, “Wick”, “Smith”

Print(x) # John

Print(y) # Wick

Print(z) # Smith

**Unpacking a collection**

**Ex:**

Cities = [“Hyderabad”, “Pune”, “Kashmir”]

x, y, z = cities

print(x) # Hyderabad print(y) #Pune print(z) # Kashmir

**Assign multiple variables:**

X, y, z = “John”, “Wick”, “Smith”

**Assigning single value to multiple variables**

X = y = z = “John”

**Unpack a collection**

Cities = [“Hyderabad”, “Pune”, “Kashmir”]

**Global Variables:**

1. **Accessible Outside Functions**
2. **Accessible Inside Functions**

**Ex:**

x = “awesome” # variable is created outside the function, that’s why it is called as global variable. It can be used anywhere in our code.

def myfunc():

print(“Python is” + x)

myfunc() # Python is awesome

x = “awesome”

def myfunc():

x = “fantastic” # variable is created with the same name inside the function is a local variable, only accessible within myfun()

print(“Python is” + x)

myfunc() # Python is fantastic

print(“Python is” + x) # Python is awesome # global x remains unchanged.

**Create a global variable inside the function.**

**Ex:**

def myfunc():

global x

x = “fantastic”

myfunc()

print(“Python is” + x) # Python is fantastic

**DATATYPES:** Variables can store data of different types

Text Type: **str**

Numeric Types: **int, float, complex**

Sequence Types: **list, tuple, range**

Mapping Type: **dict**

Set Types: **set, frozenset**

Boolean Type: **bool**

Binary Types: **bytes, bytearray, memoryview**

None Type: **NoneType**

We can get the data type of any object using **type()** function.

|  |  |  |
| --- | --- | --- |
| **Example** | **DataType** | **Output** |
| x = "Hello World" | str | Print(x)  print(type(x))  Hello World <class 'str'> |
| x = 20 | int | Print(x)  print(type(x))  20 <class 'int'> |
| x = 20.5 | float | Print(x)  print(type(x))  20.5 <class 'float'> |
| x = 1j | complex | Print(x)  print(type(x))  lj <class 'complex'> |
| x = ["apple", "banana", "cherry"] | list | Print(x)  print(type(x))  ['apple', 'banana', 'cherry'] <class 'list'> |
| x = ("apple", "banana", "cherry") | tuple | Print(x)  print(type(x))  ('apple', 'banana', 'cherry') <class 'tuple'> |
| x = range(6) | range | Print(x)  print(type(x))  range(0, 6) <class 'range'> |
| x = {"name" : "John", "age" : 36} | dict | Print(x)  print(type(x))  {'name': 'John', 'age': 36} <class 'dict'> |
| x = {"apple", "banana", "cherry"} | set | Print(x)  print(type(x))  {'banana', 'apple', 'cherry'} <class 'set'> |
| x = frozenset({"apple", "banana", "cherry"}) | frozenset | Print(x)  print(type(x))  frozenset({'cherry', 'banana', 'apple'}) <class 'frozenset'> |
| x = True | bool | Print(x)  print(type(x))  True <class 'bool'> |
| x = b"Hello" | byte | Print(x)  print(type(x))  b'Hello' <class 'bytes'> |
| x = bytearray(5) | bytearray | Print(x)  print(type(x))  bytearray(b'\x00\x00\x00\x00\x00') <class 'bytearray'> |
| x = memoryview(bytes(5)) | memoryview | Print(x)  print(type(x))  <memory at 0x00D58FA0> <class 'memoryview'> |
| x = None | NoneType | Print(x)  print(type(x))  None  <class 'NoneType'> |

**OPERATORS:** Operators are used to perform operations on variables and values.

**Python** divides operators in the following groups:

1. Arithmetic Operators
2. Assignment Operators
3. Comparison Operators
4. Logical Operators
5. Identity Operators
6. Membership Operators
7. Bitwise Operators
8. **Arithmetic Operators:** Arithmetic operators are used with numeric values to perform common mathematic operations.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Name** | **Example** | **Output** |
| + | Addition | x + y | x=5, y=3  Print(x+y)  8 |
| - | Subtraction | x - y | x=5, y=3  Print(x-y)  2 |
| \* | Multiplication | x \* y | x=5, y=3  Print(x\*y)  15 |
| / | Division | x / y | x=12, y=3  Print(x/y)  4 |
| % | Modulus | x % y | x=5, y=2  Print(x%y)  1 |
| \*\* | Exponentiation | x \*\* y | x=2, y=5  Print(x\*\*y)  2\*2\*2\*2\*2=32 |
| // | Floor Division | x // y | x=15, y=2  Print(x//y)  #The floor division // rounds the result down to the nearest whole number  7 |

