Python

Author: Mahdis Bagheri

Based on lectures by: Howsam Academy

More details to learn on w3school

More examples and tutorials on exercism

Index

- **Section 1: variables**

Variables are containers used to store data values. You don't need to set their type explicitly; Python figures it out automatically based on the value you assign.

```
a = 2 #int
b = 2. #float
c = True #bool
d = "hi" #str
type(a)
```

```
int
```

- **How to define a variable?** Defining a variable has three elements: the variable name, the assignment operator, and the value.
- **Variable types:** int, float, bool, string
  You can use " "or ' ' to define a string
- **How to call a variable?**
- **How to check a variable type?**

```
a = 2.3
type(a)
```

```
float
```

```
a = 2
print(a)
#or
a
```

```
2
```

```
#how to see python's keywords?
help('keywords')
```

```
Here is a list of the Python keywords.  Enter any keyword to get more help.

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

- **Variables Key points:**

1. Variable names should start with a letter or underscore
2. Variable names contain letters, digits, and underscores.
3. Python is dynamically typed, allowing variables to change type.
4. Python is case sensitive (A ≠a)
5. Variable names cannot be Python keywords.

- **Variables type conversion**
  - Python truncates[1](rounds down) floating-point numbers when converting them to integers. **Data loss** happens because the fractional part is discarded.
  - When converting numbers to Boolean, any non-zero number (even negative ones) is converted to True and only zero is converted to False

```
a = 2.3
print(int(a), bool(a))
str(a)
```

```
2 True
'2.3'
```

```
b = 2.9
print(int(a), bool(b))
```

```
2 True
```

```
c = 4
float(c)
```

```
4.0
```

Tip: each variable is an object and it enables some features (objects methods or functions)

```
#an integer with value 0.
a = int()
a
```

```
0
```

```
#a floating point with value 0.
b = float()
b
```

```
0.0
```

```
# False
c = bool()
c
```

```
False
```

```
#an empty string
d = str()
d
```

```
''
```

- **Python style guide:** The primary Python style guide is **PEP 8** (Python Enhancement Proposal 8). It provides conventions for writing clean, readable, and consistent Python code.
  - Choose proper names for variables,
  - Capitalize class names (e.g., AreaCircle).
  - Use space to enhance code readability

[1] به پایین گرد میکند

- section 2: Mathematical operations
  - Addition +
  - Subtraction -
  - Multiplication *
  - Division / With / the remainder can be a floating point, but // gives an integer remainder.
  - Power **
  - Remainder or Modulus %
  - Quotient or Floor division //

```
#You cannot add or subtract a string with a number.
#to perform mathematical operations, the variables must be of the same type
a = 4
b = 2.2
c = True
d = False
e = "hel"
f = "lo"
g = True
print (a + b , a + c, a+d , e + f, c + g, a-d, a-c, sep = ",")
```

```
6.2,5,4,hello,2,4,3
```

```
a =  6
print(a / 4 , a //4 , a %4)
```

```
1.5 1 2
```

```
a = 6
a ** 2
```

```
36
```

- Operator precedence
  1. Parentheses ()
  2. Exponentiation **
  3. Unary positive/negative(sign) +x, -x
  4. Multiplication, Division, Floor division, Modulus *, /, //, %
  5. Addition and Subtraction +, -
  6. Comparison operators ==, !=, <, >, <=, >=
  7. Locgical operators(not, and, or)
  8. Assignment operators
- Assignment operators: assignment operators are used to assign values to variables.
  - Equal =
  - Add and equal +=
  - Subtract and equal -=
  - Multiply and equal *=
  - Divide and equal /=
  - Power and equal **=
  - Quo and equal //=
  - Remainder and equal %=

```
a = 6
a += 1   #7   a = a + 1
a -= 1   #5   a = a - 1
a *= 2   #12  a = a * 2
a /= 4   #1.5 a = a / 4
a **= 2 #36   a = a**2
a //= 4  #1   a = a // 4
a %= 4   #2   a = a % 4
```

- examples

```
#bmi
weight = 65
height = 1.6
bmi = weight / height ** 2
bmi
```

```
25.390624999999996
```

```
#calculate the volume of cylinder
p = 3.14
r = 5
height = 10
v = p*r**2*height
v
```

```
785.0
```

- Types of data structures in python

  1. **primitive(basic):** primitives contain a value in a variable. For example, **float, integer, string, Boolean**
  2. **Non primitive:** a container of items or a collection of values.
     - **Built-in:** Predefined, flexible data containers. For example, **list, tuple, dictionary, set**
     - **User-defined:** Custom classes to represent more complex or specific data models that aren't covered by built-in types. For example, **stack, queue, LinkedList, tree**

- Section 3: list [item1, item2, item3]

```
My_list = [1, 'apple', 3.14, True]
```

In Python, a list is an ordered collection of items, enclosed in brackets [], that can hold elements of various data types.

```
#defining empty list
a = [] #or
a = list()   #using keyword
a

[]
```

  - How to define a list? Lists are defined using brackets and commas.
  - How to define an empty list?
    An empty list has many uses. Sometimes, you create an empty list initially and then gradually add items to it.
    Tip: List is also a keyword which is used to convert a data type (like a string) into a list.

```
string = "howsam"
list(string)

['h', 'o', 'w', 's', 'a', 'm']
```

  - Nested list: a list inside a list. This feature allows you to define matrices in Python. Example:

```
My_list = [[1, 'apple', 3.14, True], 5 , 7]
```

  - Indexing: Indexing provides access to each item in a list and allows you to know it's value

```
a = [3,4,5]
a[0]

3
```

    There are two ways to index: positive indexing and negative indexing. (Both can be used at the same time.)

```
a = [15, 25, 30,70,80]
print(a[-1], a[-2], a[-5])
print(a[4], a[3], a[0])

80 70 15
80 70 15
```

```
a = [15, 25, 30,70,80]
a[1]*a[-1]

2000
```

    Negative indexing starts from the right side of the list (-1) and goes to the left ($-\infty$).With negative indexing, it is easier to access the elements at the end of the list.

    Positive indexing starts from the left side (0) of the list and goes to the right(∞.) With positive indexing, it is easier to access the elements at the beginning of the list.

  - How to access elements of nested lists? Easily, With two or more indices.

```
a = [[15, 25, 30],70,80]
a[0][1]

25
```

- Slicing: Slicing extracts a specific part of a list by specifying a **start, stop, and step** indices. You can use negative or positive indexing or both. my_list [start: stop: step]. Step slicing simplifies access to elements at regular intervals, such as even or odd positions.

```
a = [100, 200, 300, 400, 500, 600, 700, 800, 900]
a[5:9]  #[600, 700, 800, 900]
a[:-5]   #[100, 200, 300, 400]
a[7:] #[800, 900]
a[::2] #[100, 300, 500, 700, 900]
a[1::2] #[200, 400, 600, 800]
```

a[5:9] = a[5], a[6], a[7], a[8]= a[5,9)
a[:-5] = from beginning to a[-6]
a[7:] = from a[7] to the end = open end

  - How to replace values in a list?

```
a = [100, 200, 300, 400, 500, 600, 700, 800, 900]
a[2] = 330
a[3::2] = [450,650,850]
a

[100, 200, 330, 450, 500, 650, 700, 850, 900]
```

```
a = [5,6,10,6]
a.remove(6)
a

[5, 10, 6]
```

- How to create matrices using lists? You can represent a matrix with a nested list, where each inner list is a row. However, libraries like Numpy offer more efficient matrix representations.

```
#method1
row1 = [1,2,0,7]
row2 = [9,4,8,1]
row3 = [5,3,1,6]
A = [row1,row2,row3]
#method2
A = [[1,2,0,7],
     [9,4,8,1],
     [5,3,1,6]]
A[0][1]

2
```

- List methods(functions): List functions are accessed using dot notation and pressing Tab

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the first item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

```
a = [5,6,10,1]
a.pop(3)
a
```
```
[5, 6, 10]
```

```
a = [5,6]
b = a.copy()
b[1] = 9
print(a,b)
```
```
[5, 6] [5, 9]
```

```
a = [5,6,7,5,7]
a.index(7)
```
```
2
```

```
a = [5,6,7]
b = [8,9,10]
a.extend(b)
a
```
```
[5, 6, 7, 8, 9, 10]
```

```
a = [5,6,7]
a.extend([8,9,10])
a
```
```
[5, 6, 7, 8, 9, 10]
```

```
a = [5,6,7,5,1]
a.sort()
a
```
```
[1, 5, 5, 6, 7]
```

```
a = [5,6,7,5,1]
a.sort(reverse=True)
a
```
```
[7, 6, 5, 5, 1]
```

```
a = [5,6,10,1]
a.reverse()
a
```
```
[1, 10, 6, 5]
```

```
a = [5,6,7]
a.append(8)
a
```
```
[5, 6, 7, 8]
```

```
a = [5,6,7]
a.clear()
a
```
```
[]
```

```
a = [5,6,10,1]
a.insert(2,"howsam")
a
```
```
[5, 6, 'howsam', 10, 1]
```

```
a = [5,6,7,5,7]
a.count(5)
```
```
2
```

```
a = [5,6,7]
a.__len__() #or len(a)
```
```
3
```

```
a = [5,6,7]
len(a)
```
```
3
```

Tip: To create a true list backup, use the copy function. This ensures changes to the backup or the original list don't affect each other. Simply assigning one list to another (e.g., b = a) creates a reference, not a copy, so modifications to either variable will impact both.

- Tips and tricks
  - Concatenate two lists without using extend
  - Repeat the list
  - Check if a value exists in a list

```
a = [5,6]
b = [7,8]
a + b
```
```
[5, 6, 7, 8]
```

```
a = [5,6]
3 * a
```
```
[5, 6, 5, 6, 5, 6]
```

```
a = [5,6,8,9]
10 in a
```
```
False
```

- List characteristics
  1. **Lists are ordered** [1,2,3] ≠[3,2,1]
  2. **List items are accessed by indexing/slicing**
  3. **Lists are dynamic:** list length can be decreased or increased.
  4. **Lists are heterogenous**[1]**,**allowing items of different types (e.g., strings and integers) within the same list.
  5. **Lists are nestable:** a list item can be a list itself.
  6. **Lists are mutable:** You can change each item's value

Example: Assign the following text to the variable 'text'. then, split the text into a collection of words using .split() then count the occurrences of "NLP".

Tip1: use ‘’’ ‘’’ to convert a text into a string. (especially when the text includes vocabularies with “ “ or ‘ ‘)

Tip2: Split is a string method, here, using .split( ), text will be split based on the spaces between vocabularies. Therefore, each word will be turned into a string, and the output will be a **list** of these strings.

However, the problem with the code is that it won't count(NLP) or "NLP" or NLP? and so on.

```
text = ''' hi, hello howsam 'hi' hellow '''
text.split()
```
```
['hi,', 'hello', 'howsam', "'hi'", 'hellow']
```

```
text =""" Neuro-linguistic programmin
words = text.split()
words.count("NLP")
```
```
8
```

[1] غیرهمگن

- **Section 4: Logical operators (and, or, not)**

```
a = True
b = False
print(a and b, a or b, not a)
```
```
False True False
```

```
#nor
a = True
b = False
not(a or b)
```
```
False
```

```
#nand
a = True
b = False
not(a and b)
```
```
True
```

In logical operations, inputs must be logical.

- **Comparison operators**

The comparison result is a Boolean value (True or False).

```
a = "howsam"
b = "howsam"
a == b
```
```
True
```

```
a = 10
b = 0
a == b
```
```
False
```

```
a = "HOWSAM"
b = "SARA"
a < b
```
```
True
```

  - **Equal to** a==b
  - **Not Equal to** a!=b
  - **Less than** a<b
  - **Less than or Equal to** a<=b
  - **Greater than** a>b
  - **Greater than or equal to** a>=b

Numerical variables and strings can be compared for equality or inequality. The last 4 operators are mostly used in numerical variable comparisons. But you can use them to compare strings, too. (Not common) in this case: A<B<C<...<Z<a<b<c<...<z

  - **Comparison of lists**

```
list1 = [1, 2, 3]
list2 = [1, 2, 3]
print(list1 == list2)  # Output: True
```

Equal comparison returns True if both lists have the same elements in the same order, and False otherwise.

Other comparison operators can be used with lists, but it's uncommon and wrong.

```
[1,2,0] < [1,3]
```
```
True
```
```
[1,2] > [1,2,2]
```
```
False
```

  - **Comparison for floating-point numbers**

```
a = 2.2 + 1.2
b = 3.4
a == b
```
```
False
```

```
c = 1.2+0.5
d = 1.7
c == d
```
```
True
```

why a != b ?

When the result of an operation is a floating-point number, due to **floating-point nature and the limited number of storage bits**, the result numbers may sometimes be rounded.

```
a = 2.2 + 1.2
a
```
```
3.4000000000000004
```

Therefore, this is not a suitable method for comparing floating-point numbers.

The correct way to compare floating-point numbers (a and b) resulting from operations:

```
abs(a-b) < 1e-5 #abs(a-b) < 10**-5
```
```
True
```

this means: If the absolute value of the a-b is smaller than 1e-5, then we are sure that these two numbers are pretty close to each other and are approximately equal.

Tip: 1000 = 1e3 ۱۰ به نمای ۳

0.001 = 1e-3 ۱۰ به نمای-۳

0.002 = 2e-3 ۲۰ به نمای ۳-

  - **Combine math, comparison and logical operators**

```
c = 6
d = 2*3
e = "a"
f = "a"
not(c == 2*3 and e == f)
```
```
False
```

  - **Chained comparison**

In Python, chained comparisons like a < b < c are a special feature that makes multiple comparisons concise and more readable.

