



### Python Keywords:

We cannot use a keyword as variable name, function name or any other identifier. They are used to define the syntax and structure of the Python language. Keywords are case sensitive.

All the keywords except True, False and None are in lowercase and they must be written as it is. The list of all the keywords are given below.

### Keywords in Python programming language

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

## Python Identifiers

Identifier is the name given to entities like class, functions, variables etc. in Python. It helps differentiating one entity from another.

### Rules for writing identifiers

1. Identifiers can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore (\_). Names like myClass, var\_1 and print\_this\_to\_screen, all are valid example.
2. An identifier cannot start with a digit. 1variable is invalid, but variable1 is perfectly fine.
3. Keywords cannot be used as identifiers.
4. We cannot use special symbols like !, @, #, \$, % etc. in our identifier.

### Points to remember:

1. Python is a case-sensitive language. This means, **Variable** and **variable** are not the same. Always name identifiers that make sense.
2. While, c = 10 is valid. Writing count = 10 would make more sense and it would be easier to figure out what it does even when you look at your code after a long gap.
3. Multiple words can be separated using an underscore, this\_is\_a\_long\_variable.
4. We can also use camel-case style of writing, i.e., capitalize every first letter of the word except the initial word without any spaces. For example: camelCaseExample

## Python Statement

Instructions that a Python interpreter can execute are called statements. For example, a = 1 is an assignment statement. if statement, for statement, while statement etc. are other kinds of statements which will be discussed later.

### Multi-line statement

In Python, end of a statement is marked by a newline character. But we can make a statement extend over multiple lines with the line continuation character (\). For example:

```
a = 1 + 2 + 3 + \  
    4 + 5 + 6 + \  
    7 + 8 + 9
```

This is explicit line continuation. In Python, line continuation is implied inside parentheses ( ), brackets [ ] and braces { }. For instance, we can implement the above multi-line statement as:

```
a = (1 + 2 + 3 +  
     4 + 5 + 6 +  
     7 + 8 + 9)
```

Here, the surrounding parentheses ( ) do the line continuation implicitly. Same is the case with [ ] and { }. For example:

```
colors = ['red',  
          'blue',  
          'green']
```

We could also put multiple statements in a single line using semicolons, as follows:

```
a = 1; b = 2; c = 3
```

### Python Indentation:

Most of the programming languages like C, C++, Java use braces { } to define a block of code. Python uses indentation.

A code block (body of a function, loop etc.) starts with indentation and ends with the first unindented line. The amount of indentation is up to you, but it must be consistent throughout that block.

Generally four whitespaces are used for indentation and is preferred over tabs. Here is an example.

```
for i in range(1,11):
```

```
    print(i)
```

```
    if i == 5:
```

```
        break
```

Indentation can be ignored in line continuation. But it's a good idea to always indent. It makes the code more readable. For example:

```
if True:  
    print('Hello')  
    a = 5
```

and

```
if True: print('Hello'); a = 5
```

both are valid and do the same thing. But the former style is clearer.

Incorrect indentation will result into ***IndentationError***

### Python Comments:

In Python, we use the hash (#) symbol to start writing a comment. Comments are for programmers for better understanding of a program. Python Interpreter ignores comment.

```
#This is a comment  
#print out Hello  
print('Hello')
```

### Multi-line comments:

If we have comments that extend multiple lines, one way of doing it is to use hash (#) in the beginning of each line. For example:

```
#This is a long comment  
#and it extends  
#to multiple lines
```

Another way of doing this is to use triple quotes, either `'''` or `"""`.

These triple quotes are generally used for multi-line strings. But they can be used as multi-line comment as well. Unless they are not docstrings, they do not generate any extra code.

```
"""This is also a  
perfect example of  
multi-line comments"""
```

### Docstring in Python:

Docstring is short for documentation string.

It is a string that occurs as the first statement in a module, function, class, or method definition. We must write what a function/class does in the docstring.

Triple quotes are used while writing docstrings. For example:

```
def double(num):  
    """Function to double the value"""
```

```
return 2*num
```

Docstring is available to us as the attribute `__doc__` of the function. Issue the following code in shell once you run the above program.

```
>>> print(double.__doc__)  
Function to double the value
```

## Python Variables, Constants and Literals:

### Variable

In most of the programming languages a variable is a named location used to store data in the memory. Each variable must have a unique name called identifier. It is helpful to think of variables as container that hold data which can be changed later throughout programming.

Non technically, you can suppose variable as a bag to store books in it and those books can be replaced at anytime.

Note: In Python we don't assign values to the variables, whereas Python gives the reference of the object (value) to the variable.

### Declaring Variables in Python

In Python, variables do not need declaration to reserve memory space. The "variable declaration" or "variable initialization" happens automatically when we assign a value to a variable.

Example:

```
a = "abc "  
  
print(a)
```

Note : Python is a ***type inferred*** language, it can automatically infer (know) Apple.com is a String and declare website as a String.

### Changing value of a variable

```
a = "abc "  
  
# assigning a new variable to website  
  
a = "xyz "  
  
print(a)
```

If we want to assign the same value to multiple variables at once, we can do this as

```
x = y = z = "same"
```

```
print (x)
```

```
print (y)
```

```
print (z)
```

### **Constants:**

A constant is a type of variable whose value cannot be changed. It is helpful to think of constants as containers that hold information which cannot be changed later

Non technically, you can think of constant as a bag to store some books and those books cannot be replaced once placed inside the bag.

### **Assigning value to a constant in Python**

In Python, constants are usually declared and assigned on a module. Here, the module means a new file containing variables, functions etc which is imported to main file. Inside the module, constants are written in all capital letters and underscores separating the words.

### **Declaring and assigning value to a constant:**

#### ***Create a constant.py***

```
PI = 3.14
```

```
GRAVITY = 9.8
```

#### ***Create a main.py***

```
import constant
```

```
print(constant.PI)
```

```
print(constant.GRAVITY)
```

In the above program, we create a constant.py module file. Then, we assign the constant value to PI and GRAVITY. After that, we create a main.py file and import the constant module. Finally, we print the constant value.

**Note:** In reality, we don't use constants in Python. The globals or constants module is used throughout the Python programs.

### Rules and Naming convention for variables and constants:

- Create a name that makes sense. Suppose, vowel makes more sense than v.
- Use camelCase notation to declare a variable. It starts with lowercase letter. For example:

myName

myAge

myAddress

- Use capital letters where possible to declare a constant. For example:

PI

G

MASS

TEMP

- Never use special symbols like !, @, #, \$, %, etc.
- Don't start name with a digit.
- names should have combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore (\_). For example

snake\_case

MACRO\_CASE

camelCase

CapWords

**Literals:**

Literal is a raw data given in a variable or constant. In Python, there are various types of literals they are as follows

```
a = 0b1010 #Binary Literals
```

```
b = 100 #Decimal Literal
```

```
c = 0o310 #Octal Literal
```

```
d = 0x12c #Hexadecimal Literal
```

```
#Float Literal
```

```
float_1 = 10.5
```

```
float_2 = 1.5e2
```

```
#Complex Literal
```

```
x = 3.14j
```

```
print(a, b, c, d)
```

```
print(float_1, float_2)
```

```
print(x, x.imag, x.real)
```

In the above program,

- We assigned integer literals into different variables. Here, a is binary literal, b is a decimal literal, c is an octal literal and d is a hexadecimal literal.
- When we print the variables, all the literals are converted into decimal values.
- 10.5 and 1.5e2 are floating point literals. 1.5e2 is expressed with exponential and is equivalent to  $1.5 * 10^2$ .
- We assigned a complex literal i.e 3.14j in variable x. Then we use imaginary literal (x.imag) and real literal (x.real) to create imaginary and real part of complex number.

**String literals**

A string literal is a sequence of characters surrounded by quotes. We can use both single, double or triple quotes for a string. And, a character literal is a single character surrounded by single or double quotes.

```
strings = "This is Python"
```

```
char = "C"
```

```
multiline_str = """This is a multiline string with more than one line code."""
```



```
unicode = u"\u00dcnic\u00f6de"
print(strings)
print(char)
print(multiline_str)
print(unicode)
```

In the above program, "This is Python" is a string literal and C is a character literal. The value with triple-quote """" assigned in the **multiline\_str** is multi-line string literal. The **u"\u00dcnic\u00f6de"** is a unicode literal which supports characters other than English.

### Boolean literals

A Boolean literal can have any of the two values: True or False.

```
x = (1 == True)
y = (1 == False)
a = True + 4
b = False + 10
```

```
print("x is", x)
print("y is", y)
print("a:", a)
print("b:", b)
```

In the above program, we use boolean literal True and False. In Python, True represents the value as 1 and False as 0. The value of x is True because 1 is equal to True. And, the value of y is False because 1 is not equal to False.

Similarly, we can use the True and False in numeric expressions as the value. The value of a is 5 because we add True which has value of 1 with 4. Similarly, b is 10 because we add the False having value of 0 with 10.

### Special literals

Python contains one special literal i.e. **None**. We use it to specify to that field that is not created.

```
drink = "Available"
```

```
food = None
```

```
def menu(x):  
    if x == drink:  
        print(drink)  
    else:  
        print(food)
```

```
menu(drink)
```

```
menu(food)
```

### **Literal Collections:**

There are four different literal collections List literals, Tuple literals, Dict literals, and Set literals.

```
fruits = ["apple", "mango", "orange"] #list  
numbers = (1, 2, 3) #tuple  
alphabets = {'a':'apple', 'b':'ball', 'c':'cat'} #dictionary  
vowels = {'a', 'e', 'i', 'o', 'u'} #set
```

```
print(fruits)  
print(numbers)  
print(alphabets)  
print(vowels)
```

### **Data types in Python:**

Every value in Python has a datatype. Since everything is an object in Python programming, data types are actually classes and variables are instance (object) of these classes.

There are various data types in Python. Some of the important types are listed below.

### Python Numbers:

Integers, floating point numbers and complex numbers falls under Python numbers category. They are defined as int, float and complex class in Python.