1. **Assignment Operators:** Assignment operators are used to assign values to variables.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Example** | **Same As** | **Output** |
| = | x = 5 | x = 5 | x = 5  print(x)  5 |
| += | x += 3 | x = x + 3 | x = 5  x += 3  print(x)  8 |
| -= | x -= 3 | x = x – 3 | x = 5  x -= 3  print(x)  2 |
| \*= | x \*= 3 | x = x \* 3 | x = 5  x \*= 3  print(x)  15 |
| /= | x /= 3 | x = x / 3 | x = 5  x /= 3  print(x)  1.6666666666666667 |
| %= | x %= 3 | x = x % 3 | x = 5  x %= 3  print(x)  2 |
| //= | x //= 3 | x = x // 3 | x = 5  x //= 3  print(x)  1 |
| \*\*= | x \*\*= 3 | x = x \*\* 3 | x = 5  x \*\*= 3  print(x)  125 |
| &= | x &= 3 | x = x & 3 | x = 5  x &= 3  print(x)  1 |
| |= | x |= 3 | x = x | 3 | x = 5  x |= 3  print(x)  7 |
| ^= | x ^= 3 | x = x ^ 3 | x = 5  x ^= 3  print(x)  6 |
| >>= | x >>= 3 | x = x >> 3 | x = 5  x >>= 3  print(x)  0 |
| <<= | x <<= 3 | x = x << 3 | x = 5  x <<= 3  print(x)  40 |
| := | x := 3 | x = x : 3 | print(x := 3)  3 |

1. **Comparison Operators:** Comparison operators are user to compare two values.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Name** | **Example** | **Output** |
| == | Equal | x == y | x = 5, y = 3  print(x==y)  False |
| != | Not Equal | x != y | x = 5, y = 3  print(x!=y)  True |
| > | Greater Than | x > y | x = 5, y = 3  print(x>y)  True |
| < | Less Than | x < y | x = 5, y = 3  print(x<y)  False |
| >= | Greater than or equal to | x >= y | x = 5, y = 3  print(x>=y)  True |
| <= | Less than or equal to | x <= y | x = 5, y = 3  print(x<=y)  False |

1. **Logical Operators:** Logical operators are used to combine conditional statements.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Description** | **Example** | **Output** |
| and | Returns true if both statements are true | x<5 and x<10 | x = 5  print(x>3 and x<10)  # returns True because 5 is greater than 3 AND 5 is less than 10  True |
| or | Returns true if one of the statements is true | x<5 or x<4 | x = 5  print(x>3 and x<4)  # returns True because one of the conditions are true (5 is greater than 3, but 5 is not less than 4)  True |
| not | Reverse the result, returns false if the result is false | not(x<5 and x<10) | x = 5  print(not(x > 3 and x < 10))  # returns False because not is used to reverse the result  False |

1. **Identity Operators:** Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Description** | **Example** | **Output** |
| is | Returns true if both variables are the same object | x is y | x = ["John", "Dev"]  y = ["John", "Dev"]  z = x  print(x is z) # True  # returns True because z is the same object as x  print(x is y) # False  # returns False because x is not the same object as y, even if they have the same content  print(x == y) # True  # to demonstrate the difference between "is" and "==": this comparison returns True because x is equal to y |
| is not | Returns true if both variables are not the same object | x is not y | x = ["John", "Dev"]  y = ["John", "Dev"]  z = x  print(x is not z) # False  # returns False because z is the same object as x  print(x is not y) # True  # returns True because x is not the same object as y, even if they have the same content  print(x != y) # False  # to demonstrate the difference between "is not" and "!=": this comparison returns False because x is equal to y |