```
a = 20
b = 50
c = 30
a<c<b #in matlab you write  a<c and c<b
```
```
True
```

- **any() , all()** : belongs to built-in commands, useful in writing conditions

- **any()** checks if **any** element in an iterable (like a list or tuple) is true. It returns True if **at least one** element is true; otherwise, False.
- **all()** checks if **all** elements in an iterable are true. It returns True only if **every** element is true; otherwise, False.

These functions are commonly used with iterables containing boolean values or expressions that result in boolean values, such as comparison operations. They can also be used with numbers, but it's not common. (Then Zero = False, Nonzero = True.)

- EXAMPLES

```
#How to implement X.OR ? a xor b
a = True
b = True
(not a and b) or (a and not b)
```

```
False
```

```
#Define a variable named a with the value 2.5 and then check whether it's a floating point.
a = 2.5
type(a) == float
```

```
True
```

```
#the math scores of students in a class are as follows:
names = ["hana" , "karter", "hector" , "john"]
scores = [68,83,70,83]
```

```
#what is hector's score?
#names.index("hector") #2
scores[names.index("hector")]
```

```
70
```

```
#are john and karter's scores the same?
scores[names.index("john")] == scores[names.index("karter")]
```

```
True
```

- Section 5: Conditional statements

Conditions use mathematical, logical, and comparison operations to execute different code blocks based on a true or false result. If a condition evaluates to true, its corresponding code block is executed; otherwise, the code block is skipped. The indentation before the statement is one tab character (Correctly written "if" condition automatically create the proper indentation.)

- If condition :

  || Statement

```
text = "hellow world!   hellow world!   hellow world!"
if len(text) >10:
    print("above 10")
```

```
above 10
```

```
#if grade is greater than or equal to 50, then display "passed"
#and if it's less than 50, display "failed
grade = 30
if grade >= 50:
    print("passed")
if grade < 50:
    print("failed")
```

```
failed
```

- If condition :

  || Statement

  else:

  || Statement

```
x = -5
if x>= 0 :
    print("positive")
else:
    print("negative")
```

```
negative
```

- elif = else if = ... در غیر این صورت، اگر

  If condition1 :

  || Statement 1

  elif condition2 :

  || Statement 2

  ...

  elif condition 100 :

  || Statement 100

  else:

  || Statement 101

```
x = -5
if x>0:
    print("positive")
elif x <0:
    print("negative")
else:
    print("zero")
```

```
negative
```

```
#implement the following function
#x<-4  --> y=13
#-4 <= x <10  --> y= x-6
#x >= 10 --> y= -9x
x = 2
if x<-4 :
    y = 13
elif -4 <= x <10:
    y = x-6
else :
    y = -9*x
print(x,"-->",y)
```

```
2 --> -4
```

- Nested if

```
#x is positive and even or positive and odd or negative and...?
x = -6
if x>0:
    if x%2 == 0:
        print("x is positive and even")
    else:
        print("x is positive and odd")
elif x<0:
    if x%2 == 0:
        print("x is negative and even")
    else:
        print("x is negative and odd")
else:
    print("x is 0")
```

```
x is negative and even
```

- Conditional statements in one line

Statement1 If condition else Statement2

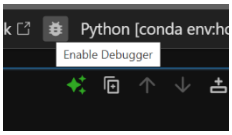
One difference in writing a one-line statement is that you assign the whole line to y one time at the end.

```
x=5
t=2
#if  x<=t:
#    y=0
#else :
#    y = x-t
y = 0 if x<=t else x-t  #one-line method
```

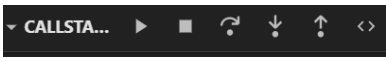
- Debug (finding and fixing bugs in the code) & trace (monitoring the flow of the program)

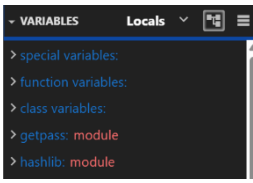
  Debugging often involves using tools like debuggers to identify and fix the errors. **Tracing** refers to monitoring the execution flow of your code. This helps understand how a program runs, which functions are called, and in what order. **One way to debug is to trace. If you run the code line by line, then you'll figure out the whereabout of the problem and why.**

  - How to trace in Jupiter lab?



1. **Enable Debug Mode:** Click on the **"enable debugging"** button at the top of JupyterLab; numbers will appear alongside code lines.
2. **Set Breakpoints:** click the line number you want to start tracing from to place a red circle (breakpoint).
3. **Run the Cell:** Start execution of your code by running the cell so it will stop at the breakpoints.
4. **Use Debug Controls:**
   - ▶ **Continue (F9):** Runs code until the next breakpoint.
   - → **Next (F10):** Executes the current line and pauses at the next.
   - ■ **Stop:** Ends debugging.
5. **Inspect Variables:** View current variable values in the variable box to understand the state.
6. **Finish Debugging:** After completing, disable debugging mode to improve performance because debug mode slows your device down





for more advanced debugging, tools like **PyCharm** are often used.

Tip: A one-line IF cannot be traced.

- Examples

```
#beaufort scale(b) (is an scale for wind pace) is an integer between 0 and 12.#if b is 0, then print "there is no wind"
#if b is between 1 and 6,then print "there is a breeze" #if b is between 7 and 9, then print "there is a gale"
#if b is between 10 and 11, then print "there is a storm" #if b is between 10 and 11, then print "there is a storm"
#if b is 12, then print "hellow huricane" #if b is less than 0 or beyond 12, then print "not valid "
b = 5
if b == 0:
    print("there is no wind")
elif 1<= b <=6 :
    print("there is a breeze")
elif 7<= b <=9 :
    print("there is a gale")
elif 10<= b <=11:
    print("there is a storm")
elif b==12:
    print("hellow huricane")
else :
    print("not valid ")
```

```
#compute the roots of   (a*x**2)+(b*x)+c=0
a,b,c = 1,2,1
delta = b**2 - 4*a*c
if delta >0 :
    root1 = (-b + delta**0.5) / 2*a
    root2 = (-b - delta**0.5) / 2*a
    print(root1,root2)
elif delta ==0:
    root1 = -b / (2*a)
    root2 = root1
    print(root1,root2)
else :
    print("there is no real root")
#you can simplify the code
if delta >= 0 :
    root1 = (-b + delta**0.5) / 2*a
    root2 = (-b - delta**0.5) / 2*a
    print(root1,root2)
else :
    print("there is no real root")
```

```
-1.0 -1.0
-1.0 -1.0
```

```
#Sort a, b, and c in decreasing order.
#(Assign a the greatest value, assign c the least value and assign b the middle)
a,b,c = 2,8,5
if a<b:
    a,b = b,a  # it is called swapping
if a<c:
    a,c = c,a  #at this stage, a contains the greatest value
if b<c:
    b,c = c,b
print(a,b,c)
#there are other permutations of this code
```

- Section 6: Loops (for & while)

- For loop: for loop is used when the number of repetitions is known. A for loop is used to iterate over a sequence (like a list, tuple, string, or range) and execute a block of code for each item.

for variable in sequence:
||||statements

```
a = ["salam", True, 3, 4, 5]
for item in a:
    print(item, end=" ")
    #print(item, end="\t")  It puts a Tab between each output.
    #print(item, end="\n")It puts an enter between each output.
    #print(item, end="\n\n") It puts 2 enters between each output.
```

```
for x in [1,2,3,4,5] :
    print(x, end=" ")
```
```
1 2 3 4 5
```

  - Range as sequence: The range () function generates a sequence of numbers, which is often used in for loops for iteration. range (start, stop, step) tip: steps must be integers.
    range (10) = [0,10) = 0,1,2, 3, ...,9
    range (-3,1) = [-3,1) = -3, -2, -1, 0
    range (2,10,2) = [2,10) = 2,4,6,8

```
for i in range(10):
    print(i, end = " ")
```
```
0 1 2 3 4 5 6 7 8 9
```

```
list(range(5))
```
```
[0, 1, 2, 3, 4]
```

Tip: You can use the list command to see the numbers in the range.

  - String as sequence:

```
s = "howsam"
for a in s:
    print(a, end = "_")
```
```
h_o_w_s_a_m_
```

- Nested loop:

```
mat = [[1,2,3],
       [4,5,6],
       [7,8,9]]
for r in mat:
    print(r)
```
```
[1, 2, 3]
[4, 5, 6]
[7, 8, 9]
```

```
mat = [[1,2,3],
       [4,5,6],
       [7,8,9]]
for r in mat:
    for j in r:
        print(j, end= " ")
```
```
1 2 3 4 5 6 7 8 9
```

```
#Using Four, increase the matrix values by one
matrix = [[1,2,3],
          [4,5,6],
          [7,8,9]]
for i in range(3):
    for j in range (3):
        matrix[i][j] += 1
matrix
```
```
[[2, 3, 4], [5, 6, 7], [8, 9, 10]]
```

```
#increase the matrix values by one
#the numbers of rows and columns are undetermeined
matrix = [[1,2,3],
          [4,5,6],
          [7,8,9]]
row_number = len(matrix)  #3
column_number = len(matrix[0])  #3
for i in range(row_number):
    for j in range(column_number):
        matrix[i][j] += 1
matrix
```
```
[[2, 3, 4], [5, 6, 7], [8, 9, 10]]
```

```
#print even numbers less than 30
for i in range(0,30,2) :
    print(i, end = " ")
#or
for i in range(30) :
    if i % 2 == 0:
        print(i, end = "  ")
```
```
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28
```

In a for loop, you can have more than one variable, provided that the sequence can be assigned to that number of variables. This method is called “unpacking”

```
matrix = [["ali",20],
          ["hanie",19],
          ["shahin",18]]
for name , score in matrix:
    print(name, score)
```
```
ali 20
hanie 19
shahin 18
```

- Unpacking: you can unpack multiple values in a for loop by iterating over a sequence of tuples (or other iterable collections). This allows you to assign multiple variables simultaneously, making your code cleaner and more efficient and also reduces the need for nested loops or manual unpacking inside the loop. It’s especially handy when working with data in tuples, lists, or other iterables where each element contains multiple related values. (_ means you don’t want that variable in output)

```
a = [(1,2) , (3,4)]
for _ , j in a :
    print (j)
```
```
2
4
```

- One-line for [statement for variable in sequence] It’s output would be a list

```
[i for i in [1,2,3,4,5,6]]
```
```
[1, 2, 3, 4, 5, 6]
```

Why should you enclose a one-line for loop in brackets? Because each time the loop runs, it's equivalent to adding an item to an empty list

```
#increase each item by .
numbers = [5,2,4,6,7]
numbers2 = [ i+1 for i in numbers]
numbers2
```
```
[6, 3, 5, 7, 8]
```

When you put print() inside brackets, these None values are produced.
"Tip: you can't assign variables inside a one-line for loop. You must assign the entire bracketed expression to the variable at the end."

```
y = [i**2 for i in [1,2,3,4,5,6]]
y
```

```
[1, 4, 9, 16, 25, 36]
```

- Combination of a one-line for and a condition:

```
[i for i in range(30) if i%2 ==0]
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28]
```

- While loop: A while loop keeps executing as long as a specified condition is True.

While condition :

||||||statement

```
#print numbers less than 10 using while
num = 0
while num <= 10 :
    print(num, end = " ")
    num += 1
```

```
# compute 5! use while
numf = 5
s = 1
out = 1
while s <= numf :
    out *= s
    s += 1
print(out)
```

```
120
```

- Break & continue (useful in for and while loops)

Break: Sometimes, you decide to break the loop before reaching the end of the loop (before the condition is violated). For example, there is a list and as soon as an unwanted value in that list is seen, the loop must be broken and exit.

Continue: The continue statement doesn't break the loop, but the commands after the continue statement will not be executed.

```
for i in range(10) :
    print(i, end = " ")
    if i >3 :
        break
```

```
0 1 2 3 4
```

```
for i in range (10):
    print(i, end = " " )
    print(i, end = " " )
```

```
0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9
```

```
for i in range (10):
    print(i, end = " " )
    continue
    print(i, end = " " )
```

```
0 1 2 3 4 5 6 7 8 9
```

```
#In this list, -1 is a red flag. If this number is seen, this list is unsuitable
a = [10,4,9,-1,1,2,14]
for ai in a :
    if ai == -1 :
        break
    else :
        print(ai)
```

```
10
4
9
```

```
#write a code for HOPE game
for i in range(1,30) :
    if i % 5 == 0:
        print("HOPE", )
    else :
        print (i, end = " " )
```

```
1 2 3 4 HOPE
6 7 8 9 HOPE
11 12 13 14 HOPE
16 17 18 19 HOPE
21 22 23 24 HOPE
26 27 28 29
```

```
#or
for i in range(1,30) :
    if i % 5 == 0:
        continue
    else :
        print (i, end = " " )
```

```
1 2 3 4 6 7 8 9 11 12 13 14 16 17 18 19 21 22 23 24 26 27 28 29
```

- Examples

```
#store fibonacci sequence in a list
num0 = 0
num1 = 1
fibonacci = []
fibonacci.append(num0)
fibonacci.append(num1)
for i in range(10) :
    num2 = num0 + num1
    fibonacci.append(num2)
    num0 = num1
    num1 = num2
fibonacci
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

```
#write a code to generate fibonacci
num0 = 0
num1 = 1
print(num0,num1, end = " ")
for i in range(15):
    num2 = num0+num1
    print(num2, end = " ")
    num0 = num1
    num1 = num2
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

```
#Without using the count function, how many 4s are there in this list?
a = [1,2,3,4,6,5,4,9,8,7,4]
count = 0
for i in a :
    if i == 4 :
        count += 1
count
```

```
3
```

- section 7: Dictionary

{key1: value1, key2: value2}

A dictionary has a fundamental difference from a list. In a list, items are ordered and accessible by their position (index). However, in a dictionary, items are stored as key-value pairs, where each key serves as a label to access its corresponding value.

- Keys should be immutable: Once a dictionary is defined, the keys cannot be changed. That's why a key cannot be a list. Tip1: Keys can be string, number, or tuple(immutable) but cannot be lists, sets, or dictionaries (mutable), while Values have no type restrictions.

```
a = {1: "one" , 2: "two" , 1 : "1"}
a[1]
```

```
'1'
```

- Duplicate keys are not allowed Tip2: In a dictionary, we cannot have a key with two different values. So, if we assign two values to a key, Python get the second value.
- Nested dictionaries: the value of a key, can be a dictionary itself.
- Index dictionary by key: use []

```
dict1 = {"key1" : 1 , "key2" : 2 , "key3" : [1,2,3]}
dict1["key3"][-1]
```

```
3
```

```
#index dictioanry by key
dict0 = {"ali" : 3 , "reza" : 2 }
dict1 = {"key1" : 1 , "key2" : 2 , "key3" : dict0}
print(dict1["key3"])
print(dict1["key3"]["ali"])  #-->how to acces nested dic
```

```
{'ali': 3, 'reza': 2}
3
```

```
a = {1: "one" , 2: "two" , 3 : "1"}
b = {1: a, 2:2}
b[1]
```

```
{1: 'one', 2: 'two', 3: '1'}
```

- Check if a key exists in a dictionary

```
dict1 = {"ali" : 17 , "sara" : 19 , "samin" : 20}
"ali" in dict1
```

```
True
```

- How to define an empty dictionary?