We can use the type() function to know which class a variable or a value belongs to and the isinstance() function to check if an object belongs to a particular class.

```
a = 5

print(a, "is of type", type(a))

a = 2.0

print(a, "is of type", type(a))

a = 1+2j

print(a, "is complex number?", isinstance(1+2j,complex))
```

### List:

List is an ordered sequence of items. It is one of the most used datatype in Python and is very flexible. All the items in a list do not need to be of the same type.

```
>>> a = [1, 2.2, 'python']
```

### Slicing:

We can use the slicing operator [ ] to extract an item or a range of items from a list. Index starts form 0 in Python.

```
a = [5,10,15,20,25,30,35,40]

print("a[2] = ", a[2])

print("a[0:3] = ", a[0:3])

print("a[5:] = ", a[5:])
```

Lists are mutable, meaning, value of elements of a list can be altered.

```
>>> a = [1,2,3]
>>> a[2]=4
>>> a
[1, 2, 4]
```

**Tuple:**

Tuple is an ordered sequence of items same as list. The only difference is that tuples are immutable. Tuples once created cannot be modified.

```
t = (5, 'program', 1+3j)
```

We can use the slicing operator [] to extract items but we cannot change its value.

```
t = (5, 'program', 1+3j)
```

```
print("t[1] = ", t[1])
```

```
print("t[0:3] = ", t[0:3])
```

```
# Generates error
```

```
# Tuples are immutable
```

```
t[0] = 10
```

**Strings:**

String is sequence of Unicode characters. We can use single quotes or double quotes to represent strings. Multi-line strings can be denoted using triple quotes, ''' or ''''.

```
s = "This is a string"
```

Like list and tuple, slicing operator [ ] can be used with string. Strings are immutable.

```
s = 'Hello world!'
```

```
print("s[4] = ", s[4])
```

```
print("s[6:11] = ", s[6:11])
```

```
# Generates error
```

```
# Strings are immutable in Python
```

```
s[5] = 'd'
```

**Set:**

Set is an unordered collection of unique items. Set is defined by values separated by comma inside braces { }. Items in a set are not ordered.

```
a = {5,2,3,1,4}

print("a = ", a)

print(type(a))
```

We can perform set operations like union, intersection on two sets. Set have unique values. They eliminate duplicates.

```
>>> a = {1,2,2,3,3,3}
>>> a
{1, 2, 3}
```

**Note:** Since, set are unordered collection, indexing has no meaning. Hence the slicing operator [] does not work.

### Dictionary:

Dictionary is an unordered collection of key-value pairs.

It is generally used when we have a huge amount of data. Dictionaries are optimized for retrieving data. We must know the key to retrieve the value.

In Python, dictionaries are defined within braces {} with each item being a pair in the form key:value. Key and value can be of any type.

```
>>> d = {1:'value', 'key':2}
>>> type(d)
<class 'dict'>
```

We use key to retrieve the respective value. But not the other way around.

```
d = {1:'value','key':2}

print(type(d))

print("d[1] = ", d[1]);

print("d['key'] = ", d['key']);

# Generates error

print("d[2] = ", d[2]);
```

### Conversion between data types:

We can convert between different data types by using different type conversion functions like `int()`, `float()`, `str()` etc.

```
float(5)
5.0
```

Conversion from float to int will truncate the value (make it closer to zero).

```
>>> int(10.6)
10
>>> int(-10.6)
-10
```

Conversion to and from string must contain compatible values.

```
>>> float('2.5')
2.5
>>> str(25)
'25'
>>> int('1p')
Traceback (most recent call last):
  File "<string>", line 301, in runcode
  File "<interactive input>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '1p'
```

We can even convert one sequence to another.

```
>>> set([1,2,3])
{1, 2, 3}
>>> tuple({5,6,7})
(5, 6, 7)
>>> list('hello')
['h', 'e', 'l', 'l', 'o']
```

To convert to dictionary, each element must be a pair

```
>>> dict([[1,2],[3,4]])
{1: 2, 3: 4}
>>> dict([(3,26),(4,44)])
{3: 26, 4: 44}
```

### Type Conversion:

- Implicit Type Conversion
- Explicit Type Conversion

#### Implicit Type Conversion:

In Implicit type conversion, Python automatically converts one data type to another data type.

Let's see an example where Python promotes conversion of lower datatype (integer) to higher data type (float) to avoid data loss.

```
num_int = 123
num_flo = 1.23
num_new = num_int + num_flo
print("datatype of num_int:",type(num_int))
print("datatype of num_flo:",type(num_flo))
print("Value of num_new:",num_new)
print("datatype of num_new:",type(num_new))
```

Now, let's try adding a string and an integer, and see how Python treats it.

Addition of string(higher) data type and integer(lower) datatype:

```
num_int = 123
num_str = "456"
print("Data type of num_int:",type(num_int))
print("Data type of num_str:",type(num_str))
```

```
print(num_int+num_str)
```

In the above program,

- We add two variable **num\_int** and **num\_str**.
- As we can see from the output, we got **TypeError**. Python is not able to use Implicit Conversion in such condition.
- However Python has the solution for this type of situation which is known as Explicit Conversion.

### **Explicit Type Conversion:**

In Explicit Type Conversion, users convert the data type of an object to required data type. We use the predefined functions like `int()`, `float()`, `str()`, etc to perform explicit type conversion.

This type conversion is also called **typecasting** because the user casts (change) the data type of the objects

```
num_int = 123
```

```
num_str = "456"
```

```
print("Data type of num_int:",type(num_int))
```

```
print("Data type of num_str before Type Casting:",type(num_str))
```

```
num_str = int(num_str)
```

```
print("Data type of num_str after Type Casting:",type(num_str))
```

```
num_sum = num_int + num_str
```

```
print("Sum of num_int and num_str:",num_sum)
```

```
print("Data type of the sum:",type(num_sum))
```

We converted `num_str` from string (higher) to integer (lower) type using `int()` function to perform the addition.

### **Key Points to Remember:**

- Type Conversion is the conversion of object from one data type to another data type.
- Implicit Type Conversion is automatically performed by the Python interpreter.
- Python avoids the loss of data in Implicit Type Conversion.
- Explicit Type Conversion is also called Type Casting, the data types of object are converted using predefined function by user.
- In Type Casting loss of data may occur as we enforce the object to specific data type.

### **Python Output Using `print()` function:**



```
a = 5
```

```
print('The value of a is', a)
```

In the second `print()` statement, we can notice that a space was added between the string and the value of variable `a`. This is by default, but we can change it.

The actual syntax of the `print()` function is:

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

```
print(1,2,3,4)
```

```
# Output: 1 2 3 4
```

```
print(1,2,3,4,sep='*')
```

```
# Output: 1*2*3*4
```

```
print(1,2,3,4,sep='#',end='&')
```

```
# Output: 1#2#3#4&
```

### Output formatting:

Sometimes we would like to format our output to make it look attractive. This can be done by using the **`str.format()`** method. This method is visible to any string object.

```
x = 5; y = 10
```

```
print('The value of x is {} and y is {}'.format(x,y))
```

```
The value of x is 5 and y is 10
```

Here the curly braces `{}` are used as placeholders. We can specify the order in which it is printed by using numbers (tuple index).

```
print('I love {0} and {1}'.format('bread','butter'))
```

```
# Output: I love bread and butter
```

```
print('I love {1} and {0}'.format('bread','butter'))
```

```
# Output: I love butter and bread
```

Also, we can even use keyword arguments to format the string.

```
print('Hello {name}, {greeting}'.format(greeting = 'Goodmorning', name = 'John'))  
Hello John, Goodmorning
```

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

```
>>> x = 12.3456789  
>>> print('The value of x is %3.2f' %x)  
The value of x is 12.35  
>>> print('The value of x is %3.4f' %x)  
The value of x is 12.3457
```

### Python Input:

To allow flexibility we might want to take the input from the user. In Python, we have the `input()` function to allow this. The syntax for `input()` is:

```
>>> num = input('Enter a number: ')  
Enter a number: 10  
>>> num  
'10'
```

Here, we can see that the entered value 10 is a string, not a number. To convert this into a number we can use `int()` or `float()` functions.

```
>>> int('10')  
10  
>>> float('10')  
10.0
```

This same operation can be performed using the `eval()` function. But it takes it further. It can evaluate even expressions, provided the input is a string

```
>>> int('2+3')
Traceback (most recent call last):
  File "<string>", line 301, in runcode
  File "<interactive input>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '2+3'
>>> eval('2+3')
5
```

## Operators:

### Arithmetic operators:

Arithmetic operators in Python

Operator	Meaning	Example
+	Add two operands or unary plus	x + y +2
-	Subtract right operand from the left or unary minus	x - y -2
*	Multiply two operands	x * y
/	Divide left operand by the right one (always results into float)	x / y
%	Modulus - remainder of the division of left operand by the right	x % y (remainder of x/y)
//	Floor division - division that results into whole number adjusted to the left in the number line	x // y
**	Exponent - left operand raised to the power of right	x**y (x to the power y)

x = 15

y = 4

```

# Output: x + y = 19
print('x + y =',x+y)

# Output: x - y = 11
print('x - y =',x-y)

# Output: x * y = 60
print('x * y =',x*y)

# Output: x / y = 3.75
print('x / y =',x/y)

# Output: x // y = 3
print('x // y =',x//y)

# Output: x ** y = 50625
print('x ** y =',x**y)

```

### Comparison operators:

It either returns **True** or **False** according to the condition.

Comparison operators in Python

Operator	Meaning	Example
>	Greater than - True if left operand is greater than the right	<code>x &gt; y</code>
<	Less than - True if left operand is less than the right	<code>x &lt; y</code>
==	Equal to - True if both operands are equal	<code>x == y</code>
!=	Not equal to - True if operands are not equal	<code>x != y</code>
>=	Greater than or equal to - True if left operand is greater than or equal to the right	<code>x &gt;= y</code>
<=	Less than or equal to - True if left operand is less than or equal to the right	<code>x &lt;= y</code>

```
x = 10
```

```
y = 12
```

```
# Output: x > y is False
```

```
print('x > y is',x>y)
```

```
# Output: x < y is True
```

```
print('x < y is',x<y)
```

```
# Output: x == y is False
```

```
print('x == y is',x==y)
```

```
# Output: x != y is True
```

```
print('x != y is',x!=y)
```

```
# Output: x >= y is False
```

```
print('x >= y is',x>=y)
```

```
# Output: x <= y is True
```

```
print('x <= y is',x<=y)
```

### Logical operators:

#### Logical operators in Python

Operator	Meaning	Example
and	True if both the operands are true	x and y
or	True if either of the operands is true	x or y
not	True if operand is false (complements the operand)	not x

```
x = True
```

```
y = False
```

# Output: x and y is False

```
print('x and y is',x and y)
```

# Output: x or y is True

```
print('x or y is',x or y)
```

# Output: not x is False

```
print('not x is',not x)
```

### Assignment operators:

#### Assignment operators in Python

Operator	Example	Equivatent to
=	x = 5	x = 5
+=	x += 5	x = x + 5
-=	x -= 5	x = x - 5
*=	x *= 5	x = x * 5
/=	x /= 5	x = x / 5
%=	x %= 5	x = x % 5
//=	x //= 5	x = x // 5
**=	x **= 5	x = x ** 5
&=	x &= 5	x = x & 5
=	x  = 5	x = x   5
^=	x ^= 5	x = x ^ 5
>>=	x >>= 5	x = x >> 5
<<=	x <<= 5	x = x << 5

**Identity operators:**

**is** and **is not** are the identity operators in Python. They are used to check if two values (or variables) are located on the same part of the memory.

Identity operators in Python

Operator	Meaning	Example
is	True if the operands are identical (refer to the same object)	x is True
is not	True if the operands are not identical (do not refer to the same object)	x is not True

```
x1 = 5
```

```
y1 = 5
```

```
x2 = 'Hello'
```

```
y2 = 'Hello'
```

```
x3 = [1,2,3]
```

```
y3 = [1,2,3]
```

```
# Output: False
```

```
print(x1 is not y1)
```

```
# Output: True
```

```
print(x2 is y2)
```

```
# Output: False
```

```
print(x3 is y3)
```

**Note:**

Here, we see that x1 and y1 are integers of same values, so they are equal as well as identical. Same is the case with x2 and y2 (strings).

But x3 and y3 are list. They are equal but not identical. Since list are mutable (can be changed), interpreter locates them separately in memory although they are equal.