1. **Membership Operators:** Membership operators are used to test if a sequence is present in an object.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Description** | **Example** | **Output** |
| in | Returns true if a sequence with the specified value is present in the object | x in y | x = [“John”, “Dev”]  print(“Dev” in y)  # returns True because a sequence with the value "Dev" is in the list  True |
| not in | Returns true if a sequence with the specified value is not present in the object | x not in y | x = [“John”, “Dev”]  print(“David” not in y)  # returns True because a sequence with the value "David" is in the list  True |

1. **Bitwise Operators:** Bitwise operators are used to compare (binary) numbers.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Operator** | **Name** | **Description** | **Example** | **Output** |
| & | AND | Sets each bit to 1 if both bits are 1 | x & y | print(6 & 3) # 2  # The & operator compares each bit and set it to 1 if both are 1, otherwise it is set to 0:  6 = 0000000000000110  3 = 0000000000000011  --------------------  2 = 0000000000000010  ====================  Decimal numbers and their binary values:  0 = 0000000000000000  1 = 0000000000000001  2 = 0000000000000010  3 = 0000000000000011  4 = 0000000000000100  5 = 0000000000000101  6 = 0000000000000110  7 = 0000000000000111 |
| | | OR | Sets each bit to 1 if one of two bits is 1 | x | y | print(6 | 3) # 7  # The | operator compares each bit and set it to 1 if one or both is 1, otherwise it is set to 0:  6 = 0000000000000110  3 = 0000000000000011  --------------------  7 = 0000000000000111  ====================  Decimal numbers and their binary values:  0 = 0000000000000000  1 = 0000000000000001  2 = 0000000000000010  3 = 0000000000000011  4 = 0000000000000100  5 = 0000000000000101  6 = 0000000000000110  7 = 0000000000000111 |
| ^ | XOR | Sets each bit to 1 if only one of two bits is 1 | x ^ y | print(6 ^ 3) # 5  # The ^ operator compares each bit and set it to 1 if only one is 1, otherwise (if both are 1 or both are 0) it is set to 0:  6 = 0000000000000110  3 = 0000000000000011  --------------------  5 = 0000000000000101  ====================  Decimal numbers and their binary values:  0 = 0000000000000000  1 = 0000000000000001  2 = 0000000000000010  3 = 0000000000000011  4 = 0000000000000100  5 = 0000000000000101  6 = 0000000000000110  7 = 0000000000000111 |
| ~ | NOT | Inverts all the bits | ~x | print(~3) # -4  # The ~ operator inverts each bit (0 becomes 1 and 1 becomes 0).  Inverted 3 becomes -4:  3 = 0000000000000011  -4 = 1111111111111100  Decimal numbers and their binary values:  4 = 0000000000000100  3 = 0000000000000011  2 = 0000000000000010  1 = 0000000000000001  0 = 0000000000000000  -1 = 1111111111111111  -2 = 1111111111111110  -3 = 1111111111111101  -4 = 1111111111111100 |
| << | Zero fill left shift | Shift left by pushing zeros in from the right and let the leftmost bits fall off | x << 2 | print(3 << 2) # 12  # The << operator inserts the specified number of 0's (in this case 2) from the right and let the same amount of leftmost bits fall off:  If you push 00 in from the left:  3 = 0000000000000011  becomes  12 = 0000000000001100  Decimal numbers and their binary values:  0 = 0000000000000000  1 = 0000000000000001  2 = 0000000000000010  3 = 0000000000000011  4 = 0000000000000100  5 = 0000000000000101  6 = 0000000000000110  7 = 0000000000000111  8 = 0000000000001000  9 = 0000000000001001  10 = 0000000000001010  11 = 0000000000001011  12 = 0000000000001100 |
| >> | Signed right shift | Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off | x >> 2 | print(8 >> 2) # 2  # The >> operator moves each bit the specified number of times to the right. Empty holes at the left are filled with 0's.  If you move each bit 2 times to the right, 8 becomes 2:  8 = 0000000000001000  becomes  2 = 0000000000000010  Decimal numbers and their binary values:  0 = 0000000000000000  1 = 0000000000000001  2 = 0000000000000010  3 = 0000000000000011  4 = 0000000000000100  5 = 0000000000000101  6 = 0000000000000110  7 = 0000000000000111  8 = 0000000000001000  9 = 0000000000001001  10 = 0000000000001010  11 = 0000000000001011  12 = 0000000000001100 |

**LOOPS:** Python has 2 primitive loop commands:

**while** loop

**for** loop

**The while loop:**

With the **while** loop, we can execute a set of statements as long as condition is true.