```
#empty dictionary
a = {} #or a = dict()
a
```

```
{}
```

- Dictionary does not support slicing: you cannot extract a specific part of a dictionary
- Add a key-value to dictionary: This is done through indexing.
  One other method is using update function.
- Change value of a key:
  one other method is using update function

```
#add "ali" : 20 to the dict1
dict1 = {"sara" : 15, "mamad" : 16}
dict1["ali"] = 20
dict1["sana"] = 20
dict1
```

```
{'sara': 15, 'mamad': 16, 'ali': 20, 'sana': 20}
```

```
# change the math averege to 20
scores = {"math" : 18, "sciene" : 19}
scores["math"] = 20
scores
```

```
{'math': 20, 'sciene': 19}
```

```
dict2 = {"name" : "ali", "grade" : 3, "scores" : [20,19,17]}
dict2["scores"][2] = 18
dict2
```

```
{'name': 'ali', 'grade': 3, 'scores': [20, 19, 18]}
```

- dictionary characteristics

```
{"ali" : 10 , "sara" : 15} == {"sara" : 15,"ali" : 10 }
```

```
True
```

1. Dictionaries are unordered
2. Dictionary items are accessed by keys. (Not indices)
3. Dictionaries are dynamic: dictionary's length can be decreased or increased.
4. Dictionaries are heterogenous: Keys can be strings, numbers, or tuples. Values have no type restrictions.
5. Dictionaries are nestable: a value of a key can be a dictionary.
6. Dictionaries are mutable: you can change a value of a key

- **Dictionary methods**

clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and value
get()	Returns the value of the specified key
items()	Returns a list containing a tuple for each key value pair
keys()	Returns a list containing the dictionary's keys
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair
setdefault()	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
update()	Updates the dictionary with the specified key-value pairs
values()	Returns a list of all the values in the dictionary

```
dict2 = {"one" : 1, "two" : 2, "three" : 3}
dict2.get("two")
```

```
2
```

```
dict2 = {"one" : 1, "two" : 2, "three" : 3}
dict2.keys()
```

```
dict_keys(['one', 'two', 'three'])
```

```
dict2 = {"one" : 1, "two" : 2, "three" : 3}
dict2.values()
```

```
dict_values([1, 2, 3])
```

```
dict2 = {"one" : 1, "two" : 2, "three" : 3}
dict2.items()
```

```
dict_items([('one', 1), ('two', 2), ('three', 3)])
```

```
dict2 = {"name" : "ali", "heigh": 165, "weight": 65}
dict2.popitem()
dict2
```

```
{'name': 'ali', 'heigh': 165}
```

```
dict2 = {"name" : "ali", "height": 165, "weight": 65}
dict2.update({"height":190, "average":19})
dict2
```

```
{'name': 'ali', 'height': 190, 'weight': 65, 'average': 19}
```

```
dict2 = {"name" : "ali", "height": 165, "weight": 65}
dict2.pop("height")
dict2
```

```
{'name': 'ali', 'weight': 65}
```

- **Fromkeys ():** With this command, you can have a dictionary with multiple keys and assign them all the same value. Dictname.fromkeys([key_1, key_2,…,key_n] , specified value)
- You can use list () function when using keys, values and items to turn the output to a list

```
dict2 = {"name" : "ali", "height": 165, "weight": 65}
list(dict2.items())
```

```
[('name', 'ali'), ('height', 165), ('weight', 65)]
```

```
#fromkey on an empty dictionary
dict1.fromkeys(["sara", "ali","sana"],20)
```

```
{'sara': 20, 'ali': 20, 'sana': 20}
```

```
#fromkey on a non empty dictionary
dict2 = {"heigh": 165, "weight": 65}
dict2.fromkeys(["height","weight"], "not given")
```

```
{'height': 'not given', 'weight': 'not given'}
```

- **Loop dictionary**

```
dict1 = {"key1" : 1, "key2" : 2, "key3" : 3}
for i in dict1 :
    print(i)
```

```
key1
key2
key3
```

```
dict1 = {"key1" : 1, "key2" : 2, "key3" : 3}
for i in dict1 :
    print(i, "-->", dict1[i])
```

```
key1 --> 1
key2 --> 2
key3 --> 3
```

```
dict1 = {"key1" : 1, "key2" : 2, "key3" : 3}
for key , value in dict1.items():
    print(key , "-->", value)
```

```
key1 --> 1
key2 --> 2
key3 --> 3
```

```
#one line code
dict1 = {"key1" : 1, "key2" : 2, "key3" : 3}
[[key, value] for key , value in dict1.items()]
#You can use [], {}, or () to enclose key-value pairs as needed.
```

```
[['key1', 1], ['key2', 2], ['key3', 3]]
```

- How to turn a nested list into a dictionary?

- Examples:
  - Sum () function: sum is used to add all the elements of an iterable (like a list , tuple and set) and return the total sum.

```
#best_way
#calculate each student's average and store it in a new dictionary.
scores = {"sara" : [18,19,20],
          "emily": [14,15,16],
          "ann" : [12,13,14]}
averages = {}
for name , score in scores.items() :
    averages[name] = sum(score) / len(score)
averages
```

```
{'sara': 19.0, 'emily': 15.0, 'ann': 13.0}
```

```
# store the elements of a list  with their number of occurrences in a dictionary
mylist = [1,2,3,7,8,9,3,6,9,7,4,1,7,8,9]
dic = {}
for i in mylist :
   dic[i] = mylist.count(i)
dic
```

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

```
#turn nested list into a dictionary
mylist = [['key1', 1], ['key2', 2], ['key3', 3]]
mydict = {}
for key, value in mylist:  #unpacking
    mydict[key] = value
mydict
```

```
{'key1': 1, 'key2': 2, 'key3': 3}
```

```
mydict = {}
for i in mylist:
    mydict[i[0]]=i[1]
mydict
```

```
{'key1': 1, 'key2': 2, 'key3': 3}
```

```
#one line
{key : value for key, value in mylist}  #unpacking
```

```
{'key1': 1, 'key2': 2, 'key3': 3}
```

```
#another one line
{i[0]: i[1] for i in mylist }
```

```
{'key1': 1, 'key2': 2, 'key3': 3}
```

```
# print 2 countries with the highest population
countries = {"a" :9, "b":3, "c":1}
pops = list(countries.values()) #[9, 3, 1, 3]
pops.sort(reverse = True) #[9, 3, 3, 1]
rank1 = pops[0]
rank2 = pops[1]
for key, value in countries.items():
    if value == rank1 or value ==rank2:
        print(key,"-->", value)
        #or
    if value in[rank1 , rank2]:
        print(key,"-->", value)
```

```
a --> 9
b --> 3
```

```
a = [1] #single list
b = {1:1} #single dict
c = {"d"} #single set
d = (1) #it is an integer not a single tuple.
print(type(a),type(b),type(c),type(d))
```
```
<class 'list'> <class 'dict'> <class 'set'> <class 'int'>
```

- Section 8: Tuple (item1, item2, item3)
  - How to define a Tuple? by using parentheses and comma.
  - Single tuple: A single-element tuple is not a tuple type. However, you can have a single list, dictionary and set.
    But how to create a single-element tuple?
    There is a trick for that

```
#correct way to have a single tuple
d = (1, )
type(d)
```
```
tuple
```

  - epmty tuple

```
#empty tuple
a = tuple()
#or
a = ()
a
```
```
()
```

  - Converting tuple to list and vice versa

```
a = [1,2,3]
tuple(a)
```
```
(1, 2, 3)
```
```
b = (1,2,3)
list(b)
```
```
[1, 2, 3]
```

  - Concatenate tuples:
    you can concatenate **lists**, **tuples**, and **strings** using the + operator.

```
a = (1,2,3)
b = (4,5,6)
a + b
```
```
(1, 2, 3, 4, 5, 6)
```

  - Repeat tuples:
    you can repeat **lists**, **tuples**, and **strings** using the * operator

```
a = (1,2)
b = [3,4]
c = "hi"
print(a*3, b*3,c*3)
```
```
(1, 2, 1, 2, 1, 2) [3, 4, 3, 4, 3, 4] hihihi
```

  - Check if a value exists in a tuple

```
a = (1,2)
1 in a
```
```
True
```

  - Indexing

```
a = (1,2,3)
a[-1]
```
```
3
```

  - Slicing
    Both lists and tuples can be sliced.

```
a = (1,2,3,4,5,6,7,8)
a[ :7:2]
```
```
(1, 3, 5, 7)
```

- tuple characteristics

1. tuples are ordered (1,2,3)≠ (3,2,1)
2. tuples items are accessed by index
3. tuples are fixed length: items cannot be added or removed.
4. tuples are heterogenous
5. tuples are nestable
6. tuples are immutable: Therefore, you use tuples for data that you are sure doesn't need to change.
   To modify a tuple, convert it to a list, make the desired change, and then convert it back to a tuple.
7. Most importantly, tuples are faster than lists
8. due to tuple's immutability, they can be used as a key in dictionaries.

```
tuple1 = (1,2.4,True,"hi", [2,3,4],(1,2,3),{1: "one", 2:"two"})
tuple1[5][2]
```
```
3
```

```
tuple1 = (1,2.4,True,"hi", [2,3,4],(1,2,3),{1: "one", 2:"two"})
listed = list(tuple1) #[1, 2.4, True, 'hi', [2, 3, 4], (1, 2, 3), {1: 'one', 2: 'two'}]
listed[2] = False
listed.pop(-1)
listed.insert(6,"salam") #[1, 2.4, False, 'hi', [2, 3, 4], (1, 2, 3), 'salam']
tuple1 = tuple(listed)
tuple1
```
```
(1, 2.4, False, 'hi', [2, 3, 4], (1, 2, 3), 'salam')
```

- Tuple methods

count()	Returns the number of times a specified value occurs in a tuple
index()	Searches the tuple for a specified value and returns the position of where it was found

```
a = (1,2,3,1)
a.count(1)
```
```
2
```

```
a = (1,2,3,1)
a.index(3)
```
```
2
```

```
a = (1,2,3)
a.__len__()
```
```
3
```

- Unpacking tuples (same as lists)
  Tip: When the number of variables doesn't match the tuple's length, use * to assign multiple items to a single variable in list format.

```
tuple1 = (10,12,14)
a,b,c = tuple1
print(a,b,c)
```
```
10 12 14
```

```
tuple1 = (10,12,14)
a, *b = tuple1
print(a,b)
```
```
10 [12, 14]
```

```
tuple1 = (10,12,14)
*a, b = tuple1
print(a,b)
```
```
[10, 12] 14
```

```
tuple1 = (1,2,3,4,5,6,7)
a,*b, c = tuple1
print(a,b,c)
```
```
1 [2, 3, 4, 5, 6] 7
```

```
tuple1 = ((1,2,-1),(3,4,-3),(5,6,-6))
for *item1,item2 in tuple1:
    print(item1, item2)
```
```
[1, 2] -1
[3, 4] -3
[5, 6] -6
```

- Tuple loops

```
tuple1 = ((1,2),(3,4),(5,6))
for item in tuple1:
    print(item)
```

```
(1, 2)
(3, 4)
(5, 6)
```

```
tuple1 = ((1,2),(3,4),(5,6))
for item in tuple1:
    for t in item:
            print(t, end = " ")
```

```
1 2 3 4 5 6
```

```
tuple1 = ((1,2),(3,4),(5,6))
for item1,item2 in tuple1:
    print(item1)
```

```
1
3
5
```

```
dict1 = {"sara":23,"ali":15,"mahsa":22}
[(name, age) for name, age in dict1.items()]
```

```
[{23, 'sara'}, {15, 'ali'}, {22, 'mahsa'}]
```

```
#what if you put the line in a () instead []? the output will be a generator object
((name, age) for name, age in dict1.items())
```

```
<generator object <genexpr> at 0x000001B366418110>
```

- Examples

```
#remove 4 from tuple
my_tuple = (10,2,3,4,5,7,9)
my_list = list(my_tuple)
my_list.remove(4)
my_tuple = tuple(my_list)
my_tuple
```

```
(10, 2, 3, 5, 7, 9)
```

```
#slicing way
my_tuple = (10,2,3,4,5,7,9)
my_tuple = my_tuple[0:3] + my_tuple[4:]
my_tuple
```

```
(10, 2, 3, 5, 7, 9)
```

```
#increase each item by one
numeric_tuple = (5,3,7,9)
numeric_list = list(numeric_tuple)
for i in range(len(numeric_list)):
    numeric_list[i] += 1
numeric_tuple = tuple(numeric_list)
numeric_tuple
```

```
(6, 4, 8, 10)
```

```
#or
numeric_tuple = (5,3,7,9)
numeric_list = list(numeric_tuple)
numeric_list = [i + 1 for i in numeric_list]
numeric_tuple = tuple(numeric_list)
numeric_tuple
```

```
#compute the average of each lesson and store it in a new tuple.
scores = [("math", [18,19,20]),("science", [17,18,19])]
averages = []
for lesson, score in scores:
    avg = sum(score) / len(score)
    averages.append((lesson , avg))
averages
```

```
[('math', 19.0), ('science', 18.0)]
```

- Section 9: Set {item1, item2, item3}
  - How to define a set: sets are defined using curly braces. {}

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

  - Sets do not support Indexing & slicing

```
my_set = {"a","b","c"}
my_set[1]
```

```
TypeError                                 Tra
Cell In[28], line 2
      1 my_set = {"a","b","c"}
----> 2 my_set[1]

TypeError: 'set' object is not subscriptable
```

- Set characteristics

1.sets are unordered {1,2,3} = {3,2,1}

The set's output might not be in the same
e order as the input set.

2. set items can't be accessed by index or key

3. sets are dynamic Elements can be added or removed, but not modified.

4. sets are heterogenous: Dictionaries, lists and sets cannot be elements of a set.

5. sets are not nestable

6. sets are immutable

So set items must be immutable and That's why Dictionaries, lists cannot be set items.