**Membership operators:**

in and not in are the membership operators in Python. They are used to test whether a value or variable is found in a sequence (string, list, tuple, set and dictionary).

But, in a dictionary we can only test for presence of key, not the value.

Operator	Meaning	Example
in	True if value/variable is found in the sequence	5 in x
not in	True if value/variable is not found in the sequence	5 not in x

```
x = 'Hello world'
```

```
y = {1:'a',2:'b'}
```

```
# Output: True
```

```
print('H' in x)
```

```
# Output: True
```

```
print('hello' not in x)
```

```
# Output: True
```

```
print(1 in y)
```

```
# Output: False
```

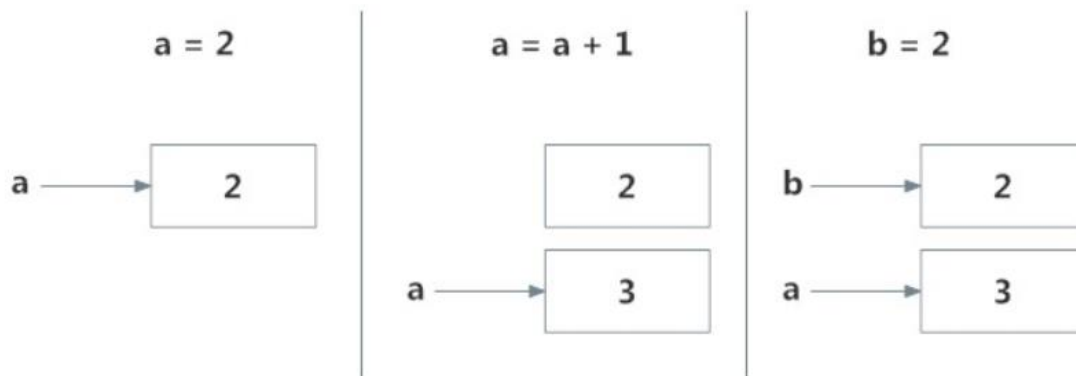
```
print('a' in y)
```

**Note:**

Here, 'H' is in x but 'hello' is not present in x (remember, Python is case sensitive). Similarly, 1 is key and 'a' is the value in dictionary y. Hence, 'a' in y returns False.

Object creation (names):





**if Statement Syntax:**

```
if test expression:
```

```
    statement(s)
```

## Python if Statement Flowchart

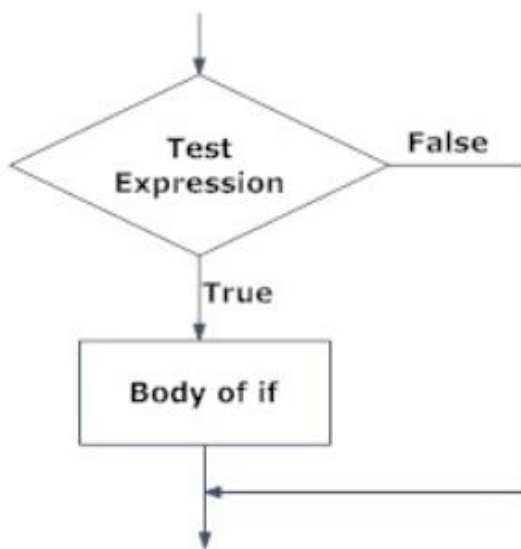


Fig: Operation of if statement

```
num = 3
```

```
if num > 0:
```

```
    print(num, "is a positive number.")
```

```
print("This is always printed.")
```

```
num = -1
```

```
if num > 0:
```

```
    print(num, "is a positive number.")
```

```
print("This is also always printed.")
```

**if...else Statement:**

```
if test expression:
```

```
    Body of if
```

```
else:
```

```
    Body of else
```

## Python if..else Flowchart

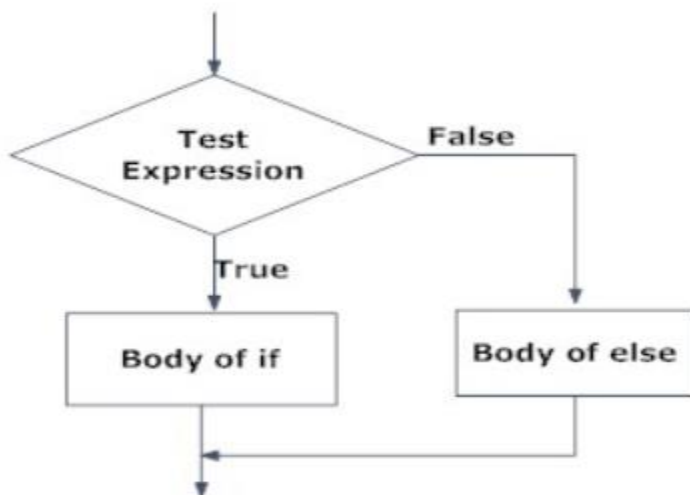


Fig: Operation of if...else statement

# Program checks if the number is positive or negative

# And displays an appropriate message

num = 3

# Try these two variations as well.

# num = -5

# num = 0

if num >= 0:

print("Positive or Zero")

else:

print("Negative number")

**if...elif...else Statement:**

```
if test expression:
```

```
    Body of if
```

```
elif test expression:
```

```
    Body of elif
```

```
else:
```

```
    Body of else
```

# Flowchart of if...elif...else

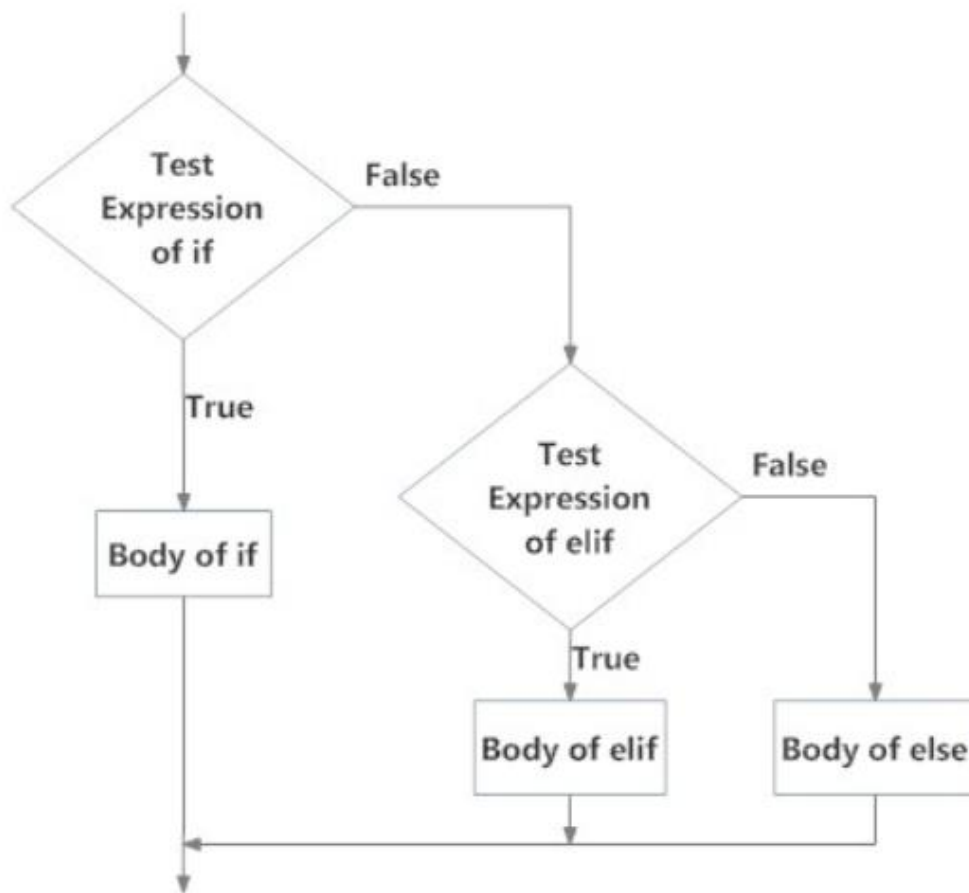


Fig: Operation of if...elif...else statement

```
# In this program,  
# we check if the number is positive or  
# negative or zero and  
# display an appropriate message  
num = 3.4  
  
# Try these two variations as well:  
  
# num = 0  
  
# num = -4.5
```

```
if num > 0:
    print("Positive number")
elif num == 0:
    print("Zero")
else:
    print("Negative number")
```

#### **Nested if Example:**

```
num = float(input("Enter a number: "))
if num >= 0:
    if num == 0:
        print("Zero")
    else:
        print("Positive number")
else:
    print("Negative number")
```

#### **for loop:**

```
for val in sequence:
```

    Body of for

# Flowchart of for Loop

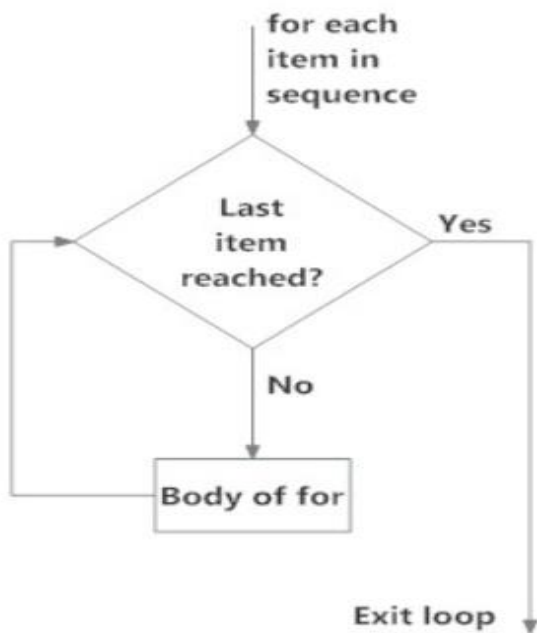


Fig: operation of for loop

```
# Program to find the sum of all numbers stored in a list
```

```
# List of numbers
```

```
numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]
```

```
# variable to store the sum
```

```
sum = 0
```

```
# iterate over the list
```

```
for val in numbers:
```

```
    sum = sum+val
```

```
# Output: The sum is 48
```

```
print("The sum is", sum)
```

### The range() function:

We can generate a sequence of numbers using range() function. range(10) will generate numbers from 0 to 9 (10 numbers).

We can also define the start, stop and step size as range(start,stop,step size). step size defaults to 1 if not provided.

This function does not store all the values in memory, it would be inefficient. So it remembers the start, stop, step size and generates the next number on the go.

To force this function to output all the items, we can use the function list().

```
# Output: range(0, 10)

print(range(10))

# Output: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

print(list(range(10)))

# Output: [2, 3, 4, 5, 6, 7]

print(list(range(2, 8)))

# Output: [2, 5, 8, 11, 14, 17]

print(list(range(2, 20, 3)))
```

# Program to iterate through a list using indexing

```
genre = ['pop', 'rock', 'jazz']

# iterate over the list using index
for i in range(len(genre)):
    print("I like", genre[i])
```

### for loop with else:

A for loop can have an optional else block as well. The else part is executed if the items in the sequence used in for loop exhausts. break statement can be used to stop a for loop. In such case, the else part is ignored. Hence, a for loop's else part runs if no break occurs.

```
digits = [0, 1, 5]
```

```
for i in digits:
```

```
    print(i)
```

```
else:
```

```
    print("No items left.")
```

### while loop:

The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is true.