**Example:**  print i as long as i is less than 6

i = 1

while i<6:

print(i)

i += 1

**o/p:** 1 2 3 4 5

**The break statement:**

With the **break** statement we can stop the loop event if the while condition is true.

**Example:** Exit the loop when i is 3

i = 1

while i < 6:

print(i)

if i == 3:

break

i += 1

**o/p:** 1 2 3

**The continue statement:**

With the **continue** statement we can stop the current iteration, and can continue with the next.

**Example:** Continue to the next iteration if i is 3

i = 1

while i < 6:

i += 1

if i == 3:

continue

print(i)

**o/p:** 1 2 4 5 6

**The else statement:**

With the **else** statement we can run a block of code once when the condition is no longer is true.

**Example:** Print a message once a condition is false.

i = 1  
while i < 6:  
  print(i)  
  i += 1  
else:  
  print("i is no longer less than 6")

**The for loop:** A **for** loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set or a string).

With the **for** loop we can execute a set of statements, once for each item in a list, tuple, set, etc.

**Example:** Print each country in a countries list.

countries = [“India”, “Australia”, “England”]

for x in countries:

print(x) – **o/p:** India Australia England

**Looping through a string:** Even strings are iterable objects, they contain a sequence of characters.

**Example:** Loop through the letters in the word “India”

for x in “India”:

print(x)

**o/p:**

I

n

d

i

a

**The break statement:** With the **break** statement we can stop the loop before it has looped through all the items.

**Example:** Exit the loop when x is “England”

for x in countries:

if x == “England”:

break

print(x)

**o/p:** India Australia

**The continue statement:** With the **continue** statement we can stop the current iteration, and continue with the next.

**Example:** Do not print “Australia”

for x in countries:

if x == “Australia”:

continue

print(x)

**o/p:** India England

**The range function:** To loop through a set of code a specified number of times, we can use the range().

The range() returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default) and ends at a specified number.

**Example:** Using the range() function.

for x in range(6):

print(x)

**o/p:** 0 1 2 3 4 5

**Example:** Using the **start** parameter

for x in range(2, 6):

print(x)

**o/p:** 2 3 4 5

**Example:** Increment the **sequence** with 3 (default is 1)

for x in range(2, 30, 3):

print(x)

**o/p:** 2 5 8 11 14 17 20 23 26 29

**Else in for loop:** The **else** keyword in **for** loop specifies a block of code to be executed when the loop has ended.

**Example:** Print all numbers from 0 to 5, and print a message when the loop has ended.

for x in range(6):

print(x)

else:

print(“Finally finished!”)

**o/p:** 0 1 2 3 4 5 Finally finished!

**Example:** Break the loop when x is 3, and see what happens with the **else** block.

for x in range(6):

if (x == 3): break

else:

print(“Finaaly Finished!”)

**o/p:** 0 1 2

**Nested loops:** A nested loop is a loop inside a loop. The inner loop will be executed one time for each iteration of the outer loop.

**Example:** Print each adjective for every country.

adj = [“Great”, “Good”, “Nice”]

countries = [“India”, “Australia”, “England”]

for x in adj:

for y in countries:

print(x, y)

**o/p:**  Great India Great Australia Great England Good India Good Australia Good England Nice India Nice Australia Nice England

**The pass statement:** for loops cannot be empty, but if you for some reason have a **for** loop with no content, put the **pass** statement to avoid getting an error.

**Example:**

**f**or x in [0,1, 2]:

pass

**LISTS:** Lists are used to store multiple items in a single variable.

**Example:** Create a list.

thislist = [“John”, “Wright”, “England”]

print(thislist)

**o/p:** [‘John’, “Wright’, ‘England’]

**List Items:** List items are ordered, changeable and allow duplicate values.

List items are indexed, the first item has [0] index, the second item has [1] etc.

**Access Items:** We can access list items by referring to the index number.

**Example:** Print the second item of the list.

thislist = [“John”, “Wright”, “England”]

print(thislist[1])

**o/p:** Wright

**Navigate Indexing:** Navigate indexing means start from the end.

-1 refers to the last item, -2 refers to the second last item.