```
a = {"ali":5, "sara":9}
my_set = {1,2,3,a}
```

```
TypeError
Cell In[21], line 2
      1 a = {"ali":5, "sara":9}
----> 2 my_set = {1,2,3,a}

TypeError: unhashable type: 'dict'
```

```
a = {7,8,9}
my_set = {1,2,3,a}
```

```
TypeError
Cell In[15], line 2
      1 a = {7,8,9}
----> 2 my_set = {1,2,3,a}

TypeError: unhashable type: 'set'
```

7. duplicates are not allowed.

Sets eliminate duplicates.

Therefore, to remove duplicate items from a list or tuple, convert it to a set.

```
my_set = {1,2,3}
my_set[0] = 5
```

```
TypeError                                 Traceback (most
Cell In[13], line 2
      1 my_set = {1,2,3}
----> 2 my_set[0] = 5

TypeError: 'set' object does not support item assignment
```

  - Empty set

```
#empty set
a = set()
```

```
#remove duplicates.
a = [1,2,3,4,5,6,2,6,3]
list(set(a))
```

```
[1, 2, 3, 4, 5, 6]
```

```
a = {1,2,"hi",1,5,"hi",False}
a
```

```
{1, 2, 5, False, 'hi'}
```

- Loop through a set

```
a = {1,2,3,"hi",(5,6),True}
for i in a :
    print(i, end = " " )
```

```
1 2 3 (5, 6) hi
```

```
a = {1,2,3,"hi",(5,6),True}
{ i for i in a }
```

```
{(5, 6), 1, 2, 3, 'hi'}
```

```
a = {1,2,3,"hi",(5,6),True, 4.4}
{i for i in a if type(i) in[float, int]}
```

```
{1, 2, 3, 4.4}
```

- Set methods

add()	Adds an element to the set
clear()	Removes all the elements from the set
copy()	Returns a copy of the set
difference()	Returns a set containing the difference between two or more sets
difference_update()	Removes the items in this set that are also included in another, specified set
discard()	Remove the specified item
intersection()	Returns a set, that is the intersection of two or more sets
intersection_update()	Removes the items in this set that are not present in other, specified set(s)
isdisjoint()	Returns whether two sets have a intersection or not
issubset()	Returns whether another set contains this set or not
issuperset()	Returns whether this set contains another set or not
pop()	Removes an element from the set
remove()	Removes the specified element
symmetric_difference()	Returns a set with the symmetric differences of two sets
symmetric_difference_update()	inserts the symmetric differences from this set and another
union()	Return a set containing the union of sets
update()	Update the set with another set, or any other iterable

```
a = {1,2,3,4}
b = {4,5,6,7}
c = a.intersection(b)
print(a , c)
```

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

```
a = {1,2,3,4}
b = {4,5,6,7}
a.intersection_update(b)
print(a)
```

```
{4}
```

```
a = {1,2,3,4}
b = {4,5,6,7}
c = a.difference(b)
print(a , c)
```

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

```
a = {1,2,3,4}
b = {4,5,6,7}
a.difference_update(b)
print(a)
```

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

```
a = {1,2,3,4}
b = {4,5,6,7}
a.pop()
b.pop()
print(a,b)
```

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

```
a = {1,2,3,4}
a.discard(1) #or a.remove(1)
a
```

```
{2, 3, 4}
```

```
a = {1,2,3,4}
a.discard(8)
a
```

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

```
a = {1,2,3,4}
a.remove(8)
a
```

```
---------------------------
KeyError
Cell In[107], line 2
      1 a = {1,2,3,4}
----> 2 a.remove(8)
      3 a

KeyError: 8
```

```
a = {1,2,3,4}
b = {4,5,6,7}
c = a.union(b)
print(a , c)
```

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

```
a = {1,2,3,4}
b = {4,5,6,7}
a.update(b)
print(a)
```

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

```
a = {1,2,3,4}
b = {4,5,6,7}
a.isdisjoint(b)
```

```
False
```

```
a = {1,2,3,4}
b = {5,6,7}
a.isdisjoint(b)
```

```
True
```

```
a = {1,2,3}
b = {1,2,3,4}
a.issubset(b)
```

```
True
```

```
a = {1,2,3}
b = {1,2,3,4}
b.issuperset(a)
```

```
True
```

```
#symmetric_differencea = (a-b) U (b-a)  or (a U b) - (a ∩ b)
a = {1,2,3,4}
b = {4,5,6,7}
c = a.symmetric_difference(b)
print(c, a)
```

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

```
a = {1,2,3,4}
b = {4,5,6,7}
a.symmetric_difference_update(b)
print(a)
```

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

-**Add ()** functions similarly to **append ()** for lists. **update ()** functions similarly to **union ()** or **extend ()** for lists
-**union ()**, **intersection (), difference () and symmetric_difference ()** create new sets with the result, while **update ()**, **intersection_update () , difference_update () and symmetric_difference_update ()** modify the existing set directly.
-**Remove ()** and **discard ()** are similar, except that if the item we want to delete doesn't exist, remove throws an error, but discard doesn't.
-**pop ()** removes an element randomly and then update the set
- **isdisjoint ()**, **issubset ()** and **issupersert ()** return True or False.

```
a = {1,2,3,4}
b = {4,5,6,7}
a.symmetric_difference(b) == a.union(b).difference(a.intersection(b)) == a.difference(b).union(b.difference(a))
```

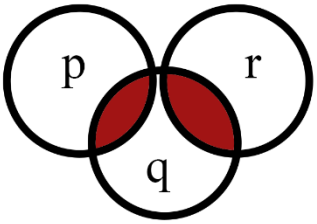
```
True
```

- Examples

```
#Store list elements in a dictionary along with their occurrence count.
mylist = [5,6,2,2,7,4,5,5,6,9,7,2]
a = [{item : mylist.count(item)} for item in set(mylist)]
a
```

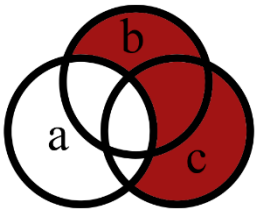
```
[{2: 3}, {4: 1}, {5: 3}, {6: 2}, {7: 2}, {9: 1}]
```

```
mylist = [5,6,2,2,7,4,5,5,6,9,7,2]
myset = set(mylist)
mydict = {}
for item  in myset :
    mydict[item] = mylist.count(item)
mydict
```



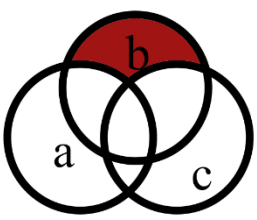

```
p = {0,1,2,3,4}
q = {4,6,8}
r = {6,12,18}
#(pnq) U (rnq)
p.intersection(q).union(r.intersection(q))
```

```
{4, 6}
```




```
a = {2,4,6,8}
b = {2,3,5,7}
c = {1,3,5,15}
# (b U c) - c
b.union(c).difference(a)
```

```
{1, 3, 5, 7, 15}
```




```
a = {2,4,6,8}
b = {2,3,5,7}
c = {1,3,5,15}
# b - ( a U c )
b.difference(a.union(c))
```

```
{7}
```

- Section 10: Strings
  - how to define a string: using single quotes ‘ ‘ or double quotes “ “ or triple quotes’’’ ’’’ for multi-line strings
    tip: \n in a string represents a newline character (an enter).
  - indexing and slicing for strings are supported

```
a = "hellow world"
b = 'howsam'
c = ''' this is 'multi line' string '''
```

```
s = '''i love
 python'''
s
```

```
'i love \n python'
```

Each character in the string, including spaces and enters, can be referenced and has an index.

```
txt = "i love python"
txt[0:8:2]
```

```
'ilv '
```

```
txt = "i love python"
print(txt[2],txt[8])
txt[6] #is a space
```

```
l y
' '
```

  - modifying strings: Strings are immutable and cannot be changed directly.

```
txt = "i love python"
txt[7] = "P"
```

```
TypeError                                 Traceback (most
Cell In[38], line 2
      1 txt = "i love python"
----> 2 txt[7] = "P"

TypeError: 'str' object does not support item assignment
```

- string methods

capitalize()	Converts the first character to upper case
casefold()	Converts string into lower case
center()	Returns a centered string
count()	Returns the number of times a specified value occurs in a string
encode()	Returns an encoded version of the string
endswith()	Returns true if the string ends with the specified value
expandtabs()	Sets the tab size of the string
find()	Searches the string for a specified value and returns the position of where it was found
format()	Formats specified values in a string
format_map()	Formats specified values in a string
index()	Searches the string for a specified value and returns the position of where it was found
isalnum()	Returns True if all characters in the string are alphanumeric
isalpha()	Returns True if all characters in the string are in the alphabet
isascii()	Returns True if all characters in the string are ascii characters
isdecimal()	Returns True if all characters in the string are decimals
isdigit()	Returns True if all characters in the string are digits
isidentifier()	Returns True if the string is an identifier

```
txt = "hello"
print(txt.endswith("llo") , txt.startswith("hel"))
```

```
True True
```

```
txt = "im mahdis im 24 years old im student"
txt.find("mahdis")
```

```
3
```

```
txt.index("mahdis")
```

```
3
```

```
# find and index fuction similarly but there is a difference
#find Returns -1 for any character that does not exist
txt.find("j")
```

```
-1
```

```
txt.index("j")
```

```
ValueError                                Traceback (most re
Cell In[65], line 1
----> 1 txt.index("j")

ValueError: substring not found
```

```
txt = "im mahdis im 24 years old im student"
txt.count("m")
```

```
4
```

```
#you can specify a range for this method
txt.count("m", 3,10)  # or yoy can use slicing instead of defining a range.
#txt[3:10].count("m")
```

```
1
```

```
txt = "howsamī"
txt.encode()
```

```
b'howsam\xd8\xa2'
```

```
"ī".encode()
```

```
b'\xd8\xa2'
```

islower()	Returns True if all characters in the string are lower case
isnumeric()	Returns True if all characters in the string are numeric
isprintable()	Returns True if all characters in the string are printable
isspace()	Returns True if all characters in the string are whitespaces
istitle()	Returns True if the string follows the rules of a title
isupper()	Returns True if all characters in the string are upper case
join()	Converts the elements of an iterable into a string
ljust()	Returns a left justified version of the string
lower()	Converts a string into lower case
lstrip()	Returns a left trim version of the string
maketrans()	Returns a translation table to be used in translations
partition()	Returns a tuple where the string is parted into three parts
replace()	Returns a string where a specified value is replaced with a specified value
rfind()	Searches the string for a specified value and returns the last position of where it was found
rindex()	Searches the string for a specified value and returns the last position of where it was found
rjust()	Returns a right justified version of the string
rpartition()	Returns a tuple where the string is parted into three parts
rsplit()	Splits the string at the specified separator, and returns a list
rstrip()	Returns a right trim version of the string
split()	Splits the string at the specified separator, and returns a list
splitlines()	Splits the string at line breaks and returns a list
startswith()	Returns true if the string starts with the specified value

```
#The functions of split and join are opposite to each other.
txt = "im mahdis im 24 years old im student"
txt.split()  #split by spaces
```

```
['im', 'mahdis', 'im', '24', 'years', 'old', 'im', 'student']
```

```
txt.split("m") #split by m
```

```
['i', ' ', 'ahdis i', ' 24 years old i', ' student']
```

```
#You can specify the number of times this command is executed.
txt.split("m" , 2)
```

```
['i', ' ', 'ahdis im 24 years old im student']
```

```
#isalnum = is alpha or is digit
txt1 = "hihello"
print(txt1.isalnum(),txt1.isalpha(),txt1.isdigit())
```

```
True True False
```

```
txt2 = "123456"
print(txt2.isalnum(),txt2.isalpha(),txt2.isdigit())
```

```
True False True
```

```
txt3 = "hi85"
print(txt3.isalnum(),txt3.isalpha(),txt3.isdigit())
```

```
True False False
```

```
#Because the symbol is neither a number nor an alphabet
"@".isalnum()
```

```
False
```

```
txt = "hi my name is mahdis"
txt.replace("h" , "H",)
```

```
'Hi my name is maHdis'
```

```
txt = "hi my name is mahdis"
txt.replace("name" , "NAME",)
```

```
'hi my NAME is mahdis'
```

```
#a trick: delete whatever you dont want by raplacing with ""
txt = "hi! how! are! you!"
txt.replace("!" , "")
```

```
'hi how are you'
```

```
#you can use join on iterables
a = ["m","a","h","d","i","s"]
print("_".join(a) , "".join(a))
```

```
m_a_h_d_i_s mahdis
```

```
b = ("hi", "world")
" ".join(b)
```

```
'hi world'
```

```
#you can even use join on a string
c = "kffjf"
" ".join(c)
```

```
'k f f j f'
```

strip()	Returns a trimmed version of the string
swapcase()	Swaps cases, lower case becomes upper case and vice versa
title()	Converts the first character of each word to upper case
translate()	Returns a translated string
upper()	Converts a string into upper case
zfill()	Fills the string with a specified number of 0 values at the beginning

```
txt = "i love python"
txt1 = "MAHDIS"
u = txt.upper()
l = txt1.lower()
print( u , l)
```

```
I LOVE PYTHON mahdis
```

```
txt = "50"
num = 5
txt.zfill(num)
```

```
'00050'
```

```
txt1 = "mah"
txt1.zfill(8)
```

```
'00000mah'
```

```
#YOU CAN SPECIFY A RANGE FOR THESE METHODS
txt = "hello SWEET HAERT"
txt.startswith("SWEET", 6,11)
```

```
True
```

```
txt.endswith("ERT",0,-1)
```

```
False
```

```
txt.endswith("ERT",0,)
```

```
True
```

- . Format ()

This method is used to insert and concatenate strings dynamically. It allows you to specify placeholders within a string using curly braces {}, which can be replaced with variables or values

```
#without format()
name = "ali"
score = 18
lesson = "math"
txt = name + " got " + str(score) + " in " + lesson
txt
```

```
'ali got 18 in math'
```

```
#with format()
#delete variables and use {} instead
"{} got {} in {}".format(name, score, lesson)
```

```
'ali got 18 in math'
```

```
#or you can put labels in {} to clarify
"{n} got {s} in {l}".format(n = name, s = score, l = lesson)
```

```
'ali got 18 in math'
```

```
"{}/{}/{}".format(14.1, 10, 15)
```

- F-strings

  f-strings provide a concise way to embed expressions directly inside string literals by prefixing the string with f or F. Variables and expressions are inserted using curly braces {}, making string formatting more readable and efficient.