Syntax of while Loop:

```
while test_expression:
```

```
    Body of while
```

## Flowchart of while Loop

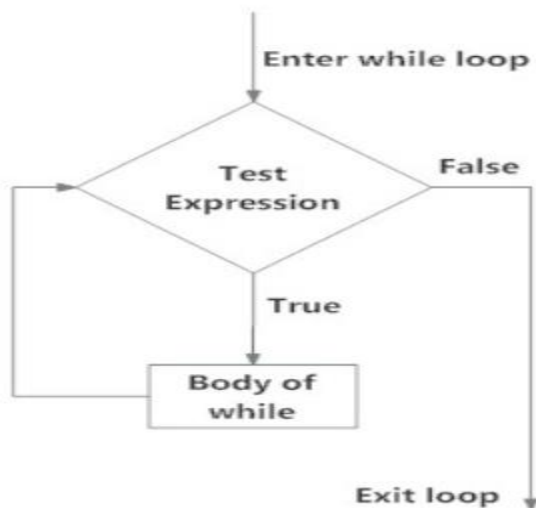


Fig: operation of while loop



```
# Program to add natural
# numbers upto
# sum = 1+2+3+...+n
# To take input from the user,
# n = int(input("Enter n: "))
n = 10

# initialize sum and counter
sum = 0
i = 1

while i <= n:
    sum = sum + i
    i = i+1  # update counter

# print the sum
print("The sum is", sum)
```

**Note:** We need to increase the value of counter variable in the body of the loop. This is very important (and mostly forgotten). Failing to do so will result in an infinite loop (never ending loop).

**while loop with else:**

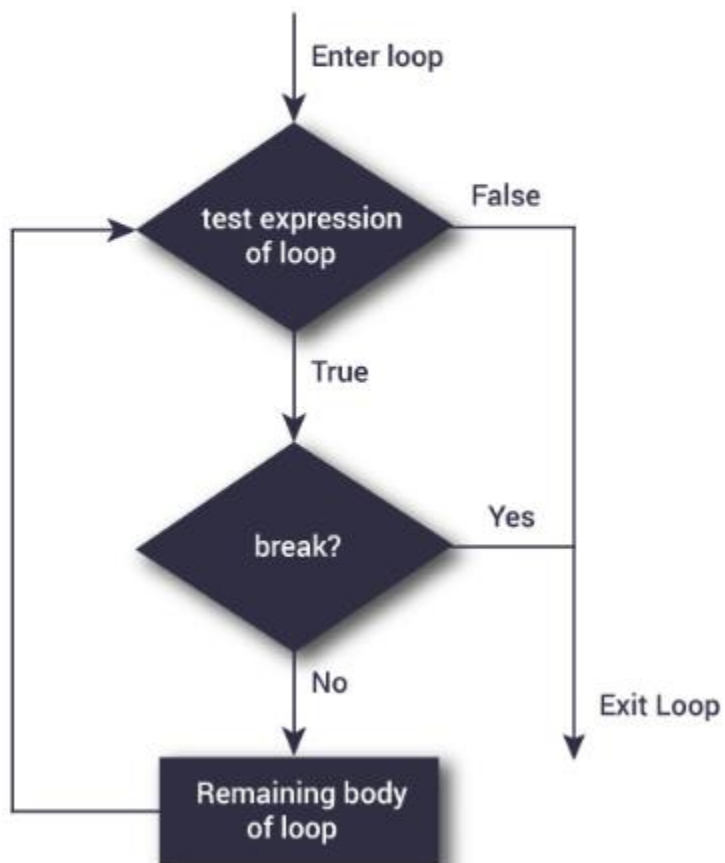
```
counter = 0

while counter < 3:
    print("Inside loop")
    counter = counter + 1
else:
    print("Inside else")
```

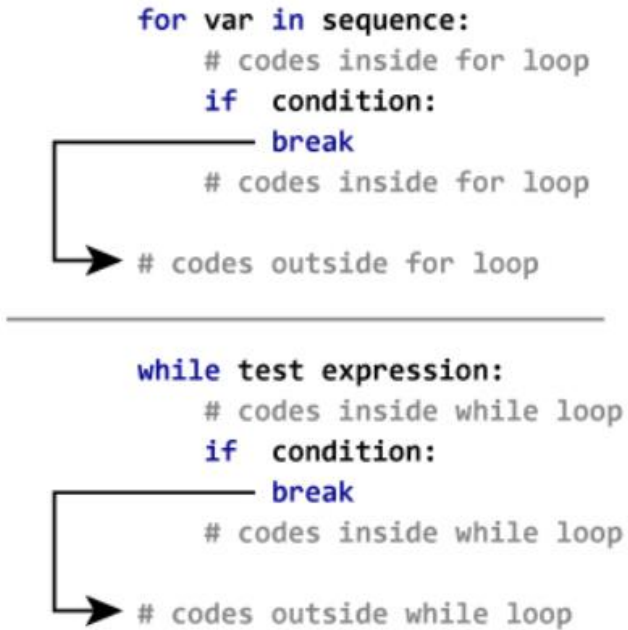
Syntax of break:

```
break
```

## Flowchart of break



The working of break statement in `for loop` and `while loop` is shown below.



# Use of break statement inside loop

```
for val in "string":
```

```
    if val == "i":
```

```
        break
```

```
    print(val)
```

```
print("The end")
```

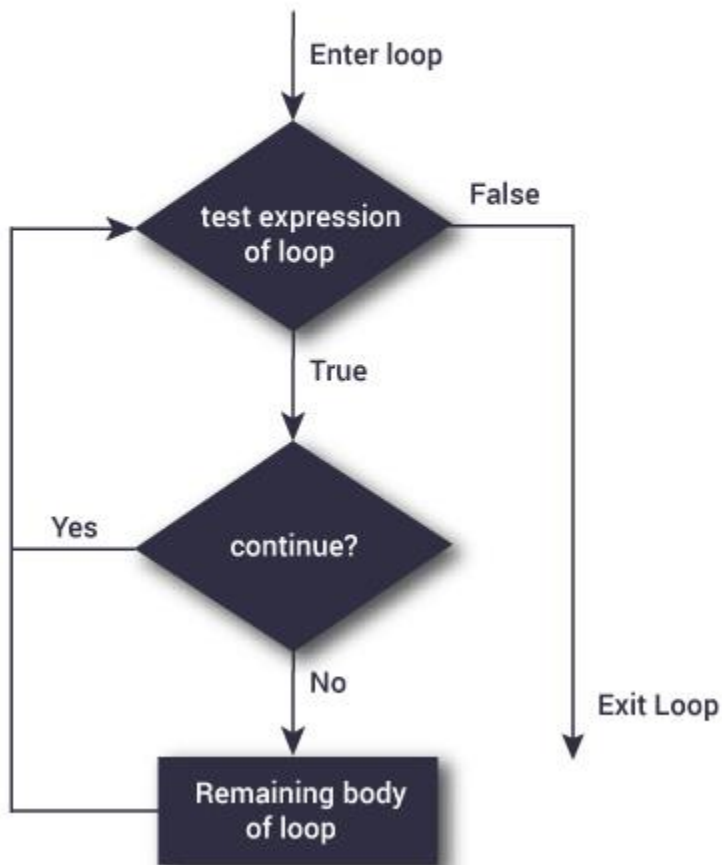
**continue statement:**

The continue statement is used to skip the rest of the code inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration

### Syntax of Continue:

```
continue
```

## Flowchart of continue



# Program to show the use of continue statement inside loops

```
for val in "string":
```

```
    if val == "i":
```

```
        continue
```

```
    print(val)
```

```
print("The end")
```

The working of continue statement in for and while loop is shown below.

```
for var in sequence:
    # codes inside for loop
    if condition:
        continue
    # codes inside for loop

# codes outside for loop
```

---

```
while test expression:
    # codes inside while loop
    if condition:
        continue
    # codes inside while loop

# codes outside while loop
```

**Note:** We continue with the loop, if the string is "i", not executing the rest of the block

#### Pass statement:

In Python programming, pass is a null statement. The difference between a comment and pass statement in Python is that, while the interpreter ignores a comment entirely, pass is not ignored.

However, nothing happens when pass is executed. It results into no operation (NOP).

Syntax of pass:

```
pass
```

```
# pass is just a placeholder for
# functionality to be added later.
```

```
sequence = {'p', 'a', 's', 's'}
```

```
for val in sequence:
```

```
    pass
```

We can do the same thing in an empty function as well.

```
def function(args):  
    pass
```

### Functions:

Function is a group of related statements that perform a specific task.

### Syntax of Function:

```
def function_name(parameters):  
  
    """docstring"""  
  
    statement(s)
```

- Keyword **def** marks the start of function header.
- A function name to uniquely identify it. Function naming follows the same rules of writing **identifiers** in Python.
- Parameters (arguments) through which we pass values to a function. They are optional.
- A colon (:) to mark the end of function header.
- Optional documentation string (docstring) to describe what the function does.
- One or more valid python statements that make up the function body. Statements must have same **indentation** level (usually 4 spaces).
- An optional return statement to return a value from the function.

```
def greet(name):  
    """This function greets to  
    the person passed in as parameter"""  
    print("Hello, " + name + ". Good morning!")
```

### Call a function:

```
greet('Paul')
```

### Docstring:

Documentation string is used to explain in brief, what a function does. Although optional, documentation is a good programming practice. Unless you can remember what you had for dinner last week, always document your code.

```
>>> print(greet.__doc__)
This function greets to
    the person passed into the name parameter
```

### The return statement:

The return statement is used to exit a function and go back to the place from where it was called.

### Syntax of return:

```
return [expression_list]
```

This statement can contain expression which gets evaluated and the value is returned. If there is no expression in the statement or the **return** statement itself is not present inside a function, then the function will return the **None** object.

```
>>> print(greet("Dad"))
Hello, Dad. Good morning!
None
```

Here, **None** is the returned value.

```
def absolute_value(num):
```

```
    """This function returns the absolute
    value of the entered number"""
```

```
    if num >= 0:
```

```
        return num
```

```
    else:
```

```
        return -num
```

# Output: 2

```
print(absolute_value(2))
```

# Output: 4

```
print(absolute_value(-4))
```

### **Scope and Lifetime of variables:**

Scope of a variable is the portion of a program where the variable is recognized. Parameters and variables defined inside a function is not visible from outside. Hence, they have a local scope.

Lifetime of a variable is the period throughout which the variable exists in the memory. The lifetime of variables inside a function is as long as the function executes.

They are destroyed once we return from the function. Hence, a function does not remember the value of a variable from its previous calls.

```
def my_func():
```

```
    x = 10
```

```
    print("Value inside function:",x)
```

```
x = 20
```

```
my_func()
```

```
print("Value outside function:",x)
```

#Output

Value inside function: 10

Value outside function: 20

Here, we can see that the value of x is 20 initially. Even though the function my\_func() changed the value of x to 10, it did not effect the value outside the function.