**Example:** Print the last item of the list.

thislist = [“John”, “Wright”, “England”]

print(thislist[-1])

**o/p:** England

**Range of indexes:** We can specify a range of indexes by specifying where to start and where to end the range.

When specifying the range, the return value will be new list with specified items.

**Example:** Return the third, fourth and fifth item.

products = ["Laptop", "Smartphone", "Headphones", "Camera", “Webcam”, “Keyboard”, “Mouse”]

print(products[2:5])

**o/p:** [‘Headphones’, ‘Camera’, ‘Webcam’]

By leaving out the start value, the range will start at the first item.

**Example:** Returns the item from the beginning to, but not including “Webcam”.

products = ["Laptop", "Smartphone", "Headphones", "Camera", “Webcam”, “Keyboard”, “Mouse”]

print(products[:4])

**o/p:** [‘Laptop’, ‘Samrtphone’, ‘Headphones’, ‘Camera’]

By leaving out the end value, the range will go on the end of the list.

**Example:** Return the items from “Headphones” to the end.

products = ["Laptop", "Smartphone", "Headphones", "Camera", “Webcam”, “Keyboard”, “Mouse”]

print(products[2:])

**o/p: [‘**Headphones’, ‘Camera’, ‘Webcam’, ‘Keyboard’, ‘Mouse’]

**Range of negative indexes:** Specify negative indexes if you want to start the search from the end of the list.

**Example:** Return the items from “Camera” (-4) to, but not including “Mouse” (-1)

products = ["Laptop", "Smartphone", "Headphones", "Camera", “Webcam”, “Keyboard”, “Mouse”]

print(products[-4:-1])

**o/p:** [‘camera’, ‘Webcam’, ‘Keyboard’]

**Check if item exists:** To determine if a specified item is present in a list use the **in** keyword.

**Example:** Check if ‘John’ is present in the list.

thislist = [“John”, “Wright”, “England”]

if “John” in thislist:

print(“Yes, ‘John’ is present in the list”)

**o/p:** Yes, ‘John’ is present in the li

**Ordered:** When we say the lists are ordered, it means the items have a defined order, and that order will not change.

If you add new items to a list, the new items will be placed at the end of the list.

**List Methods:**

|  |  |  |
| --- | --- | --- |
| **Method** | **Description** | **Example** |
| append() | Adds an element at the end of the list | fruits = ['apple', 'banana', 'cherry'] fruits.append("orange")  print(fruits)  ['apple', 'banana', 'cherry', 'orange'] |
| clear() | Removes all elements from the list | fruits = ['apple', 'banana', 'cherry', 'orange'] fruits.clear()  print(fruits)  [] |
| copy() | Returns a copy of the list | fruits = ['apple', 'banana', 'cherry', 'orange'] x = fruits.copy()  print(x)  ['apple', 'banana', 'cherry'] |
| count() | Return the number of elements with the specified value | fruits = ['apple', 'banana', 'cherry'] x = fruits.count("cherry")  print(x)  1 |
| extend() | Add the elements of a list to the end of the current list | fruits = ['apple', 'banana', 'cherry'] cars = ['Ford', 'BMW', 'Volvo'] fruits.extend(cars)  print(fruits)  ['apple', 'banana', 'cherry', 'Ford', 'BMW', 'Volvo'] |
| index() | Returns the index of the first element with specified value | fruits = ['apple', 'banana', 'cherry'] x = fruits.index("cherry")  print(x)  2 |
| insert() | Add an element at the specified position | fruits = ['apple', 'banana', 'cherry'] fruits.insert(1, "orange")  print(fruits)  ['apple', 'orange', 'banana', 'cherry'] |
| pop() | Removes the element at the specified position | fruits = ['apple', 'banana', 'cherry'] fruits.pop(1)  print(fruits)  ['apple', 'cherry'] |
| remove() | Removes the item with the specified value | fruits = ['apple', 'banana', 'cherry'] fruits.remove("banana")  print(fruits)  ['apple', 'cherry'] |
| reverse() | Reverses the order of the list | fruits = ['apple', 'banana', 'cherry'] fruits.reverse()  print(fruits)  ['cherry', 'banana', 'apple'] |
| sort() | Sorts the list | cars = ['Ford', 'BMW', 'Volvo'] cars.sort()  print(cars)  ['BMW', 'Ford', 'Volvo'] |