```
#to use f-strings, you should put variables in {}.
name = "sara"
score = 18
lesson = "class"
f"{name} got {score} in {lesson}"
```

```
'sara got 18 in class'
```

```
name = "Bob"
age = 25
print(f"My name is {name} and I'm {age} years old.")
```

```
My name is Bob and I'm 25 years old.
```

- tips and tricks
  - how to get help about a string method?
  - Check if character exists in a string

```
txt = "i love python"
"python" in txt
```

```
True
```

```
help(str.count)
```

```
Help on method_descriptor:

count(...) unbound builtins.str method
    S.count(sub[, start[, end]]) -> int

    Return the number of non-overlapping occurrences of substring sub in
    string S[start:end].  Optional arguments start and end are
    interpreted as in slice notation.
```

- examples

```
#Modify the text so that one of each letter appears in the output.
txt = "hiiiiiii mmyy ffffriiiennnddsss"
out = ""
for i in range(len(txt)):
    if txt[i] != txt[i-1]:
        out += txt[i]
out
```

```
'hi my friends'
```

```
#In this method, "out" cannot be an empty string.
#This is because if it is an empty string, the first time the loop executes,
#there will be nothing inside "out" to compare with "i".
out = " "
for i in txt:
    if i != out[-1]:
        out += i
out
```

```
' hi my friends'
```

```
#Write code that prints the words of a string in reverse
#Each word should be reversed in its place.
#"this is an example " --> "siht si na elpmaxe"
txt = "this is an example"
txt = "this is an example"
#txt[ : :-1] #slicing with -1 step reverses the string totally = 'elpmaxe na si siht'
#txt[ : :-1].split()  #['elpmaxe', 'na', 'si', 'siht']
#txt[ : :-1].split()[::-1] #List slicing with -1 step = ['siht', 'si', 'na', 'elpmaxe']
" ".join(txt[ : :-1].split()[::-1])
```

```
#Print the 4-digit numbers in the text.
txt = " ab abcd 1 1234 5689 ab12  12cd"
txt.split() #['ab', 'abcd', '1', '1234', '5689', 'ab12', '12cd']
for i in txt.split() :
    if i.isdigit() and len(i) == 4:
        print(i)
```

```
1234
5689
```

```python
#turn the text into this "this_is_a_string"
txt = "this is a string"
txt.replace(" " , "_")
```

```
'this_is_a_string'
```

```python
txt = "this is a string"
"_".join(list(txt.split())) #['this', 'is', 'a', 'string'])
```

```
'this_is_a_string'
```

```python
# without any prepered functoin
txt = "this is a string"
new_text = ""
for i in txt:
    if i == " " :
        new_text += "_"
    else :
        new_text += i
new_text
```

```
'this_is_a_string'
```

```python
#Determine which words are written with both hands.
#mycode
left = ["q","w","e","r","t","a","s","d","f","g","z","x","c","v","b"]
right = ["y","u","i","o","p","h","j","k","l","n","m"]
words = ["yellow" , "test" , "pool"]
for word in words :
    lis = []
    for item in word:
        if item in right:
            lis.append(right)
        else :
            lis.append(left)
    if right in lis and left in lis :
        print(word)
```

```
yellow
```

```python
#better code using flags
#what if you don't use break ? it will print yellow 5 times
for word in words :
    left_flag = False
    right_flag = False
    for item in word:
        if item in right:
            right_flag = True
        elif item in left:
            left_flag = True
        if left_flag and right_flag :
            print(word)
            break
```

```
yellow
```

```python
#easier
for word in words:
    left_found = any(letter in left for letter in word)
    right_found = any(letter in right for letter in word)
    if  left_found and  right_found:
        print(word)
```

- Section 11 : Built-in functions (predefined functions)

abs()	Returns the absolute value of a number
all()	Returns True if all items in an iterable object are true
any()	Returns True if any item in an iterable object is true
ascii()	Returns a readable version of an object. Replaces none-ascii characters with escape character
bin()	Returns the binary version of a number
bool()	Returns the boolean value of the specified object
bytearray()	Returns an array of bytes
bytes()	Returns a bytes object
callable()	Returns True if the specified object is callable, otherwise False
chr()	Returns a character from the specified Unicode code.
classmethod()	Converts a method into a class method
compile()	Returns the specified source as an object, ready to be executed
complex()	Returns a complex number
delattr()	Deletes the specified attribute (property or method) from the specified object
dict()	Returns a dictionary (Array)
dir()	Returns a list of the specified object's properties and methods
divmod()	Returns the quotient and the remainder when argument1 is divided by argument2

```
#absolute value (abs)
abs(-5)
```

```
5
```

```
abs(8)
```

```
8
```

```
any([False,False,False,True])
```

```
True
```

```
b = (True,True,False)
all(b)
```

```
False
```

```
#complex(real, imaginary)
#Both real and imaginary are optional and default to 0 if not provided
# Create a complex number with real part 2 and imaginary part 3
c1 = complex(2, 3)
print(c1)
```

```
(2+3j)
```

```
# Create a complex number with only real part
c2 = complex(5)
print(c2)
```

```
(5+0j)
```

```
# Create a complex number from a string
c3 = complex("4+5j")
print(c3)
```

```
(4+5j)
```

```
# divmod(dividend , divisor)
divmod(10 , 3) #output = (quotient, remainder)
```

```
(3, 1)
```

```
#conversion functions
print(int(2.5),float(4),str(6),bool(5))
```

```
2 4.0 6 True
```

```
#bin () converts an integer to its binary string representation
bin(8)
```

```
'0b1000'
```

```
# hex () converts an integer to its hexadecimal string representation
hex(17)
```

```
'0x11'
```

```
#ord () and chr() are inverse
print(ord("ؤ"),ord("1"))
```

```
1570 108
```

```
print(chr(1570),chr(108))
```

```
ؤ l
```

enumerate()	Takes a collection (e.g. a tuple) and returns it as an enumerate object
eval()	Evaluates and executes an expression
exec()	Executes the specified code (or object)
filter()	Use a filter function to exclude items in an iterable object
float()	Returns a floating point number
format()	Formats a specified value
frozenset()	Returns a frozenset object
getattr()	Returns the value of the specified attribute (property or method)
globals()	Returns the current global symbol table as a dictionary
hasattr()	Returns True if the specified object has the specified attribute (property/method)
hash()	Returns the hash value of a specified object
help()	Executes the built-in help system
hex()	Converts a number into a hexadecimal value
id()	Returns the id of an object
input()	Allowing user input
int()	Returns an integer number
isinstance()	Returns True if a specified object is an instance of a specified object
issubclass()	Returns True if a specified class is a subclass of a specified object
iter()	Returns an iterator object
len()	Returns the length of an object
list()	Returns a list
locals()	Returns an updated dictionary of the current local symbol table

```
# format () used to format values into strings with specific formatting options
#It allows you to embed variables into strings and customize how they appear
#format(value, format_spec)
```

```
# display 22.5465 in scientific notation
format(22.5465, "e")
```

```
'2.254650e+01'
```

```
# Calculate the percentage of 0.5
format(0.5, "%")
```

```
'50.000000%'
```

```
#Convert 10 to binary.
format(10 , "b")
```

```
'1010'
```

```
#Separate the digits into groups of three, using _ as a separator.
format(52478965556635, "_")
```

```
'52_478_965_556_635'
```

```
#filter() filters items of an iterable based on a condition.
#it resembles map() but the function output must be True or False
#the output is the items that met the condition
my_list = ["hi" , "howsam" , "12", "ai" , "20" ]
list(filter(str.isdigit , my_list ))
```

```
['12', '20']
```

```
#without filter()
my_list = ["hi" , "howsam" , "12" , "ai" , "20" ]
output = []
for i in my_list :
    if i.isdigit():
        output.append(i)
output
```

```
['12', '20']
```

```
# Filter only strings that are purely digits, then convert to int
scores = ["mahdis" , "2.3" , "a" , "5" , "8", "hi", "True"]
list(map(int,(filter(str.isdigit, scores))))
```

```
[5, 8]
```

```
#iter() creates an iterator from an iterable object.
#An iterator allows you to traverse through all the elements in the iterable using methods like next().
x = [1,0,-1,5]
x_iter = iter(x)
```

```
# you must write next() in the next cell or it wont work
#each time you run the cell, you get an element, and if you run the cell more than the iterable lenght
#,it will drop an error.
next(x_iter)
```

```
5
```

```
#enumerate =  allows you to loop over an iterable while keeping track of the index of each item.
#enumerate(iterable, start)  default of  start is 0  #output is (index , item)
#use list  to see the output
a = ["ali" , "mahsa" , "sara","mohsen"]
list(enumerate(a)) #start default = 0
```

```
[(0, 'ali'), (1, 'mahsa'), (2, 'sara'), (3, 'mohsen')]
```

```
b = ("a" , "b" , "c" , "d")
list(enumerate(b, 5))
```

```
[(5, 'a'), (6, 'b'), (7, 'c'), (8, 'd')]
```

```
#enumerate loop and unpacking
for indx , item in enumerate(a,1):
    print(indx ,":",item)
```

```
1 : ali
2 : mahsa
3 : sara
4 : mohsen
```

```
#here, we dont need the second variable value so we use _ in unpacking.
for _ , item in enumerate(a,1):
    print(item)
```

```
ali
mahsa
sara
mohsen
```

```
#isinstance() checks the type
#isinstance(variable or value, type)
a = [1,2,3]
isinstance(a , list) # or type(a) == True
```

```
True
```

```
isinstance(3.5 , int)
```

```
False
```

```
isinstance((2,3,5) , tuple)
```

```
True
```

map()	Returns the specified iterator with the specified function applied to each item
max()	Returns the largest item in an iterable
memoryview()	Returns a memory view object
min()	Returns the smallest item in an iterable
next()	Returns the next item in an iterable
object()	Returns a new object
oct()	Converts a number into an octal
open()	Opens a file and returns a file object
ord()	Convert an integer representing the Unicode of the specified character
pow()	Returns the value of x to the power of y
print()	Prints to the standard output device
property()	Gets, sets, deletes a property
range()	Returns a sequence of numbers, starting from 0 and increments by 1 (by default)
repr()	Returns a readable version of an object
reversed()	Returns a reversed iterator
round()	Rounds a numbers
set()	Returns a new set object
setattr()	Sets an attribute (property/method) of an object
slice()	Returns a slice object
sorted()	Returns a sorted list
staticmethod()	Converts a method into a static method
str()	Returns a string object

```
#min and max function on list, tuple and set
a = [1,2,3]
b = (4,5,6)
c = {7,8,9}
print(min(a),max(b),min(c))
```

```
1 6 7
```

```
max(1,2,3,4,5,6)
```

```
6
```

```
a = (9,5,8)
b = [8,6,4]
c = {2,6,7}
d = {"a":10 , "b": 5}
print(sorted(a),sorted(b),sorted(c),sorted(d))
```

```
[5, 8, 9] [4, 6, 8] [2, 6, 7] ['a', 'b']
```

```
(9, 5, 8)
```

```
sorted(a, reverse = True)
```

```
[9, 8, 5]
```

```
#round function = round(number, digit)
#If digit = n, the function rounds the number to n digits after the decimal point
print(round(2.1), round(2.5), round(2.9))
print(round(6.99157, 1),round(6.99157, 2),round(6.99157, 3),round(6.99157, 4))
```

```
2 2 3
7.0 6.99 6.992 6.9916
```

```
#pow (x , y , z) = (x ** y) % z
pow(2,3) # == 2 ** 3
```

```
8
```

```
pow(2,3,3) # == (2 ** 3 ) % 3
```

```
2
```

```
# map applies a specified function to each item of an iterable and returns an iterator
# map(function, iterable, ...)
# use list() to see the result
a = [-1, 2, -3, 4]
list(map(abs ,a))
```

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

```
#or
a = [-1, 2, -3, 4]
for i , ai in enumerate(a):
    a[i] = abs(ai)
a
```

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

```
#range(start,stop,step)
#use list() to see the output of range inside a list
list(range(2,10,2))
```

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

```
#range's output is iterable and thats why you can use it in a for loop
for i in range(2,10,2) :
    print(i, end = " ")
```

```
2 4 6 8
```

```
#slice() can be used as indices of a list unlike range()
mylist = [1,2,3,4,5,6,7,8]
mylist[range(0,5,2)]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[81], line 3
      1 #slice() can be used as indices of a list unlike range()
      2 mylist = [1,2,3,4,5,6,7,8]
----> 3 mylist[range(0,5,2)]

TypeError: list indices must be integers or slices, not range
```

```
mylist[slice(0,5,2)]
```

```
[1, 3, 5]
```

```
#Slice() resembles ranges but produces non-iterable outputs, preventing its use in a for loop.
#slice(start,stop,step)
for i in slice(2,10,2) :
    print(i, end = " ")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[79], line 3
      1 #Slice() resembles ranges but produces non-iterable outputs, preventing its use in a for loop
      2 #slice(start,stop,step)
----> 3 for i in slice(2,10,2) :
      4     print(i, end = " ")

TypeError: 'slice' object is not iterable
```

```
#you cannot list a slice function because its output isn't iterable
list(slice(2,10,2))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[81], line 2
      1 #you cannot list a slice function because its output isn't iterable
----> 2 list(slice(2,10,2))

TypeError: 'slice' object is not iterable
```

sum()	Sums the items of an iterator
super()	Returns an object that represents the parent class
tuple()	Returns a tuple
type()	Returns the type of an object
vars()	Returns the dict property of an object
zip()	Returns an iterator, from two or more iterators

```
# add a and b element-wise using map
a = [1,2,3]
b = [-1,-2,-3]
list(map(sum,zip(a,b)))
```

```
[0, 0, 0]
```

```
# zip ()combines multiple iterables into an iterator of tuples,
#use list() to see the output
names = ['Alice', 'Bob', 'Charlie']
ages = (25, 30, 35)
list(zip(names, ages))
```

```
[('Alice', 25), ('Bob', 30), ('Charlie', 35)]
```

```
#how about Unequal iterable lengths?
# zip() will make tuples as much as it can combine
a = [1,2,3,4,5,6]
b = [-1,-2,-3]
c = (5,6,8,9)
list(zip(a,b,c))
```