### **Function Arguments:**



```
def greet(name,msg):  
    """This function greets to  
    the person with the provided message"""  
    print("Hello",name + ', ' + msg)  
  
greet("guys","Good morning!")
```

If we call it with different number of arguments, the interpreter will complain. Below is a call to this function with one and no arguments along with their respective error messages.

```
>>> greet("guys")      # only one argument  
  
TypeError: greet() missing 1 required positional argument: 'msg'
```

### **Default Arguments:**

We can provide a default value to an argument by using the assignment operator (=)

```
def greet(name, msg = "Good morning!"):   
    """ This function greets to the person with the provided message.  
    If message is not provided, it defaults to "Good morning!" """  
    print("Hello",name + ', ' + msg)  
  
greet("Kate")  
greet("Bruce","How do you do?")
```

Note: Any number of arguments in a function can have a default value. But once we have a default argument, all the arguments to its right must also have default values.

This means to say, non-default arguments cannot follow default arguments. For example, if we had defined the function header above as:

```
def greet(msg = "Good morning!", name):
```

We would get an error as:

```
SyntaxError: non-default argument follows default argument
```

### **Keyword Arguments:**

```
>>> # 2 keyword arguments

>>> greet(name = "Bruce",msg = "How do you do?")

>>> # 2 keyword arguments (out of order)

>>> greet(msg = "How do you do?",name = "Bruce")

>>> # 1 positional, 1 keyword argument

>>> greet("Bruce",msg = "How do you do?")
```

As we can see, we can mix positional arguments with keyword arguments during a function call. But we must keep in mind that keyword arguments must follow positional arguments.

### **Arbitrary Arguments:**

Sometimes, we do not know in advance the number of arguments that will be passed into a function.

In the function definition we use an asterisk (\*) before the parameter name to denote this kind of argument.

```
def greet(*names):

    """This function greets all

    the person in the names tuple."""

    # names is a tuple with arguments

    for name in names:

        print("Hello",name)

greet("Monica","Luke","Steve","John")
```

## Recursion:

Recursion is the process of defining something in terms of itself

# An example of a recursive function to find the factorial of a number

```
def calc_factorial(x):  
    """This is a recursive function  
    to find the factorial of an integer"""  
  
    if x == 1:  
        return 1  
    else:  
        return (x * calc_factorial(x-1))  
  
num = 4  
print("The factorial of", num, "is", calc_factorial(num))
```

```
calc_factorial(4)          # 1st call with 4  
4 * calc_factorial(3)      # 2nd call with 3  
4 * 3 * calc_factorial(2)  # 3rd call with 2  
4 * 3 * 2 * calc_factorial(1) # 4th call with 1  
4 * 3 * 2 * 1              # return from 4th call as number=1  
4 * 3 * 2                  # return from 3rd call  
4 * 6                      # return from 2nd call  
24                         # return from 1st call
```

**Note:** Our recursion ends when the number reduces to 1. This is called the base condition.

Every recursive function must have a base condition that stops the recursion or else the function calls itself infinitely

### Lambda function:

Normal functions are defined using the **def** keyword, in Python anonymous functions are defined using the **lambda** keyword.

### Syntax of Lambda Function:

```
lambda arguments: expression
```

```
double = lambda x: x * 2
```

```
# Output: 10
```

```
print(double(5))
```

```
double = lambda x: x * 2
```

is nearly the same as

```
def double(x):  
  
    return x * 2
```

### Example with filter():

The **filter()** function in Python takes in a function and a list as arguments.

The function is called with all the items in the list and a new list is returned which contains items for which the function evaluates to **True**

```
# Program to filter out only the even items from a list
```

```
my_list = [1, 5, 4, 6, 8, 11, 3, 12]
```

```
new_list = list(filter(lambda x: (x%2 == 0) , my_list))
```

```
# Output: [4, 6, 8, 12]
```

```
print(new_list)
```

**Example with map():**

The **map()** function in Python takes in a function and a list.

The function is called with all the items in the list and a new list is returned which contains items returned by that function for each item.

# Program to double each item in a list using map()

```
my_list = [1, 5, 4, 6, 8, 11, 3, 12]
new_list = list(map(lambda x: x * 2 , my_list))
# Output: [2, 10, 8, 12, 16, 22, 6, 24]
print(new_list)
```

**Global Variables:**

In Python, a variable declared outside of the function or in global scope is known as global variable. This means, global variable can be accessed inside or outside of the function.

```
x = "global"
def func():
    print("x inside :", x)
func()
print("x outside:", x)
```

**Local Variables:**

A variable declared inside the function's body or in the local scope is known as local variable.

```
def foo():
    y = "local"
foo()
print(y)
```

When we run the code, the will output be:

```
NameError: name 'y' is not defined
```

The output shows an error, because we are trying to access a local variable **y** in a global scope whereas the local variable only works inside `foo()` or local scope.

### Create a Local Variable:

```
def foo():
```

```
    y = "local"
```

```
    print(y)
```

```
foo()
```

When we run the code, it will output:

```
local
```

### Global and Local variables in same code:

```
x = "global"
```

```
def foo():
```

```
    global x
```

```
    y = "local"
```

```
    x = x * 2
```

```
    print(x)
```

```
    print(y)
```

```
foo()
```

When we run the code, the will output be:

```
global global
```

```
local
```

### Global variable and Local variable with same name:

```
x = 5

def foo():
    x = 10
    print("local x:", x)

foo()

print("global x:", x)
```

When we run the code, the will output be:

```
local x: 10

global x: 5
```

### Global Keyword:

#### Rules of global Keyword:

- When we create a variable inside a function, it's local by default.
- When we define a variable outside of a function, it's global by default. You don't have to use global keyword.
- We use **global** keyword to read and write a global variable inside a function.
- Use of global keyword outside a function has no effect

#### Accessing global Variable From Inside a Function:

```
c = 1 # global variable

def add():
    print(c)

add() #output is 1
```

#### Modifying Global Variable From Inside the Function:

```
c = 1 # global variable

def add():
    c = c + 2 # increment c by 2
    print(c)

add()
```

When we run above program, the output shows an error:

```
UnboundLocalError: local variable 'c' referenced before assignment
```

**Note:** This is because we can only access the global variable but cannot modify it from inside the function.

The solution for this is to use the **global** keyword.

#### Changing Global Variable From Inside a Function using global:

```
c = 0 # global variable
def add():
    global c
    c = c + 2 # increment by 2
    print("Inside add():", c)
add()
print("In main:", c)
```

When we run above program, the output will be:

```
Inside add(): 2

In main: 2
```

As we can see, change also occurred on the global variable outside the function, **c = 2**.



### List creation:

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

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

```
# empty list
my_list = []
# list of integers
my_list = [1, 2, 3]
# list with mixed datatypes
my_list = [1, "Hello", 3.4]
```

Also, a list can even have another list as an item. This is called nested list.

```
# nested list

my_list = ["mouse", [8, 4, 6], ['a']]
```

### Accessing elements from a list:

#### List Index:

We can use the index operator [] to access an item in a list. Index starts from 0.

Trying to access an element other than this will raise an **IndexError**. The index must be an integer. We can't use float or other types, this will result into **TypeError**.

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

```
# Output: p
```

```
print(my_list[0])
```

```
# Output: o
```

```
print(my_list[2])
```

```
# Output: e
```

```
print(my_list[4])
```

```
# Error! Only integer can be used for indexing
```

```
# my_list[4.0]
```

Nested list are accessed using nested indexing.

# Nested List

```
n_list = ["Happy", [2,0,1,5]]
```

# Nested indexing

# Output: a

```
print(n_list[0][1])
```

# Output: 5

```
print(n_list[1][3])
```

### **Negative indexing:**

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

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

# Output: e

```
print(my_list[-1])
```

# Output: p

```
print(my_list[-5])
```

### **How to slice lists:**

We can access a range of items in a list by using the slicing operator (colon).

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

# elements 3rd to 5th

```
print(my_list[2:5])
```

# elements beginning to 4th

```
print(my_list[:-5])
```

# elements 6th to end

```
print(my_list[5:])
```

# elements beginning to end

```
print(my_list[:])
```

### change or add elements to a list:

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

```
# mistake values
odd = [2, 4, 6, 8]

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

# Output: [1, 4, 6, 8]
print(odd)
```

We can add one item to a list using **append()** method or add several items using **extend()** method.

```
odd = [1, 3, 5]
odd.append(7)
# Output: [1, 3, 5, 7]
print(odd)
odd.extend([9, 11, 13])
# Output: [1, 3, 5, 7, 9, 11, 13]
print(odd)
```

### List concatenation:

We can also use + operator to combine two lists.

```
odd = [1, 3, 5]
# Output: [1, 3, 5, 9, 7, 5]
print(odd + [9, 7, 5])
#Output: ["re", "re", "re"]
print(["re"] * 3) #The * operator repeats a list for the given number of
times
```

Furthermore, we can insert one item at a desired location by using the method **insert()** or insert multiple items by squeezing it into an empty slice of a list

```
odd = [1, 9]
odd.insert(1,3)
# Output: [1, 3, 9]
print(odd)
odd[2:2] = [5, 7]
# Output: [1, 3, 5, 7, 9]
print(odd)
```

### To delete or remove elements from a list:

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

```
my_list = ['p','r','o','b','l','e','m']
# delete one item
del my_list[2]
# Output: ['p', 'r', 'b', 'l', 'e', 'm']
print(my_list)
# delete multiple items
del my_list[1:5]
# Output: ['p', 'm']
print(my_list)
# delete entire list
del my_list
# Error: List not defined
print(my_list)
```

We can use **remove()** method to remove the given item or **pop()** method to remove an item at the given index.

The **pop()** method removes and returns the last item if index is not provided. This helps us implement lists as stacks (first in, last out data structure). We can also use the **clear()** method to empty a list.

```
my_list = ['p','r','o','b','l','e','m']
my_list.remove('p')
# Output: ['r', 'o', 'b', 'l', 'e', 'm']
print(my_list)
# Output: 'o'
print(my_list.pop(1))
# Output: ['r', 'b', 'l', 'e', 'm']
print(my_list)
# Output: 'm'
print(my_list.pop())
# Output: ['r', 'b', 'l', 'e']
print(my_list)
my_list.clear()
# Output: []
print(my_list)
```

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

```
>>> my_list = ['p','r','o','b','l','e','m']
>>> my_list[2:3] = []
>>> my_list
['p', 'r', 'b', 'l', 'e', 'm']
>>> my_list[2:5] = []
>>> my_list
['p', 'r', 'm']
```

## Python List Methods

**append()** - Add an element to the end of the list

**extend()** - Add all elements of a list to the another list

**insert()** - Insert an item at the defined index

**remove()** - Removes an item from the list

**pop()** - Removes and returns an element at the given index

**clear()** - Removes all items from the list

**index()** - Returns the index of the first matched item

**count()** - Returns the count of number of items passed as an argument

**sort()** - Sort items in a list in ascending order

**reverse()** - Reverse the order of items in the list

**copy()** - Returns a shallow copy of the list

Some more examples of list methods:

```
my_list = [3, 8, 1, 6, 0, 8, 4]
```

```
# Output: 1
```

```
print(my_list.index(8))
```

```
# Output: 2
```

```
print(my_list.count(8))
```

```
my_list.sort()
```

```
# Output: [0, 1, 3, 4, 6, 8, 8]
```

```
print(my_list)
```

```
my_list.reverse()
```

```
# Output: [8, 8, 6, 4, 3, 1, 0]
```

```
print(my_list)
```

### List comprehension:

List comprehension is an elegant and concise way to create new list from an existing list

List comprehension consists of an expression followed by for statement inside square brackets.