**Tuple Methods:**

|  |  |  |
| --- | --- | --- |
| **Method** | **Description** | **Example** |
| count() | Returns the number of times a specified value occurs in a tuple | thistuple = (1, 3, 7, 8, 7, 5, 4, 6, 8, 5) x = thistuple.count(5) print(x)  2 |
| index() | Searches the tuple for a specified value and returns the position of where it was found | Search for the first occurrence of the value 8, and return its position  thistuple = (1, 3, 7, 8, 7, 5, 4, 6, 8, 5) x = thistuple.index(8) print(x)  3 |

**Set Methods:**

|  |  |  |
| --- | --- | --- |
| **Method** | **Description** | **Example** |
| add() | Adds an element to the set | fruits = {"apple", "banana", "cherry"} fruits.add("orange") print(fruits)  {'cherry', 'orange', 'banana', 'apple'} |
| clear() | Removes all elements from the set | fruits = {"apple", "banana", "cherry"} fruits.clear() print(fruits)  set() |
| copy() | Returns a copy of the set | fruits = {"apple", "banana", "cherry"} x = fruits.copy() print(x)  {'cherry', 'apple', 'banana'} |
| difference() | Returns a set the difference between two or more sets | x = {"apple", "banana", "cherry"}  y = {"google", "microsoft", "apple"} z = x.difference(y) print(z)  {'banana', 'cherry'} |
| difference\_update() | Removes the items in this set that are also included in another, specified set | x = {"apple", "banana", "cherry"} y = {"google", "microsoft", "apple"} x.difference\_update(y) print(x)  {'cherry', 'banana'} |
| discard() | Removes the specified item | fruits = {"apple", "banana", "cherry"} fruits.discard("banana") print(fruits)  {'cherry', ‘apple’ } |
| intersection() | Returns a set, that is the intersection of two other sets | x = {"apple", "banana", "cherry"} y = {"google", "microsoft", "apple"} z = x.intersection(y) print(z)  {'apple'} |
| intersection\_update() | Removes the items in the set that are not present in other specified sets | x = {"apple", "banana", "cherry"} y = {"google", "microsoft", "apple"} x.intersection\_update(y) print(x)  {'apple'} |
| isdisjoint() | Returns whether two sets have an intersection or not | x = {"apple", "banana", "cherry"} y = {"google", "microsoft", "facebook"} z = x.isdisjoint(y) print(z) # True |
| issubset() | Returns whether another set contains this set or not  Returns whether all items in this set is present in another specified sets | x = {"a", "b", "c"} y = {"f", "e", "d", "c", "b", "a"} z = x.issubset(y) print(z)  True |
| issuperset() | Returns whether the set contains another set or not  Returns whether all items in other specified sets is present in this set | x = {"f", "e", "d", "c", "b", "a"} y = {"a", "b", "c"} z = x.issuperset(y) print(z)  True |
| pop() | Removes an element from the set  Remove a random item from the set | fruits = {"apple", "banana", "cherry"} fruits.pop() print(fruits)  {'banana', 'cherry'} |
| remove() | Removes the specified item | fruits = {"apple", "banana", "cherry"} fruits.remove("banana") print(fruits)  {'apple', 'cherry'} |
| systematic\_difference() | Returns a set with systematic difference of two sets  Return a set that contains all items from both sets, except items that are present in both sets | x = {"apple", "banana", "cherry"} y = {"google", "microsoft", "apple"} z = x.symmetric\_difference(y) print(z)  {'google', 'banana', 'microsoft', 'cherry'} |
| systematic\_difference\_update() | Inserts the symmetric differences from this set and another  Remove the items that are present in both sets, AND insert the items that is not present in both sets | x = {"apple", "banana", "cherry"} y = {"google", "microsoft", "apple"} x.symmetric\_difference\_update(y) print(x)  {'cherry', 'microsoft', 'google', 'banana'} |
| union() | Returns a set containing the union of sets  Return a set that contains all items from both sets, duplicates are excluded | x = {"apple", "banana", "cherry"} y = {"google", "microsoft", "apple"} z = x.union(y) print(z)  {'microsoft', 'cherry', 'apple', 'google', 'banana'} |
| update() | Update the set with the union of this set and others | x = {"apple", "banana", "cherry"} y = {"google", "microsoft", "apple"} x.update(y) print(x)  {'banana', 'microsoft', 'cherry', 'google', 'apple'} |