```
[(1, -1, 5), (2, -2, 6), (3, -3, 8)]
```

```
for *abi , ci in zip(a,b,c):
    print(abi , ci)
```

```
[1, -1] 5
[2, -2] 6
[3, -3] 8
```

```
# zip () and enumerate()
a = [1,2,3]
b = [-1,-2,-3]
list(enumerate((zip(a,b))))
```

```
[(0, (1, -1)), (1, (2, -2)), (2, (3, -3))]
```

```
for i in enumerate((zip(a,b))) :
    print(i)
```

```
(0, (1, -1))
(1, (2, -2))
(2, (3, -3))
```

```
for i, abi in enumerate((zip(a,b))) :
    print(i, abi)
```

```
0 (1, -1)
1 (2, -2)
2 (3, -3)
```

```
for i, (ai,bi) in enumerate((zip(a,b))) :
    print(i , ai , bi)
```

```
0 1 -1
1 2 -2
2 3 -3
```

```
#calculate a+b element_wise
a = [1,2,3]
b = (-1,-2,-3)
for ai , bi in zip(a,b):
    print(ai + bi )
```

```
0
0
0
```

```
#one_line
[ai + bi for ai , bi in zip(a,b)]
```

```
[0, 0, 0]
```

```
#print(values , sep = " " , end = "\n")
a = [1,2,3]
print(a[0], a[1],a[2], end = "-")
```

```
1 2 3-
```

```
print(a[0], a[1],a[2], sep = "-")
```

```
1-2-3
```

```
print("A", "B", "C", sep="-", end="!")
```

```
A-B-C!
```

```
#Sum function on lists, tuples, and sets
a = [1, 2, 3]
b = (3, 4, 5)
c = {6,7}
print(sum(a), sum(b) , sum(c))
```

```
6 12 13
```

```
sum(a , 2)
```

```
8
```

- Input/ Output functions : a method for interacting with user

```
#input returns string
input()
```

```
↑↓ for history. Search history with c-↑/c-↓
```

```
#input returns string
input()
```

```
 1380
'1380'
```

```
#You can assign input() to a variable so you can call it later.
x = input()
x
```

```
↑↓ for history. Search history with c-↑/c-↓
```

```
x = input()
print(f"name : {x}")
```

```
 mahdis
name : mahdis
```

```
x = input("Enter your name")
print(f"name : {x}")
```

```
Enter your name mahdis
```

- Examples:

How to access values and indices in a matrix using enumerate →

```
mat = [[1,2,3],
       [4,5,6],
       [7,8,9]]
for i,r in enumerate(mat):
    for j,v in enumerate(r):
        print(f"{i},{j} --> {v}")
```

```
0,0 --> 1
0,1 --> 2
0,2 --> 3
1,0 --> 4
1,1 --> 5
1,2 --> 6
2,0 --> 7
2,1 --> 8
2,2 --> 9
```

```
#transpose the matrix
mat = [[1,2,3],
       [4,5,6],
       [7,8,9]]
#goal = [[1,4,7],
        #[2,5,8],
        #[3,6,9]]
list(map(list,(zip(mat[0],mat[1],mat[2]))))
```

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

```
#or
#using * allows you to provide multiple inputs without entirely mention them
# * will notice that mat has 3 items and provide mat[0] , mat[1] , mat[2]
list(map(list,(zip(*mat))))
```

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

```
#find the saddle point
#saddle point must be max in its row and min in its column
#here,  saddle point is 5
#my code
mat = [[9,8,7],
       [4,5,2],
       [6,6,7]]
max_of_row = list(map(max, (mat)))  #[9, 5, 7]
min_of_column = list(map(min, (zip(*mat))))  #[4, 5, 2]
for i in mat :
    for j in i:
        if j in max_of_row:
            if j in min_of_column:
                print(j)
```

```
5
```

```
#or
for i , r in enumerate(max_of_row):  #(0, 9), (1, 5), (2, 7)
    for j ,c in enumerate(min_of_column): #(0, 4), (1, 5), (2, 2)
        if c == r :
            print(f"{i} , {j} --> {c}")
```

```
1 , 1 --> 5
```

- **Section 12 : Functions**

functions are blocks of reusable code that perform a specific task. They help organize and modularize your code

- How to define a function ?

```
Def function-name(inputs) :
|||||statements
|||||return output
```

```
#defining fx2
#you can use print() instead of return
def fx2(x) :
    y = x ** 2
    return y

#calling fx2
fx2 (3)
```

```
9
```

- Define & call:

If a function doesn't include a **return** statement or a **print ()** statement, it doesn't produce any visible output or value stored in a variable. By default, functions return None if no return is specified.

```
def fx2(x) :
    y = x ** 2

print(fx2 (3))
```

```
None
```

```
def fx () :
    y = 5 ** 2
    print(y)

fx()
```

```
25
```

If there is no input variable, you can define a function without any inputs, and it still works just fine, but it's not so common.

```
def fxy (x , y) :
    z = x ** 2 + y  ** 2
    w = x ** 2 - y  ** 2
    return (z,w,x,y)

fxy(2,3)
```

```
(13, -5, 2, 3)
```

```
a , *b = fxy(2,3)
print(a , b)
```

```
13 [-5, 2, 3]
```

- Many inputs and outputs

- **Default parameter value:** you can specify default parameter values for functions using mod as an argument. This means if a caller doesn't provide an argument for that parameter, the function uses the default value.

Arithmetic Mean (Average)

$$\text{Arithmetic Mean} = \frac{\text{Sum of all values}}{\text{Number of values}} = \frac{\sum_{i=1}^{n} x_i}{n}$$

where x_i are the individual values, and n is the total number of values.

Geometric Mean

$$\text{Geometric Mean} = \left(\prod_{i=1}^{n} x_i\right)^{\frac{1}{n}} = \sqrt[n]{x_1 \times x_2 \times \cdots \times x_n}$$

where x_i are the positive individual values.

To define an average function, you need another input called (mod) to determine if you need arithmetic[1] or geometric[2] average.
Here, the default is arithmetic; therefore, if you need an arithmetic averag e, there is no need to specify it. Otherwise, if you need the geometric average, you need to mention it in calling.

```
#witout default
def avg (a, b , mod) :
    if mod == "arithmetic" :
        out = (a+b) / 2
    elif mod == "geometric" :
        out = (a*b) ** 0.5
    return out

avg(1,9,"arithmetic")
```

```
5.0
```

```
avg(1,9, "geometric")
```

```
3.0
```

```
#with default
def avg (a, b , mod = "arithmetic") :
    if mod == "arithmetic" :
        out = (a+b) / 2
    elif mod == "geometric" :
        out = (a*b) ** 0.5
    return out

avg(1,4)
```

```
2.5
```

```
avg (1,4,"geometric")
```

```
2.0
```

```
#here, there are 2 default values
#so you can call the function with only one input
def fun(a , b= 2 , c ="power") :
    if c == "power":
        return a ** b
    elif c == "multiplication":
        return a * b

fun(3) #a=3, b=default, c=default --> 3**2
```

```
9
```

```
fun(3,3) #a=3, b=3, c=default --> 3**3
```

```
27
```

```
fun(3, c="multiplication") #a= 3 , b=default , c="multiplication" -->3*2
```

```
6
```

```
def fun(a , b = 2 , c ) :
    if c == "power":
        return a ** b
    elif c == "multiplication":
        return a * b
```

```
  Cell In[156], line 1
    def fun(a , b = 2 , c ) :
                        ^
SyntaxError: parameter without a default follows parameter with a default
```

What is this error for? If you default an argument, the subsequent arguments must also be defaulted.
To fix this error, either C must also be defaulted, or B must be the last argument.

Tip: Default arguments must be followed by other default arguments. To resolve this, either provide a default value for the subsequent argument or reorder the arguments.

[1] حسابی
[2] هندسی

```
def num(*args) :
    print(args)

num(1,2,3,4,5)

(1, 2, 3, 4, 5)

num(1)

(1,)

#you can use other name instead of args
def num(*inputs) :
    print(inputs, type(inputs))

num(4,5)

(4, 5) <class 'tuple'>

def avg(*inputs):
   return sum(inputs) / len(inputs)

avg(1,2,3,4,5,6)

3.5
```

- **Unknown numbers of inputs**: *args and **kwargs are used to handle functions with an unknown number of inputs:
  - ***args:** *args receives all inputs and converts them into a **tuple**. However, anything else can be used instead of *args, and only the * is important. *args allows a function to accept any number of positional arguments.

Tip : "Pass" is a placeholder[1] statement for unimplemented code. It does nothing but prevents errors when a statement is required.

```
#avg function with *args
def avg(*inputs , mod) :
    if mod == "geometric" :
        pass  # pass and later you will write the statement
    if mod == "arithmetic" :
        return sum(inputs) / len(inputs)

avg(1,2,3,4, mod ="arithmetic")

2.5
```

```
#avg function with *args
#geo-avg function is also called in this function
def avg(*inputs, mod):
    if mod == "geometric":
        return geo_avg(*inputs) #calling geo_avg with *inputs
    elif mod == "arithmetic":
        return sum(inputs) / len(inputs)

def geo_avg(*inputs):
    result = 1
    for i in inputs:
        result *= i
    return result ** (1/len(inputs))

print(avg(1, 4, mod="geometric"))
print(avg(1, 4, mod="arithmetic"))

2.0
2.5
```

  - ****kwargs**: **kwargs receives all inputs and converts them into a **dictionary**. Kw is short for keyword.

However, anything else can be used instead of **kwargs, and only the ** is important. **kwargs allows a function to accept any number of keyword arguments (named arguments).

```
def avg(**inputs):
    result = {}
    for name, scores in inputs.items():
        average = sum(scores) / len(scores)
        result[name] = average
    return result

avg(sara=(17, 18, 19), iman=(16, 17, 18), ali=(18, 19, 20))

{'sara': 18.0, 'iman': 17.0, 'ali': 19.0}

#Can functions include other inputs besides **kwargs? Yep
def avg(a, **inputs):
    result = {}
    for name, scores in inputs.items():
        average = sum(scores) / len(scores)
        result[name] = average
    return result , a*2

avg(13 , sara=(17, 18, 19), iman=(16, 17, 18), ali=(18, 19, 20))

({'sara': 18.0, 'iman': 17.0, 'ali': 19.0}, 26)

#is it possible to have *args and **kwargs in a single function? Yep
#a is positional arguments, *b is args and **inputs is kwargs
def avg(a, *b, **inputs):
    result = {}
    for name, scores in inputs.items():
        average = sum(scores) / len(scores)
        result[name] = average
    return result , a*2, b

avg(13,3,4,5,6, sara=(17, 18, 19), iman=(16, 17, 18), ali=(18, 19, 20))

({'sara': 18.0, 'iman': 17.0, 'ali': 19.0}, 26, (3, 4, 5, 6))
```

```
def num(**inputs) :
    return inputs

num(1,2,3)  #you gotta define a key, value

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[118], line 1
----> 1 num(1,2,3)

TypeError: num() takes 0 positional arguments but 3 were given

num(in1 = 1 , in2 = (3,4))

{'in1': 1, 'in2': (3, 4)}
```

```
#D is defaulted.
def print_args(a, b, *c, d=True, **e):
    print(a)
    print(b)
    print(c)
    print(d)
    print(e)

print_args(2,4,2,3,True,"str",sara = (15,16,17) , ali = (17,18,19))

2
4
(2, 3, True, 'str')
True
{'sara': (15, 16, 17), 'ali': (17, 18, 19)}
```

[1] نگهدارنده

- Lambda ˈlamdə/ Lambda input1, input2, ... : expression

lambda creates an anonymous function (a function without a name) in **a single expression**.
A lambda function can take **any** number of arguments, but can **only** have one expression.

```
#normal function
def avg(*a) :
    return sum(a) / len(a)
avg (1,2,3)
```

```
2.0
```

```
#Lambda function
lambda a :sum(a) / len(a)
```

```
<function __main__.<lambda>(a)>
```

```
#you need to set a name for landa function
avg_lambda = lambda *a :sum(a) / len(a)
avg_lambda(2,3,4)
```

```
3.0
```

- Multiple expressions:

To include multiple expressions in a lambda function, enclose the expressions in parentheses to create a tuple. The function will return a tuple containing the results of each expression.

```
#A lambda function can take any number of arguments
sum_l = lambda a , b , c : a + b - c
```

```
sum_l(4,5,6)
```

```
3
```

```
#multiple expressions
sum_l = lambda a , b , c : (a+b , a+c, b+c)
```

```
sum_l(4,5,6)
```

```
(9, 10, 11)
```

```
#Write a function to store student names and averages in a dictionary.
grades1 = { "ali" : (17,18,19) , "mona" : (19,20,20) }
grades2 = { "nazi" : (15,16,17) , "nina" : (14,15,16) }
avg = lambda grades : {key : sum(value) / len(value) for key, value in grades.items()}
```

```
avg(grades1)
```

```
{'ali': 18.0, 'mona': 19.666666666666668}
```

```
avg(grades2)
```

```
{'nazi': 16.0, 'nina': 15.0}
```

- examples

```
#Example 1: Prime numbers:
#Define a function to determine if a given number is prime
def prime_check(num) :
    for i in range(2, num):
        if num % i == 0 :
            print ("This is not a prime number")
            break
        else:
            print ("This is  a prime number")
            break
```

```
prime_check(8)
```

```
This is not a prime number
```

```
#or
def prime_check(num) :
    status = "prime" #inital status
    for i in range(2, num):
        if num % i == 0 :
            status = "not prime"
            break
    return status
```

```
prime_check(11)
```

```
'prime'
```

```
#Example 2: Factorial function
#Define a function to calculate the factorial of a given number
def factorial(num) :
    result = 1
    for i in range(2,num+1):
        result *= i
    return result
```

```
factorial(5)
```

```
120
```

```
#or
def factorial(num) :
    m = 1
    result = 1
    while m <= num:
        result *= m
        m += 1
    return result
```

```
factorial(4)
```

```
24
```

```python
#Example 3: Fibonacci function
#Write a function to generate the Fibonacci sequence up to a given number
#for example : 10 : 0,1,1,2,3,5,8
#my method
def fib(limit) :
    num0 , num1 , num2 = 0 , 1 , 0
    print(num0 , num1 , end = " ")
    while num2 < limit :
        num2 = num0 + num1
        if num2 <= limit :
            print(num2 , end = " ")
            num0 , num1 = num1 , num2
```