Here is an example to make a list with each item being increasing power of 2.

```
pow2 = [2 ** x for x in range(10)]
```

```
# Output: [1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

```
print(pow2)
```

This code is equivalent to:

```
pow2 = []  
for x in range(10):  
    pow2.append(2 ** x)
```

A list comprehension can optionally contain more **for** or **if** statements. An optional if statement can filter out items for the new list. Here are some examples.

```
>>> pow2 = [2 ** x for x in range(10) if x > 5]  
>>> pow2  
[64, 128, 256, 512]  
>>> odd = [x for x in range(20) if x % 2 == 1]  
>>> odd  
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

```
>>> [x+y for x in ['Python ', 'C '] for y in ['Language', 'Programming']]
['Python Language', 'Python Programming', 'C Language', 'C Programming']
```

### List Membership Test:

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

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

```
# Output: True
```

```
print('p' in my_list)
```

```
# Output: False
```

```
print('a' in my_list)
```

```
# Output: True
```

```
print('c' not in my_list)
```

### Iterating Through a List:

```
for fruit in ['apple','banana','mango']:
    print("I like",fruit)
```

### Built-in Functions with List:

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

Built-in Functions with List

Function	Description
----------	-------------

<code>all()</code>	Return True if all elements of the list are true (or if the list is empty).
--------------------	---

<code>any()</code>	Return True if any element of the list is true. If the list is empty, return False.
--------------------	---



`enumerate()` Return an enumerate object. It contains the index and value of all the items of list as a tuple.

`len()` Return the length (the number of items) in the list.

`list()` Convert an iterable (tuple, string, set, dictionary) to a list.

`max()` Return the largest item in the list.

`min()` Return the smallest item in the list

`sorted()` Return a new sorted list (does not sort the list itself).

`sum()` Return the sum of all elements in the list.

### **Tuple:**

In Python programming, a tuple is similar to a list. The difference between the two is that we cannot change the elements of a tuple once it is assigned whereas in a list, elements can be changed.

# tuple with mixed datatypes

# Output: (1, "Hello", 3.4)

```
my_tuple = (1, "Hello", 3.4)
```

```
print(my_tuple)
```

# nested tuple

# Output: ("mouse", [8, 4, 6], (1, 2, 3))

```
my_tuple = ("mouse", [8, 4, 6], (1, 2, 3))
```

```
print(my_tuple)
```

# tuple can be created without parentheses

# also called tuple packing

# Output: 3, 4.6, "dog"

```
my_tuple = 3, 4.6, "dog"
```

```
print(my_tuple)
```

# tuple unpacking is also possible

# Output:

# 3

# 4.6

```
# dog
a, b, c = my_tuple
print(a)
print(b)
print(c)
```

**Note:** Creating a tuple with one element is a bit tricky.

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

```
# only parentheses is not enough
# Output: <class 'str'>
my_tuple = ("hello")
print(type(my_tuple))
```

```
# need a comma at the end
# Output: <class 'tuple'>
my_tuple = ("hello",)
print(type(my_tuple))
```

```
# parentheses is optional
# Output: <class 'tuple'>
my_tuple = "hello",
print(type(my_tuple))
```

### **Accessing Elements in a Tuple:**

#### **1.Indexing:**

```
# nested tuple
n_tuple = ("mouse", [8, 4, 6], (1, 2, 3))
```

```
# nested index  
# Output: 's'  
print(n_tuple[0][3])
```

```
# nested index  
# Output: 4  
print(n_tuple[1][1])
```

## 2. Negative Indexing:

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

```
my_tuple = ('p','e','r','m','i','t')  
# Output: 't'  
print(my_tuple[-1])  
# Output: 'p'  
print(my_tuple[-6])
```

## 3. Slicing:

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

```
my_tuple = ('p','r','o','g','r','a','m','i','z')  
# elements 2nd to 4th  
# Output: ('r', 'o', 'g')  
print(my_tuple[1:4])  
# elements beginning to 2nd  
# Output: ('p', 'r')  
print(my_tuple[:2])  
# elements 8th to end  
# Output: ('i', 'z')  
print(my_tuple[7:])
```

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

### **Changing a Tuple:**

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

```
my_tuple = (4, 2, 3, [6, 5])
# we cannot change an element
# If you uncomment line 8
# you will get an error:
# TypeError: 'tuple' object does not support item assignment
#my_tuple[1] = 9
# but item of mutable element can be changed
# Output: (4, 2, 3, [9, 5])
my_tuple[3][0] = 9
print(my_tuple)
```

### **Concatenation:**

```
# Concatenation
# Output: (1, 2, 3, 4, 5, 6)
print((1, 2, 3) + (4, 5, 6))
```

```
# Repeat
# Output: ('Repeat', 'Repeat', 'Repeat')
print(("Repeat",) * 3)
```

### **Delete:**

We cannot delete or remove items from a tuple.

But deleting a tuple entirely is possible using the keyword **del**.

## Python Tuple Methods:

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

Python Tuple Method	
Method	Description
<a href="#"><code>count(x)</code></a>	Return the number of items that is equal to <code>x</code>
<a href="#"><code>index(x)</code></a>	Return index of first item that is equal to <code>x</code>

Example:

```
my_tuple = ('a','p','p','l','e')
```

```
# Count
```

```
# Output: 2
```

```
print(my_tuple.count('p'))
```

```
# Index
```

```
# Output: 3
```

```
print(my_tuple.index('l'))
```

## String:

A string is a sequence of characters.

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

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

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

In Python, string is a sequence of Unicode character. Unicode was introduced to include every character in all languages and bring uniformity in encoding.

```
# all of the following are equivalent
```

```
my_string = 'Hello'
print(my_string)

my_string = "Hello"
print(my_string)

my_string = '''Hello'''
print(my_string)

# triple quotes string can extend multiple lines
my_string = """Hello, welcome to
    the world of Python"""
print(my_string)
```

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

#### **Concatenation of Two or More Strings:**

```
str1 = 'Hello'
str2 = 'World!'

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

# using *
print('str1 * 3 =', str1 * 3)
```

#### **Iterating Through String:**

```
count = 0

for letter in 'Hello World':
    if(letter == 'l'):
        count += 1

print(count, 'letters found')
```

### Built-in functions:

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

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

```
str = 'cold'

# enumerate()
list_enumerate = list(enumerate(str))
print('list(enumerate(str) = ', list_enumerate)

#character count
print('len(str) = ', len(str))
```

#output:

```
list(enumerate(str) = [(0, 'c'), (1, 'o'), (2, 'l'), (3, 'd')])
len(str) = 4
```

### Escape Sequence:

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

```
# using triple quotes
print("""He said, "What's there?""")

# escaping single quotes
print('He said, "What\'s there?")

# escaping double quotes
print("He said, \"What's there?\")
```

List of all the escape sequence supported by Python:

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

Sometimes we may wish to ignore the escape sequences inside a string. To do this we can place **r** or **R** in front of the string.

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



### The format() Method for Formatting Strings:

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

```
# default(implicit) order
```

```
default_order = "{}, {} and {}".format('John','Bill','Sean')
```

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

```
print(default_order)
```

```
# order using positional argument
```

```
positional_order = "{1}, {0} and {2}".format('John','Bill','Sean')
```

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

```
print(positional_order)
```

```
# order using keyword argument
```

```
keyword_order = "{s}, {b} and {j}".format(j='John',b='Bill',s='Sean')
```

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

```
print(keyword_order)
```

### Set:

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

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

It can have any number of items and they may be of different types (integer, float, tuple, string etc.). But **a set cannot have a mutable element, like list, set or dictionary, as its element.**

```
# set of integers
```

```
my_set = {1, 2, 3}
```

```
print(my_set)
```

```
# set of mixed datatypes

my_set = {1.0, "Hello", (1, 2, 3)}

print(my_set)
```

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

```
# initialize a with {}

a = {}

# check data type of a

# Output: <class 'dict'>

print(type(a))

# initialize a with set()

a = set()

# check data type of a

# Output: <class 'set'>

print(type(a))
```

### changing a set:

Sets are mutable. But since they are unordered, indexing have no meaning.

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

We can add single element using the **add()** method and multiple elements using the **update()** method. The update() method can take tuples, lists, strings or other sets as its argument. **In all cases, duplicates are avoided.**

```
my_set = {1,3}

# add an element

# Output: {1, 2, 3}

my_set.add(2)

print(my_set)

# add multiple elements

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

```
my_set.update([2,3,4])
print(my_set)
```

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

### Removing elements from a set:

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

The only difference between the two is that, while using **discard()** if the item does not exist in the set, it remains unchanged. But **remove()** will raise an error in such condition.

```
# initialize my_set
my_set = {1, 3, 4, 5, 6}
print(my_set)
```

```
# discard an element
# Output: {1, 3, 5, 6}
my_set.discard(4)
print(my_set)
```

```
# remove an element
# Output: {1, 3, 5}
my_set.remove(6)
print(my_set)
```

```
# discard an element not present in my_set
# Output: {1, 3, 5}
my_set.discard(2)
print(my_set)
```

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

Set being unordered, there is no way of determining which item will be popped. It is completely arbitrary.

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

```
# initialize my_set
```

```
# Output: set of unique elements
```

```
my_set = set("HelloWorld")
```

```
print(my_set)
```

```
# pop an element
```

```
# Output: random element
```

```
print(my_set.pop())
```

```
# pop another element
```

```
# Output: random element
```

```
my_set.pop()
```

```
print(my_set)
```

```
# clear my_set
```

```
#Output: set()
```

```
my_set.clear()
```

```
print(my_set)
```

## **Set Operations:**

### **Set Union:**

```
# initialize A and B
```

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

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

```
# use | operator  
# Output: {1, 2, 3, 4, 5, 6, 7, 8}  
print(A | B)
```

```
# use union function  
>>> A.union(B)  
{1, 2, 3, 4, 5, 6, 7, 8}  
# use union function on B  
>>> B.union(A)  
{1, 2, 3, 4, 5, 6, 7, 8}
```

### Set Intersection:

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

```
# initialize A and B
```

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

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

```
# use & operator
```

```
# Output: {4, 5}
```

```
print(A & B)
```

```
# use intersection function on A  
>>> A.intersection(B)  
{4, 5}  
# use intersection function on B  
>>> B.intersection(A)  
{4, 5}
```

### Set Difference:

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

```
# initialize A and B
```

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

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

```
# use - operator on A
```

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

```
print(A - B)
```

```
# use difference function on A
```

```
>>> A.difference(B)
```

```
{1, 2, 3}
```

```
# use - operator on B
```

```
>>> B - A
```

```
{8, 6, 7}
```

```
# use difference function on B
```

```
>>> B.difference(A)
```

```
{8, 6, 7}
```

### Set Symmetric Difference:

Symmetric Difference of **A** and **B** is a set of elements in both A and B except those that are common in both.

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

```
# initialize A and B
```

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

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

```
# use ^ operator
```

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

```
print(A ^ B)
```

```
# use symmetric_difference function on A
>>> A.symmetric_difference(B)
{1, 2, 3, 6, 7, 8}
# use symmetric_difference function on B
>>> B.symmetric_difference(A)
{1, 2, 3, 6, 7, 8}
```

### Iterating Through a Set:

```
>>> for letter in set("apple"):
...     print(letter)
...
a
p
e
l
```

### Dictionary:

#### Change or add elements in a dictionary:

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

If the key is already present, value gets updated, else a new key: value pair is added to the dictionary.