**Dictionary Methods:**

|  |  |  |
| --- | --- | --- |
| **Method** | **Description** | **Example** |
| clear() | Removes all the elements from the dictionary | car = {  "brand": "Ford",  "model": "Mustang",  "year": 1964  }  car.clear()  print(car)  {} |
| copy() | Returns a copy of the dictionary | car = {  "brand": "Ford",  "model": "Mustang",  "year": 1964  }  x = car.copy()  print(x)  {'brand': 'Ford', 'model': 'Mustang', 'year': 1964} |
| fromkeys() | Returns a dictionary with the specified keys and value | x = ('key1', 'key2', 'key3')  y = 0  thisdict = dict.fromkeys(x, y)  print(thisdict)  {'key1': 0, 'key2': 0, 'key3': 0} |
| get() | Returns the value of the specified key | car = {   "brand": "Ford",   "model": "Mustang",   "year": 1964 } x = car.get("model") print(x)  Mustang |
| items() | Returns a list containing a tuple for each key value pair | car = {   "brand": "Ford",   "model": "Mustang",   "year": 1964 } x = car.items() print(x)  dict\_items([('brand', 'Ford'), ('model', 'Mustang'), ('year', 1964)]) |
| keys() | Returns a list containing the dictionary's keys | car = {   "brand": "Ford",   "model": "Mustang",   "year": 1964 } x = car.keys() print(x)  dict\_keys(['brand', 'model', 'year']) |
| pop() | Removes the element with the specified key | car = {   "brand": "Ford",   "model": "Mustang",   "year": 1964 } car.pop("model") print(car)  {'brand': 'Ford', 'year': 1964} |
| popitem() | Removes the last inserted key-value pair | car = {   "brand": "Ford",   "model": "Mustang",   "year": 1964 } car. popitem print(car)  {'brand': 'Ford', model: ‘Mustang’} |
| setdefault() | Returns the value of the specified key. If the key does not exist: insert the key, with the specified value | car = {   "brand": "Ford",   "model": "Mustang",   "year": 1964 } x = car.setdefault("model", "Bronco") print(x)  Mustang |
| update() | Updates the dictionary with the specified key-value pairs | car = {   "brand": "Ford",   "model": "Mustang",   "year": 1964 } car.update({"color": "White"}) print(car)  {'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'color': 'White'} |
| values() | Returns a list of all the values in the dictionary | car = {   "brand": "Ford",   "model": "Mustang",   "year": 1964 } x = car.values() print(x)  dict\_values(['Ford', 'Mustang', 1964]) |

**FILE HANDLING:**

Python has several functions for creating, reading, updating and deleting files.

* The key function for working with files in the python is **fopen()** function.
* The **fopen()** function takes two parameters; **filename, mode**
* There are four different modes for opening a file:

1. **“r”- Read -** Default value. Opens a file for reading, error if the file does not exist.
2. **“a” - Append –** Opens a file for appending, creates the file if it does not exist.
3. **“w” - Write –** Opens a file for writing, creates the file if it does not exist.
4. **“x” -** **Create –** Creates the specified file, returns error if the file exist.

In addition, we can specify if the file should be handled as binary or text mode.

1. **“t” - Text –** Default value. Text mode
2. **“b” -** **Binary** **-** Binary mode (e.g. images)

**Syntax:** To open a file for reading.

**f = open("demofile.txt")**

| **f = open("demofile.txt", "rt") - "r" for read, and "t" for text are the default values, you do not need to specify them.** |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

| **Method** | **What it does** | **Return type** | **Notes** |
| --- | --- | --- | --- |
| **read()** | Reads the **entire file** content at once | String | Useful for small files |
| **readline()** | Reads **one line** from the file each time it’s called | String (one line) | Call multiple times to get more lines |
| **readlines()** | Reads **all lines** and returns a list of lines | List of strings | Each element is a line including newline \n |

**Example:**

with open("data.txt", "r") as f:

content = f.read() # Reads entire file as one string

print(content)

with open("data.txt", "r") as f:

line = f.readline() # Reads first line

print(line)

with open("data.txt", "r") as f:

lines = f.readlines() # Reads all lines into a list

print(lines)