```python
fib(10)
```

```
0 1 1 2 3 5 8
```

```python
#or the sequence can be put in a list.
def fibonacci (limit) :
    num1, num2 = 0, 1
    nums = [num1,num2]
    while True : #This  while loop is used to enable the break statement
        num1, num2 = num2, num1+num2
        if num2 > limit :
            break
        nums.append(num2)
    return(nums)
```

```python
fibonacci(16)
```

```
[0, 1, 1, 2, 3, 5, 8, 13]
```

```python
#Example 4: Palindrome
#Write a function to determine if a given string is a palindrome .
#(reads the same forwards and backward), such as "Anna"
#my method
def palindrome(string):
    reversed_s = string[::-1]
    if string.lower() == reversed_s.lower() :  #lower() converts all characters in the string to lowercase
        return "the string is palindrome"
    else:
        return "the string is not palindrome"
```

```python
palindrome("hey")
```

```
'the string is not palindrome'
```

```python
palindrome("Anna")
```

```
'the string is palindrome'
```

```python
#or you can simply use lambda to create the function
pal = lambda string : "pal" if string.lower() == string.lower()[::-1] else "not pal"
```

```python
pal("Abc")
```

```
'not pal'
```

```python
pal("Pop")
```

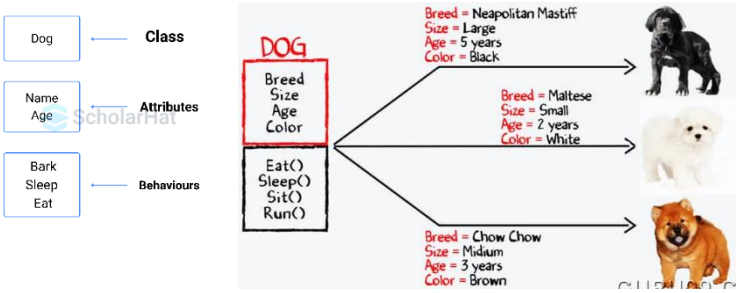
```
'pal'
```

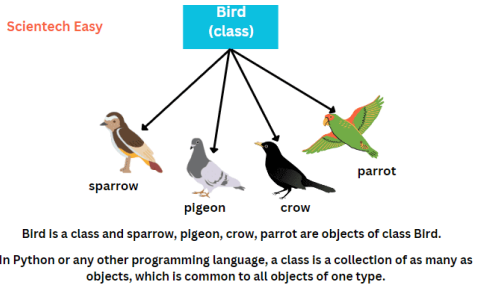
- **Section 13 : Class**

Classes provide a way to create **objects** that encapsulate both **data (attributes)** and **behaviour (methods)** into a single unit.
Python is an **object-oriented** programming language.

- **OOPs** Object-oriented programming (OOP) provides a methodology or paradigm for designing a program using **classes** and **objects**.
  - **Instance:** An object is an instance of a class.
  - **Data attribute:** The value that we store in an object is called **data attribute.**
  - **Method(behavior):** A function defined in a class

Thus, a class in Python is a way of creating, organizing, and managing objects with a set of attributes and methods.
In simple words, a class contains data and methods that we can access or manipulate this data. (Scienentech Easy)





- **Flashback: variable as an object**

```
#each variable is an object
x = 5 + 2j
y = 5.5
z = "salam howsam"
```

```
x.__class__ , y.__class__ , z.__class__
```

```
(complex, float, str)
```

```
#Access a variable's attributes and functions using dot and tab.
#Attributes don't require parentheses to work(unlike functions)
#Classes vary in the number of functions and instances they contain.
y.imag, y.real , x.imag , x.real
```

```
(0.0, 5.5, 2.0, 5.0)
```

```
#orange ones are functions(methods) and blue ones are attribitues(instances)
y.
```

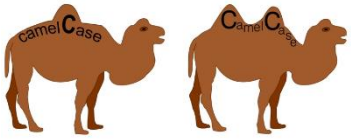
```
f as_integer_ratio function
f conjugate function
f fromhex function
f hex function
i imag instance
f is_integer function
i real instance
```

- **How to define a class:**

define a class→ make objects → call object

```
class class_name : #Class names are typically in camel case without underline.
    statements
    def function_name (self, inputs) :
        statements
        return outputs
```

```
class AreaCylinder :
```



Tip: self is the conventional name for the first parameter of instance methods in Python. It is a reference to the particular instance (object).
- self is just a strong convention and **not a keyword**. You can name the parameter anything, but always use self to keep code readable and idiomatic.
-Inside 'self', there is a series of important information. You can even store some information inside it.

```python
#id() gives you the memory address of an object.
a = "hello"
id(5), id(a)
```

```
(140727350405688, 2643694469264)
```

```python
#The purpose of this code block is to check the self ID
#and finally compare it to the object ID to see if they are the same
class  Animal :
    def check_self_id (self) :
        print(id(self))
```

```python
# id (Self) and id(animal1) are the same.
#because self is a reference to the object
animal1 = Animal()
animal1.check_self_id()
id(animal1)
```

```
2643787595072
2643787595072
```

```python
#Each new object instance gets its own unique id.
animal2 = Animal()
animal2.check_self_id()
id(animal2)
```

```
2643787647792
2643787647792
```

```python
#Within a class function, 'self' must be explicitly included as a parameter
#otherwise, Python will interpret the first argument as 'self'.
class Person:
    def name(n):
        print(f"the name is {n}")
```

```python
person1 = Person()
person1.name("ali")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[264], line 2
      1 person1 = Person()
----> 2 person1.name("ali")

TypeError: Person.name() takes 1 positional argument but 2 were given
```

```python
#corrected version
class Person:
    def name(self,n):
        print(f"the name is {n}")
person1 = Person()
person1.name("ali")
```

```
the name is ali
```

```python
#example one: class with property
class Person :
    name = " "
    family = " "
    gender = " "
    age = 0
```

```python
person1 = Person()
person1.name = "ana"
person1.family = "ahmadi"
person1. age = 25
```

```python
#person1 is an instance of the person class.
person1.__class__ , person1.name , person1.gender , person1.age
```

```
(__main__.Person, 'ana', ' ', 25)
```

```python
person2 = Person()
person2.name = "ali"
```

```python
person2.name , person2.age, person2.family
```

```
('ali', 0, ' ')
```

```python
#example 2: class with method
class Person:
    def print_name(self) :
        print("sara")

    def print_age(self) :
        print("no age")

    def print_gender(self) :
        pass
```

```python
person1 = Person()
person1.print_name() ,person1.print_age(), person1.print_gender(),
```

```
sara
no age
(None, None, None)
```

```python
class Person:
    def print_name(self, name) :
        print(name)

    def print_age(self, age) :
        print(age)
```

```python
person1 = Person()
person1.print_name()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[158], line 2
      1 person1 = Person()
----> 2 person1.print_name()

TypeError: Person.print_name() missing 1 required positional argument: 'name'
```

```python
person1.print_name("mahsa")
```

```python
#example 3 : class with property & method
class Person :
    name = " "
    age = 0

    def print_name(self) :
        pass
    def print_age(self) :
        pass
```

```python
person1 = Person()
person1.name = "mahdis"
person1.age = 24
person1.name
```

```
'mahdis'
```

Tip: Within a function of a class, use self. variable name to access instance attributes(variables).

```python
#You can access instance variables like name and age within a function using self
class Person :
    name = " "
    age = 0
    def print_name(self) :
        print(self.name, self.age)
    def grade(self , x) :
        y = x ** 2
        return y
```

```python
person1 = Person()
person1.name = "sara"
person1.age = 20
person1.print_name()
```

```
sara 20
```

```python
person1.grade(10)
```

```
100
```

Tip: Can the local variable Y, which is defined within the grade function, be accessed by other functions?
Yes,
To do this, you must introduce the variable y to the object, which means **registering** it as an attribute of the object.
by using self.y = y

```python
class Person:
    name = " "

    def print_name(self):
        print(self.name)
        return self.y

    def grade(self, x):
        y = x ** 2   #  y is local variable (not accessible outside this method)
        self.y = y   # make it an attribute of the object
        return y
```

```python
person1 = Person()
person1.name = "sara"
```

```python
person1.print_name()
```

```
sara
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[236], line 1
----> 1 person1 print_name()

Cell In[232], line 6, in Person.print_name(self)
      4 def print_name(self):
      5     print(self.name)
----> 6     return self.y

AttributeError: 'Person' object has no attribute 'y'
```

```python
#What is the reason for this error?
#Execute the grade function first to assign a value to y and convert it into an attribute
person1.grade(5)
```

```
25
```

```python
person1.print_name()
```

```
sara
25
```

Tip: Instead of declaring attribute variables directly, they are typically defined inside an initializer function, commonly named init (short for *initialize*).

```python
class Person:
    def init(self, name, age):
        self.name = name #Registering the variable is required
        self.age = age

    def print_details(self):
        print(self.name,self.age)
```

```python
person1 = Person()
person1.init("mahdis" , 24)
person1.print_details()
```

```
mahdis 24
```

```python
person1.age = 25
person1.name = "sara"
person1.print_details()
```

```
sara 25
```

```python
person1.
```

```
i age          instance
f init         function
i name         instance
f print_details function
```

- Examples

```python
#example 1 : Write a class that calculates the average grade of students. Consider four subjects:math, physics, sports, and art, with coefficients of 3, 2, 1, 1, respectively.
class Students:
    units = {"math":3, "physics":2, "sports":1, "art":1}
    def scores(self, math, physics, sports, art):
        self.math = math
        self.physics = physics
        self.sports = sports
        self.art = art
    def average(self):
        return (self.math * self.units["math"] + self.physics * self.units["physics"] + self.sports * self.units["sports"] + self.art * self.units["art"]) / sum(self.units.values())
```

```python
student1 = Students()
student1.scores(15,16,17,18)
student1.average()
```

```
16.0
```

```python
#A more advanced way
class Students:
    units = {"math":3, "physics":2, "sports":1, "art":1}
    def scores(self, **score):# **kwargs receives all inputs and converts them into a dictionary(for functions with an unknown number of inputs)
        self.score = score #registering score
#Dictionaries are iterable, enabling the use of for loop.
    def average(self):
        return (sum([self.score[u] * self.units[u] for u in self.units.keys()])) / sum(self.units.values())
```

```python
student1 = Students()
student1.scores(math = 20, physics=19, sports=2, art=19)
student1.score , student1.average()
```

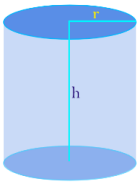
```
({'math': 20, 'physics': 19, 'sports': 2, 'art': 19}, 17.0)
```

```python
#the most advanced way to define average function
    def average(self):# using map() instead of a for loop.
        return sum(map(lambda x : self.score[x] * self.units[x],self.units.keys() )) / sum(self.units.values())#The lambda function performs multiplication.
```

```python
#example 2: Write a class that calculates the surface area and volume of a cylinder.
#v = p*r**2*h    s= h*2*r*p + 2*r**2*p
class cylinder:
    pi = 3.1416

    def init(self, r, h):
        self.r = r
        self.h = h

    def area_circle(self):
        return self.r**2 * self.pi

    def area_lateral(self):#مساحت جانبی
        return self.h * 2 * self.r * self.pi

    def area_cylinder(self):
        return 2*self.area_circle() + self.area_lateral() #Calling the previous two functions

    def volume__cylinder(self):
        return self.area_circle() * self.h
```

```python
c1 = cylinder()
c1.init(5,10)
c1.volume__cylinder() , c1.area_cylinder()
```

```
(785.3999999999999, 471.23999999999995)
```



Cylinder

$$\text{Volume} = \pi r^2 h$$

$$\text{Surface Area} = 2\pi r^2 + 2\pi r h$$


```python
#example 3 : Write a class that converts temperatures in Celsius to Kelvin, Fahrenheit, and Rankine
#k = T + 273.15
#F = 9/5 T + 32
#R = 9/5 T + 491.67
class CelsiusConverter:
    def init(self , T) :
        self.T = T

    def to_Kelvin(self):
        return self.T + 273.15
    def to_Fahrenheit(self):
        return (9/5)*self.T + 32
    def to_Rankine(self):
        return (9/5)*self.T + 491.67
```

```python
temp = CelsiusConverter()
temp.init(20)
temp.to_Fahrenheit(),temp.to_Kelvin(),temp.to_Rankine()
```

```
(68.0, 293.15, 527.6700000000001)
```

```python
#example 4 : Define a bank account class that stores a customer's name, account number, and balance
#and allows balance inquiries, deposits, and withdrawals.
class BankAccount :
    def init(self, name, ac_number, balance):
        self.name = name
        self.ac_number = ac_number
        self.balance = balance

    def balance_inquiry(self):#استعلام
        return self.balance

    def deposits(self, val):#واریز
        self.balance += val


    def withdrawals(self, val):#برداشت
        if self.balance < val :
            return "insufficient money"
        else:
            self.balance -= val
```

```python
person1 = BankAccount()
person1.init("mahdis", 1234 , 5000)
person1.deposits(2000)
person1.withdrawals(300)
person1.balance_inquiry()
```

```
6700
```

- Specially named methods

Specially named methods **are pre-defined methods**, similar to built-in functions, that simplify tasks within class definitions. They let you customize and streamline common behaviors of objects and classes **without** needing to call those methods explicitly.

- Method 1: __init__

The __init__() method is **called automatically** every time the class is used to create a new object.
It requires defining init inputs during creating an object(instantiation).
Therefore, __init__ reduces one line of code.