```
my_dict = {'name': 'Jack', 'age': 26}
```

```
# update value
```

```
my_dict['age'] = 27
```

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

```
print(my_dict)
```

```
# add item

my_dict['address'] = 'Downtown'

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

print(my_dict)
```

### Delete or remove elements from a dictionary:

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

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

```
# create a dictionary

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

# remove a particular item

# Output: 16

print(squares.pop(4))

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

print(squares)

# remove an arbitrary item

# Output: (1, 1)

print(squares.popitem())

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

print(squares)

# delete a particular item

del squares[5]

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

print(squares)

# remove all items

squares.clear()

# Output: {}
```



```
print(squares)

# delete the dictionary itself

del squares
```

### Dictionary Comprehension:

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

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

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

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

print(squares)
```

This code is equivalent to:

```
squares = {}
for x in range(6):
    squares[x] = x*x
```

### Arrays:

Arrays are fundamental part of most programming languages. It is the collection of elements of a single data type, eg. array of **int**, array of **string**.

However, in Python, there is no native array data structure. So, we use Python lists instead of an array.

Unlike arrays, a single list can store elements of any data type and does everything an array does. We can store an integer, a float and a string inside the same list. So, it is more flexible to work with.

**[10, 20, 30, 40, 50]** is an example of what an array would look like in Python, but it is actually a list.

### Multidimensional arrays:

A multidimensional array is an array within an array. This means an array holds different arrays inside it

```
multd = [[1,2], [3,4], [5,6], [7,8]]
print(multd[0])
```

```
print(multd[3])
print(multd[2][1])
print(multd[3][0])
```

When we run the above program, the output will be:

```
[1, 2]
```

```
[7, 8]
```

```
6
```

```
7
```

Matrix: In python, matrix is a nested list. A list is created by placing all the items (elements) inside a square bracket [ ], separated by commas.

```
# a is 2-D matrix with integers
a = [['Roy',80,75,85,90,95],
      ['John',75,80,75,85,100],
      ['Dave',80,80,80,90,95]]

#b is a nested list but not a matrix
b= [['Roy',80,75,85,90,95],
     ['John',75,80,75],
     ['Dave',80,80,80,90,95]]
```

In the above examples **a** is a matrix as well as nested list where as **b** is a nested list but not a matrix.

**matrix using numpy:**

```
from numpy import *
x = range(16)
x = reshape(x,(4,4))

print(x)
```

## Accessing elements in a matrix:

### List Index

Similar to list we can access elements of a matrix by using square brackets [] after the variable like a[row][col].

```
# a is 2-D matrix with integers
a = [['Roy',80,75,85,90,95],
      ['John',75,80,75,85,100],
      ['Dave',80,80,80,90,95]]
print(a[0])
print(a[0][1])
print(a[1][2])
```

output:

```
['Roy', 80, 75, 85, 90, 95]
```

```
80
```

```
80
```

Slicing a matrix in python using colon(:) and numpy:

```
from numpy import *
a = array(['Roy',80,75,85,90,95],
          ['John',75,80,75,85,100],
          ['Dave',80,80,80,90,95]])
print(a[:3,[0,1]])
```

Output:

```
[['Roy',80],
```

```
 ['John',75],
```

```
['Dave', 80]]
```

### List Comprehension vs For Loop:

Example 1: Iterating through a string Using for Loop

```
h_letters = []  
for letter in 'human':  
    h_letters.append(letter)  
print(h_letters)
```

Output:

```
['h', 'u', 'm', 'a', 'n']
```

However, Python has an easier way to solve this issue using List Comprehension.

```
h_letters = [ letter for letter in 'human' ]  
print( h_letters)
```


Output:

```
['h', 'u', 'm', 'a', 'n']
```

### Syntax of List Comprehension:

```
[expression for item in list]
```

[expression for item in list]  
[letter for letter in 'human']



**Note:** However, not every loop can be rewritten as list comprehension.

### List Comprehensions vs Lambda functions:

#### Using Lambda functions inside List:

```
h_letters = list(map(lambda x: x, 'human'))
```

output:

```
['h', 'u', 'm', 'a', 'n']
```

### Conditionals in List Comprehension:

#### Using if with List Comprehension:

```
number_list = [ x for x in range(20) if x % 2 == 0]  
print(number_list)
```

output:

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

### Nested IF with List Comprehension:

```
num_list = [y for y in range(100) if y % 2 == 0 if y % 5 == 0]  
print(num_list)
```

output:

```
[0, 10, 20, 30, 40, 50, 60, 70, 80, 90]
```

**if...else With List Comprehension:**

```
obj = ["Even" if i%2==0 else "Odd" for i in range(10)]  
print(obj)
```

output:

```
['Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even', 'Odd', 'Even',  
'Odd']
```

**Key Points to Remember:**

- List comprehension is an elegant way to define and create lists based on existing lists.
- List comprehension is generally more compact and faster than normal functions and loops for creating list.
- However, we should avoid writing very long list comprehensions in one line to ensure that code is user-friendly.
- Remember, every list comprehension can be rewritten in for loop, but every for loop can't be rewritten in the form of list comprehension.

**Iterators:**

We use the **next()** function to manually iterate through all the items of an iterator. When we reach the end and there is no more data to be returned, it will raise **StopIteration**. Following is an example.

```
my_list = [4, 7, 0, 3]
```

```
# get an iterator using iter()
```

```
my_iter = iter(my_list)
```

```
#prints 4
```

```
print(next(my_iter))
```

```
#prints 7
```

```
print(next(my_iter))
```

```
#prints 0
print(my_iter.__next__())

#prints 3
print(my_iter.__next__())

## This will raise error, no items left
next(my_iter)
```

A more elegant way of automatically iterating is by using the for loop.

```
>>> for element in my_list:
...     print(element)
...
4
7
0
3
```

The advantage of using iterators is that they save resources. Like shown above, we could get all the odd numbers without storing the entire number system in memory. We can have infinite items (theoretically) in finite memory.

### Generators:

Python generators are a simple way of creating iterators. A generator is a function that returns an object (iterator) which we can iterate over (one value at a time).

### How to create a generator?

It is fairly simple to create a generator in Python. It is as easy as defining a normal function with **yield** statement instead of a **return** statement.

If a function contains at least one **yield** statement (it may contain other yield or return statements), it becomes a generator function. Both yield and return will return some value from a function.

The difference is that, while a **return** statement terminates a function entirely, **yield** statement pauses the function saving all its states and later continues from there on successive calls.

### Differences between Generator function and a Normal function:

- Generator function contains one or more **yield** statement.
- When called, it returns an object (iterator) but does not start execution immediately.
- Methods like `__iter__()` and `__next__()` are implemented automatically. So we can iterate through the items using `next()`.
- Once the function yields, the function is paused and the control is transferred to the caller.
- Local variables and their states are remembered between successive calls.
- Finally, when the function terminates, **StopIteration** is raised automatically on further calls.

# A simple generator function

```
def my_gen():
```

```
    n = 1
```

```
    print('This is printed first')
```

```
    # Generator function contains yield statements
```

```
    yield n
```

```
    n += 1
```

```
    print('This is printed second')
```

```
    yield n
```

```
    n += 1
```

```
    print('This is printed at last')
```

```
    yield n
```

Output:

```
>>> # It returns an object but does not start execution immediately.
```

```
>>> a = my_gen()
```

```
>>> # We can iterate through the items using next().
```

```
>>> next(a)
```

```
This is printed first
```

```
1
```

```
>>> # Once the function yields, the function is paused and the control is transferred to the caller.
```



```

>>> # Local variables and theirs states are remembered between successive
calls.
>>> next(a)
This is printed second
2
>>> next(a)
This is printed at last
3
>>> # Finally, when the function terminates, StopIteration is raised
automatically on further calls.
>>> next(a)
Traceback (most recent call last):

```

**Note:** The value of variable `n` is remembered between each call.

Unlike normal functions, the local variables are not destroyed when the function yields. Furthermore, the generator object can be iterated only once.

To restart the process we need to create another generator object using something like `a = my_gen()`.

### Generators with for loops:

# A simple generator function

```
def my_gen():
```

```
    n = 1
```

```
    print('This is printed first')
```

```
    # Generator function contains yield statements
```

```
    yield n
```

```
    n += 1
```

```
    print('This is printed second')
```

```
    yield n
```

```
    n += 1
```

```
    print('This is printed at last')
```

```
    yield n
```

```
# Using for loop
for item in my_gen():
    print(item)
```

Output:

```
This is printed first

1

This is printed second

2

This is printed at last

3
```

### **Generators with a Loop:**

The above example is to just get a feel about how generators work with 'for' loop.

Let's take an example of a generator that reverses a string:

```
def rev_str(my_str):
    length = len(my_str)
    for i in range(length - 1,-1,-1):
        yield my_str[i]
```

```
# For loop to reverse the string
```

```
# Output:
```

```
# o
```

```
# l
```

```

# l
# e
# h
for char in rev_str("hello"):
    print(char)

```

Here, we use **range()** function to get the index in reverse order using the for loop.

### Generator Expression:

The syntax for generator expression is similar to that of a list comprehension in Python. But the square brackets are replaced with round parentheses.

The major difference between a list comprehension and a generator expression is that while list comprehension produces the entire list, generator expression produces one item at a time.

They are kind of lazy, producing items only when asked for. For this reason, a generator expression is much more memory efficient than an equivalent list comprehension.

```

# Initialize the list
my_list = [1, 3, 6, 10]

# square each term using list comprehension
# Output: [1, 9, 36, 100]
[x**2 for x in my_list]

# same thing can be done using generator expression
# Output: <generator object <genexpr> at 0x0000000002EBDAF8>
(x**2 for x in my_list)

```

We can see above that the generator expression did not produce the required result immediately. Instead, it returned a generator object which produces items on demand.

```

# Initialize the list

```

```

my_list = [1, 3, 6, 10]
a = (x**2 for x in my_list)
# Output: 1
print(next(a))
# Output: 9
print(next(a))
# Output: 36
print(next(a))
# Output: 100
print(next(a))
# Output: StopIteration
next(a)

```

Generator expression can be used inside functions:

```

>>> sum(x**2 for x in my_list)
146
>>> max(x**2 for x in my_list)
100

```

## Python Closures:

### Nonlocal variable in a nested function:

Before getting into what a closure is, we have to first understand what a nested function and nonlocal variable is.

```

def print_msg(msg):
    # This is the outer enclosing function

    def printer():
        # This is the nested function

        print(msg)

```

```
    printer()
# Output: Hello
print_msg("Hello")
```

We can see that the nested function **printer()** was able to access the non-local variable **msg** of the enclosing function.

### Defining a Closure Function:

What if the last line of the function **print\_msg()** returned the **printer()** function instead of calling it?

```
def print_msg(msg):
# This is the outer enclosing function

    def printer():
# This is the nested function

        print(msg)

    return printer # this got changed
```

# Now let's try calling this function.

```
# Output: Hello
another = print_msg("Hello")
another()
```

### Observations:

The **print\_msg()** function was called with the string "**Hello**" and the returned function was bound to the name **another**. On calling **another()**, the message was still remembered although we had already finished executing the **print\_msg()** function.

This technique by which some data ("**Hello**") gets attached to the code is called **closure** in Python.

Now, try running the following:

```
>>> del print_msg
>>> another()
Hello
```

```
>>> print_msg("Hello")
Traceback (most recent call last):
...
NameError: name 'print_msg' is not defined
```

When do we have a closure?

As seen from the above example, we have a closure in Python when a nested function references a value in its enclosing scope.

The criteria that must be met to create closure in Python are summarized in the following points.

- We must have a nested function (function inside a function).
- The nested function must refer to a value defined in the enclosing function.
- The enclosing function must return the nested function.

### Decorators:

Python has an interesting feature called **decorators** to add functionality to an existing code.

This is also called **metaprogramming** as a part of the program tries to modify another part of the program at compile time.

### Prerequisites for learning decorators:

```
def first(msg):
    print(msg)
first("Hello")
second = first
second("Hello")
```

#output:

Hello

Hello

When you run the code, both functions **first** and **second** gives same output. Here, the names **first** and **second** refer to the same function object.

Functions can be passed as arguments to another function. Such function that take other functions as arguments are also called **higher order functions**.

```
def inc(x):  
    return x + 1
```

```
def dec(x):  
    return x - 1
```

```
def operate(func, x):  
    result = func(x)  
    return result
```

#invoke the function:

```
>>> operate(inc, 3)  
4  
>>> operate(dec, 3)  
2
```

Also, a function can return another function:

```
def is_called():  
    def is_returned():  
        print("Hello")  
    return is_returned
```

```
new = is_called()
```

```
#Outputs "Hello"  
new()
```

Here, **is\_returned()** is a nested function which is defined and returned, each time we call **is\_called()**.

...back to Decorators:

Basically, a decorator takes in a function, adds some functionality and returns it.

```
def make_pretty(func):  
    def inner():  
        print("I got decorated")  
        func()  
    return inner
```

```
def ordinary():  
    print("I am ordinary")
```

If we run the following codes,

```
>>> ordinary()  
I am ordinary  
  
>>> # let's decorate this ordinary function  
>>> pretty = make_pretty(ordinary)  
>>> pretty()  
I got decorated  
I am ordinary
```

In the example shown above, **make\_pretty()** is a decorator.

The function **ordinary()** got decorated and the returned function was given the name **pretty**.

For eg: We can see that the decorator function added some new functionality to the original function. This is similar to packing a gift. The decorator acts as a wrapper. The nature of the object that got decorated (actual gift inside) does not alter. But now, it looks pretty (since it got decorated).



Generally, we decorate a function and reassign it as,

```
ordinary = make_pretty(ordinary).
```

This is a common construct and for this reason, Python has a syntax to simplify this.

Use the `@` symbol along with the name of the decorator function and place it above the definition of the function to be decorated. For example,

```
@make_pretty
def ordinary():
    print("I am ordinary")
```

is equivalent to...

```
def ordinary():
    print("I am ordinary")
ordinary = make_pretty(ordinary)
```

### Decorating Functions with Parameters:

The above decorator was simple and it only worked with functions that did not have any parameters. What if we had functions that took in parameters like below?

```
def divide(a, b):
    return a/b
```

This function has two parameters, **a** and **b**. We know, it will give error if we pass in **b** as **0**.

```
>>> divide(2,5)
0.4
>>> divide(2,0)
Traceback (most recent call last):
...
ZeroDivisionError: division by zero
```

Now let's make a decorator to check for this case that will cause the error.

```
def smart_divide(func):  
    def inner(a,b):  
        print("I am going to divide",a,"and",b)  
        if b == 0:  
            print("Whoops! cannot divide")  
            return  
        return func(a,b)  
    return inner  
  
@smart_divide  
def divide(a,b):  
    return a/b
```

```
>>> divide(2,5)  
I am going to divide 2 and 5  
0.4  
  
>>> divide(2,0)  
I am going to divide 2 and 0  
Whoops! cannot divide
```

Note: Parameters of the nested **inner()** function inside the decorator is same as the parameters of functions it decorates.

### Copy an Object in Python:

In Python, we use = operator to create a copy of an object. You may think that this creates a new object; it doesn't. It only creates a new variable that shares the reference of the original object.

```
old_list = [[1, 2, 3], [4, 5, 6], [7, 8, 'a']]
new_list = old_list
new_list[2][2] = 9
print('Old List:', old_list)
print('ID of Old List:', id(old_list))
print('New List:', new_list)
print('ID of New List:', id(new_list))
```

Output:

```
Old List: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

ID of Old List: 140673303268168

New List: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

ID of New List: 140673303268168
```

As you can see from the output both variables **old\_list** and **new\_list** shares the same id. So, if you want to modify any values in **new\_list** or **old\_list**, the change is visible in both.

Sometimes you may want to have the original values unchanged and only modify the new values or vice versa. So, we have 2 ways to create copies.

- Shallow Copy
- Deep Copy

### **Shallow Copy:**

A shallow copy creates a new object which stores the reference of the original elements. So, a shallow copy doesn't create a copy of nested objects, instead it just copies the reference of nested objects. This means, a copy process does not recurse or create copies of nested objects itself.

```
import copy
old_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
new_list = copy.copy(old_list)
print("Old list:", old_list)
print("New list:", new_list)
```

output:

```
Old list: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

```
New list: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

In above program, we created a nested list and then shallow copy it using **copy()** method.

This means it will create new and independent object with same content. To verify this, we print the both **old\_list** and **new\_list**.

To confirm that **new\_list** is different from **old\_list**, we try to add new nested object to original and check it.

```
import copy
```

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

```
new_list = copy.copy(old_list)
```

```
old_list.append([4, 4, 4])
```

```
print("Old list:", old_list)
```

```
print("New list:", new_list)
```

output:

```
Old list: [[1, 1, 1], [2, 2, 2], [3, 3, 3], [4, 4, 4]]
```

```
New list: [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
```

In the above program, we created a shallow copy of **old\_list**. The **new\_list** contains references to original nested objects stored in **old\_list**. Then we add the new list i.e [4, 4, 4] into **old\_list**. This new sublist was not copied in **new\_list**.

Note: However, when you change any nested objects in **old\_list**, the changes appear in **new\_list**.

```
import copy
old_list = [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
new_list = copy.copy(old_list)
old_list[1][1] = 'AA'
print("Old list:", old_list)
print("New list:", new_list)
```

output:

```
Old list: [[1, 1, 1], [2, 'AA', 2], [3, 3, 3]]
New list: [[1, 1, 1], [2, 'AA', 2], [3, 3, 3]]
```

### Deep Copy:

A deep copy creates a new object and recursively adds the copies of nested objects present in the original elements.

```
import copy
old_list = [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
new_list = copy.deepcopy(old_list)
print("Old list:", old_list)
print("New list:", new_list)
```

output:

```
Old list: [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
New list: [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
```

Note: If you make changes to any nested objects in original object **old\_list**, you'll see no changes to the copy **new\_list**.

```
import copy
old_list = [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
new_list = copy.deepcopy(old_list)
old_list[1][0] = 'BB'
print("Old list:", old_list)
print("New list:", new_list)
```

Output:

```
Old list: [[1, 1, 1], ['BB', 2, 2], [3, 3, 3]]

New list: [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
```

### **Numpy:**

A numpy array is a grid of values, all of the same type, and is indexed by a tuple of nonnegative integers. The number of dimensions is the rank of the array; the shape of an array is a tuple of integers giving the size of the array along each dimension

#### **Creating numpy array:**

One of the limitations of NumPy is that all the elements in an array have to be of the same type.

Empty array

Random array

Numpy for reading csv

Indexing numpy arrays

Assigning Values To NumPy Arrays

N-Dimensional NumPy Arrays.

### **NumPy Data Types:**

As we mentioned earlier, each NumPy array can store elements of a single data type. For example, wines contains only float values. NumPy stores values using its own data types, which are distinct from

Python types like float and str. This is because the core of NumPy is written in a programming language called C, which stores data differently than the Python data types. NumPy data types map between Python and C, allowing us to use NumPy arrays without any conversion hitches.

### Converting Data Types:

You can use the **numpy.ndarray.astype** method to convert an array to a different type. The method will actually copy the array, and return a new array with the specified data type.

**Note:** We used the Python int type instead of a NumPy data type when converting wines. This is because several Python data types, including float, int, and string, can be used with NumPy, and are automatically converted to NumPy data types.

### NumPy Array Operations:

NumPy makes it simple to perform mathematical operations on arrays. This is one of the primary advantages of NumPy, and makes it quite easy to do computations.

Single Array Math

Multiple Array Math

### Broadcasting:

The term broadcasting describes how numpy treats arrays with different shapes during arithmetic operations. Subject to certain constraints, the smaller array is “broadcast” across the larger array so that they have compatible shapes

**Disadv:** Sometimes broadcasting is a bad idea because it leads to inefficient use of memory that slows computation.

### General Broadcasting Rules:

When operating on two arrays, NumPy compares their shapes element-wise. It starts with the trailing dimensions, and works its way forward. Two dimensions are compatible when

- they are equal, or
- one of them is 1

### NumPy Array Methods:

In addition to the common mathematical operations, NumPy also has several methods that you can use for more complex calculations on arrays. Eg. Aggregate along the axis, etc...

### **NumPy Array Comparisons:**

NumPy makes it possible to test to see if rows match certain values using mathematical comparison operations like `<`, `>`, `>=`, `<=`, and `==`.

### **Subsetting:**

One of the powerful things we can do with a Boolean array and a NumPy array is select only certain rows or columns in the NumPy array.

### **Reshaping NumPy Arrays:**

We can change the shape of arrays while still preserving all of their elements. This often can make it easier to access array elements. The simplest reshaping is to flip the axes, so rows become columns, and vice versa. We can accomplish this with the `numpy.transpose` function.

### **Combining NumPy Arrays:**

With NumPy, it's very common to combine multiple arrays into a single unified array. We can use `numpy.vstack` to vertically stack multiple arrays. Think of it like the second arrays's items being added as new rows to the first array.

- `Vstack`
- `Hstack`
- `Concatenate`

### **Pandas:**

Pandas is an open source software library which is built on top of NumPy.

Pandas is used for data manipulation, analysis and cleaning. Python pandas is well suited for different kinds of data, such as, but not limited to:

- Tabular data with heterogeneously-typed columns
- Ordered and unordered time series data
- Unlabelled data
- Any other form of observational or statistical data sets

How to install Pandas:



To install Python Pandas, go to your command line/ terminal and type “pip install pandas” or else, if you have anaconda installed in your system, just type in “conda install pandas”. Once the installation is completed, go to your IDE (Jupyter, PyCharm etc.) and simply import it by typing: “import pandas as pd”.