```python
#example 4(without using __init__ method)
class BankAcount:
    def init(self, name, ac_number, balance):
        self.name = name
        self.ac_number = ac_number
        self.balance = balance
    def balance_inquiry(self):
        return self.balance
    def deposits(self , val) :
        self.balance += val
    def withdrawls(self, val) :
        if self.balance < val :
            return "insufficient money"
        else :
            self.balance -= val
```

```python
person1 =  BankAcount()
person1.init("sara" , 1234 , 5000)
person1.deposits(300)
person1.withdrawls(1000)
person1.balance_inquiry()
```

```
4300
```

```python
#using __init__
class BankAcount:
    def __init__(self, name, ac_number, balance):
        self.name = name
        self.ac_number = ac_number
        self.balance = balance
    def balance_inquiry(self):
        return self.balance
    def deposits(self , val) :
        self.balance += val
    def withdrawls(self, val) :
        if self.balance < val :
            return "insufficient money"
        else :
            self.balance -= val
```

```python
#The error occurred because the object instantiation was missing the __init__ method's required inputs.
#Using __init__ streamlines object creation and reduces a line of code
person1 =  BankAcount()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[59], line 3
      1 #The error occurred because the object instantiation was missing the __init__ method's required inputs
      2 #Using __init__ streamlines object creation and reduces a line of code
----> 3 person1 =  BankAcount()

TypeError: BankAcount.__init__() missing 3 required positional arguments: 'name', 'ac_number', and 'balance'
```

```python
person1 =  BankAcount("mahdis" , 2345, 3000)
```

```python
#y = w*x + b
class Neuron:
    def __init__(self, w, b):
        self.w = w
        self.b = b
    def forward(self ,x) :
        y = self.w * x + self.b
        return y

neuron = Neuron(w= 0.5 , b =-1)
neuron.forward(x = 3.5)
```

```
0.75
```

- Example: Neurons

Neurons, also known as units or nodes, are the fundamental computational components of artificial neural networks.

- Method 2: __cal__

__call__ enables an object to be called like a function. A class can define only one __call__ method (because method names must be unique in a class) __call__ is better to be used in classes that have a **main function**.

```python
#without using __call__
#y = w*x + b
class Neuron:
    def __init__(self, w, b):
        self.w = w
        self.b = b
    def forward(self ,x) : #forward is the main function
        y = self.w * x + self.b
        return y

neuron = Neuron(w= 0.5 , b =-1)
neuron.forward(x = 3.5)
```

```
0.75
```

```python
#using __call__
class Neuron:
    def __init__(self, w, b):
        self.w = w
        self.b = b
    def __call__(self ,x) :
        y = self.w * x + self.b
        return y

neuron = Neuron(w= 0.5 , b =-1)
neuron(x = 3.5) #you can also write neuron.__call__(3.5)
#but neuron(3.5) is preferred
```

```
0.75
```

- Method 3: __repr__ repr is short for "representation".

__repr__ returns a printable representation of an object that can be customized or predefine.
In __repr__ function, only **a string** can be returned.

```python
neuron #The output <__main__.Neuron at 0x194b691b200> is a representation of neuron object
#Which is shown to us by calling it
```

```
<__main__.Neuron at 0x194b691b200>
```

```python
#you can customize the representation using __repr__ function.
class Neuron :
    def __init__(self, w,b):
        self.w = w
        self.b = b
    def __call__(self, x):
        y = self.w * x + self.b
        return y
    def __repr__(self):#No other input is required.
        return f"Neuron(w= {self.w}), (b ={self.b})"

neuron = Neuron(0.5 , -1)
neuron(3.5)
neuron
```

```
Neuron(w= 0.5), (b =-1)
```

```python
class Vector:
    def __init__(self, *items) :
        self.data = list(items)

a = Vector(1,2,3,4)
a.data , a
```

```
([1, 2, 3, 4], <__main__.Vector at 0x194b3776a50>)
```

```python
class Vector:
    def __init__(self, *items) :
        self.data = list(items)

    def __repr__(self):
        return str(self.data)

a = Vector(1,2,3,4)
a #simply write a instead of writing a.data because the  repr function is defined
```

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

- Example: Vector

  Lists resemble vectors in appearance, but differ significantly in function. For example, unlike **element-wise addition** in vectors, adding lists results in **concatenation**. Our goal is to create a list that behaves like a vector.

- Method 4: __len__

  __len__Returns the length of an object when called using the len() function.

```python
class Vector:
    def __init__(self, *items) :
        self.data = list(items)

    def __repr__(self):
        return str(self.data)

    def __len__(self): #no other input is required
        return len(self.data)

a = Vector(1,2,3,4)
len(a)
```

```
4
```

- **Method 5: __add__**

__add__ is used to define the behavior of the + operator for objects of a class. __add__(self, other) defines self + other.

Operations like addition, subtraction, multiplication, and division require operands of the same data type.

When you write a + b Python internally calls a.__add__(b)

```
print(5 + 10)           # 15
print((5).__add__(10))  # 15

print("Hello " + "World")          # "Hello World"
print("Hello ".__add__("World"))   # "Hello World
```

```
15
15
Hello World
Hello World
```

```
# Example 1: Concatenating Vector with a list
class Vector:
    def __init__ (self, *items) : #Inputs will become a tuple.
        self.data = list(items) # list the tuple
    def __add__(self, other) :
        return self.data + other
```

```
vector1 = Vector(1,2,3)
vector1.__add__([4,6,7])
```

```
[1, 2, 3, 4, 6, 7]
```

```
# Example 2: Concatenating two Vector objects
class Vector:
    def __init__(self, *items) :
        self.data = list(items)
    def __add__(self, other):
        return self.data + other.data
```

```
vector1 = Vector(1,2,3)
vector2 = Vector(4,5,6)
vector1.__add__(vector2)  #or vector1+ vector2
```

```
[1, 2, 3, 4, 5, 6]
```

You can override __add__ to customize what + means for your objects.

```
## Example 3: Element-wise vector addition (returns list)
#The purpose of these code blocks is to simulate the element-wise addition of vectors.
class Vector:
    def __init__(self, *items) :
        self.data = list(items)
    def __repr__(self):
        return str(self.data)
    def __add__(self, other):
        if len(self.data) == len(other.data):
            return [i+j for i , j in zip(self.data , other.data)] #the output gonna be of the list type
        else:
            return "Error :Vectors must be of the same length"
```

```
vector1 = Vector(1,2,3)
vector2 = Vector(4,5,6)
vector3 = vector1.__add__(vector2)  #or vector1+ vector2
print(vector3 , type(vector3) )
```

```
[5, 7, 9] <class 'list'>
```

```
#Example 4: Element-wise vector addition (returns Vector)
class Vector:
    def __init__(self, *items) :
        self.data = list(items)
    def __repr__(self):
        return str(self.data)
    def __add__(self, other):
        if len(self.data) == len(other.data):
            #return Vector(*[i+j for i , j in zip(self.data , other.data)]) # a good way
            return Vector(*map(lambda x , y : x+y , self.data , other.data)) #also a good way using map()
        else:
            return "Error :Vectors must be of the same length"
```

```
vector1 = Vector(1,2,3)
vector2 = Vector(4,5,6)
vector3 = vector1.__add__(vector2)  #or vector1+ vector2
print(vector3 , type(vector3) )
```

```
[5, 7, 9] <class '__main__.Vector'>
```

- **Method 6: __sub__**

__sub__ is used to define the behavior of the - operator for objects of a class. __sub__(self, other) defines self - other.

```
#Example 5: Element-wise vector Subtraction (returns Vector)
class Vector:
    def __init__(self, *items) :
        self.data = list(items)
    def __repr__(self):
        return str(self.data)
    def __sub__(self, other):
        if len(self.data) == len(other.data):
            #return Vector(*[i-j for i , j in zip(self.data , other.data)]) # a good way
            return Vector(*map(lambda x , y : x-y , self.data , other.data)) #also a good way using map()
        else:
            return "Error :Vectors must be of the same length"
```

```
vector1 = Vector(10,9,8)
vector2 = Vector(2,2,2)
vector1.__sub__(vector2)  #or vector1 -  vector2
```

```
[8, 7, 6]
```

- Method 7: __mul__

__mul__ is used to define the behavior of the * operator for objects of a class. __mul__(self, other) defines self * other.

```
#Example 6: Element-wise vector Multiplication (returns Vector)
class Vector:
    def __init__(self, *items) :
        self.data = list(items)
    def __repr__(self):
        return str(self.data)
    def __mul__(self, other):
        if len(self.data) == len(other.data):
            #return Vector(*[i*j for i , j in zip(self.data , other.data)]) # a good way
            return Vector(*map(lambda x , y : x*y , self.data , other.data)) #also a good way using map()
        else:
            return "Error :Vectors must be of the same length"
```

```
vector1 = Vector(1,2,3)
vector2 = Vector(4,5,6)
vector1.__mul__(vector2)  #or vector1 *  vector2
```

```
[4, 10, 18]
```

- Method 8: __truediv__

__truediv__ is used to define the behavior of the / operator for objects of a class. __truediv__(self, other) defines self / other.

```
#Example 7: Element-wise vector division (returns Vector)
class Vector:
    def __init__(self, *items) :
        self.data = list(items)
    def __repr__(self):
        return str(self.data)
    def __truediv__(self, other):
        if len(self.data) == len(other.data):
            #return Vector(*[i/j for i , j in zip(self.data , other.data)]) # a good way
            return Vector(*map(lambda x , y : x/y , self.data , other.data)) #also a good way using map()
        else:
            return "Error :Vectors must be of the same length"
```

```
vector1 = Vector(10,8,6)
vector2 = Vector(2,2,2)
vector1.__truediv__(vector2)  #or vector1 / vector2
```

```
[5.0, 4.0, 3.0]
```

- **Method 9: __getitem__**

__getitem__ (self, index) is called when you access an item using obj[index]. For example, If you pass a single index (like obj[2]), Python calls __getitem__(2).

When you implement __getitem__ and __setitem__ in a class, you're simulating the **indexing and slicing behavior** of built-in sequences (like lists, tuples, strings).

```
#If index is an int, you get back a single element ,If index is a slice, you get back a list, because Python list slicing always returns a list.
class Vector:
    def __init__(self, *items) :
        self.data = list(items)
    def __repr__(self):
        return str(self.data)
    def __getitem__(self, index) :
        return self.data[index]
```

```
vector1 = Vector(1, 2, 3, 4)
vector1.__getitem__(2) ,type(vector1.__getitem__(2))  #or vector1[2]
```

```
(3, int)
```

```
vector1[2:] , type(vector1[2:]) #the problem is this code returned a list not a  vector
```

```
([3, 4], list)
```

```
#If it's a single element (int), you return it, If it's a list (from slicing), you wrap it in a  Vector.
class Vector:
    def __init__(self, *items) :
        self.data = list(items)
    def __repr__(self):
        return str(self.data)
    def __getitem__(self, index) :
        out = self.data[index]
        return Vector(*out) if type(out) != int else out
```

```
vector1 = Vector(1, 2, 3, 4)
vector1[2] , type(vector1[2] )
```

```
(3, int)
```

```
vector1[2:] , type(vector1[2:])
```

```
([3, 4], __main__.Vector)
```

- **Method 10: __setitem__**

__setitem__ (self, index, value) is called when you assign a value using obj[index] = value.

If you assign (obj[2] = x), Python calls __setitem__(2, x)

```
class Vector:
    def __init__(self, *items) :
        self.data = list(items)
    def __repr__(self):
        return str(self.data)
    def __getitem__(self, index) :
        out = self.data[index]
        return Vector(*out) if type(out) != int else out
    def __setitem__(self, index , value) :
        self.data[index] = value
```

```
vector1 = Vector(1, 2, 3, 4)
vector1[2] = 5 #or vector1.__setitem__(2,5)
vector1
```

```
[1, 2, 5, 4]
```

```
vector1[:2] = 10,20  #Updating a Slice with a Tuple
vector1
```

```
[10, 20, 9, 4]
```

```
vector1[2:] = Vector(90,40) #Updating a Slice with Another Vector
vector1
```

```
[10, 20, 90, 40]
```

```
class Person:
    def __init__(self, name, age):
        self.name, self.age = name, age
    def __str__(self):
        return f"{self.name} is {self.age} years old"
    def __repr__(self):
        return f" {self.name} is an object of Person class"
```

```
p = Person("Ali", 25)
print(p) #or str(p)
```

```
Ali is 25 years old
```

```
p
```

```
 Ali is an object of Person class
```

- **Method 11: `__str__`** Returns a string representation of an object.

- Used when calling str(obj) or print(obj).
- Should return a **human-readable** string.

- **Method 12: `__delitem__`** Deletes an item from a collection object at a given index or key.

- Defines behavior for del obj[key].
- Usually used in custom containers.

```
class MyList:
    def __init__(self, *items):
        self.items = list(items) #bacuese'tuple' objects doesn't support item deletion
    def __delitem__(self, index):
        del self.items[index]
```

```
ml = MyList(1, 2, 3, 4)
del ml[1] #or ml.__delitem__(2)
print(ml.items)
```

```
[1, 3, 4]
```

- **Method 13: `__contains__`** Checks if an item is contained in a collection object.

- Defines behavior for in keyword → item in obj.
- Should return True or False.

```
class MySet:
    def __init__(self, *values):
        self.values = set(values)
    def __contains__(self, item):
        return item in self.values
```

```
s = MySet(1, 2, 3)
print(2 in s)
print(5 in s)
```

```
True
False
```

- **Method 14: `__eq__`** Checks if two **objects** are **equal**.

- Defines == (equality) between two objects.
- Should return True or False.

- **Method 15: `__lt__`** Checks if an **object** is **less** than another object.
  - Defines < (less than).
  - Useful in sorting.
- **Method 16: `__gt__`** Checks if an **object** is **greater** than another object.

- Defines > (greater than).

```
class Numbers:
    def __init__(self, volume):
        self.volume = volume
    def __eq__(self, other):
        return self.volume == other.volume
    def __lt__(self, other):
        return self.volume < other.volume
    def __gt__(self, other):
        return self.volume > other.volume
num1 = Numbers(10)
num2 = Numbers(11)
```

```
num1 > num2 , num1 < num2 , num1 == num2
```

```
(False, True, False)
```

- **Method 17: `__iter__`** Returns an iterator object for a collection object.

- Makes an object **iterable**.
- Should return an iterator (often self).
- Works with for ... in ....

- **Method 18: `__next__`** Returns the next item from an iterator object.

- Used with iterators.
- Defines how to get the **next element**.
- Raises StopIteration when no more elements.

```
class Counter:
    def __init__(self, limit):
        self.limit = limit
        self.current = 0
    def __iter__(self):
        return self
    def __next__(self):
        if self.current < self.limit:
            self.current += 1
            return self.current
        raise StopIteration
```

```
c = Counter(3)
print(next(c))
print(next(c))
print(next(c))
# next(c) → StopIteration
```

```
1
2
3
```

```
class MyRange:
    def __init__(self, start, end):
        self.start, self.end = start, end
    def __iter__(self):
        self.current = self.start
        return self
    def __next__(self):  # Needed for iteration
        if self.current < self.end:
            val = self.current
            self.current += 1
            return val
        raise StopIteration
```

```
for num in MyRange(1, 5):
    print(num, end = " ")
```

```
1 2 3 4
```