<Python> Programming Language

Python is a high-level, general-purpose, interpreted programming language.

1. **High-level**

Python is a high-level programming language that makes it easy to learn. Python doesn't require you to understand the details of the computer in order to develop programs efficiently.

2. **General-purpose**

Python is a general-purpose language. It means that you can use Python in various domains including:

- ➢ Web applications
- ➢ Big data application
- ➢ Testing
- ➢ Automation
- ➢ Data science, machine learning, and AI
- ➢ Desktop software
- ➢ Mobile apps

The targeted languages like SQL which can be used for querying data from relational databases.

3. **Interpreted**

Python is an interpreted language. To develop a python program, you write python code into a file called source code.

To execute the source code, you need to convert it to the machine language that the computer can understand. And the Python interpreter turns the source code, line by line, once at a time, into the machine code when the Python program executes.

Compiled languages like Java and C# use a compiler that compiles the whole source code before the program executes.

Why we need to learn Python?

Python increase your productivity. Python allows you to solve complex problems in less time and fewer lines of code. It's quick to make a prototype in Python.

Python becomes a solution in many areas across industries, from web applications to data science and machine learning.

Python is quite easy to learn in comparison with other programming languages.

Python syntax is clear and beautiful.

Python has a large ecosystem that includes lots of libraries and frameworks. Python is cross-platform. Python programs can run on Windows, Linux, and macOS.

Python has a huge community. Whenever you get stuck, you can get help from an active community.

Python developers are in high demand.

Python versions:

Python had two major versions: 2x and 3x.

Python 2.x was released in 2000.

Python 3.x was released in 2008. Basically, python 3 isn't compatible with python 2. And you should use the latest versions of Python 3 for new projects.

The latest Python version available in Python 3.10.0

Python Syntax

1. Python syntax:

```
# defining the main function to print out something
    def main():
    i=1
    max=10
    while(i<max)
        print(i)
        i=i+1
```

```
# calling the function main
    main()
```

By using indentation and white space to organize the code, Python code gains the following advantages:

- First, you'll never miss the beginning or ending code of a block like other programming languages such as Java or C#.
- Second, the code style is essentially uniform. If you have to maintain another developer's code, that code looks the same as yours.
- Third, the code is more readable and clear in comparison with other programming languages.

2. Python comments:

The comments are as important as the code because they describe why a piece of code was written.

When the python interpreter executes the code, it ignores the comments.

In Python, a single line comments with a hash (#) symbol:

```
# This is a single line comments in Python
```

3. Continuation of statements:

Python uses a newline character to separate statements. It places each statement on one line.

However, a long statement can span multiple lines by using the backslash (\) character.

The following example illustrates how to use the backslash (\) character to continue a statement in the second line:

e.g.

```
  Addition = 10 + \
             20 + \
             30 + \
             40 + \
             50
print(Addition)
```

4. Identifiers:

Identifiers are the names that identify variables, functions, modules, classes, and other objects in python.

The name of an identifier needs to be a letter or underscore (_).

Python identifiers are case-sensitive.

5. Keywords:

Some words have special meaning in python. They are called keywords.

The following shows the list of the keywords in python:

• False	• class
• finally	• return
• is	• None
• continue	• for
• lambda	• try
• True	• def
• from	• nonlocal
• while	• and
• del	• global
• not	• with

• as	• elif
• if	• or
• yield	• assert
• else	• import
• pass	• break
• except	• in
• raise	• async
• await	

Python is a growing and evolving language. So its keywords will keep increasing and changing.

Python provides a special module for listing its keywords called keyword.

To find the current keyword list, you can use:

import keyword

print(keyword.kwlist)

6. **String literals:**

Python uses single quotes (‘), double quote (“), triple single quotes (“””) to denote a string literal.

The string literal need to be surrounding with the same type of quotes.

Python Variables

In Python, a variable is a label that you can assign a value to it. And a variable is always associated with a value.

Creating variables:

To define a variable, you use the following syntax:

```
Variable_name=value
```

The = is the assignment operator. In this syntax, you assign a value to the variable_name.

The value can be anything like a number, a string, etc., that you assign to the variable.

The following defines a variable named counter and assign the number 1 to it:

```
counter = 1
```

Naming Variables:

When you name a variable, you need to adhere to some rules. If you don't follow, you'll het an error.

The following are the variable rules that you should keep in mind:

Variable names can contain only letters, numbers, and underscore (_), not with a number.

Variable names cannot contain spaces. To separate words in variables, you use underscore for e.g.: sorted_list.

Variables names cannot be the same as keywords, reserved words, and built-in function in Python.

For e.g.: the active_user variable in more descriptive than the au.

Use underscores (_) to separate multiple words in the variable names.

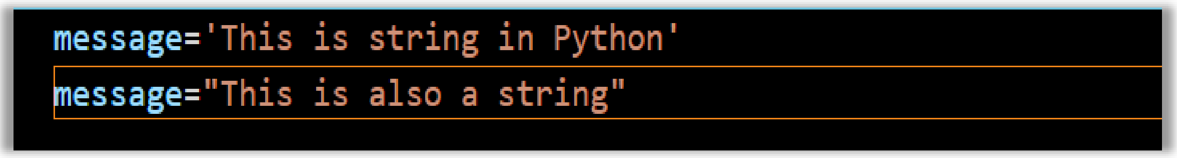
Avoid using the letter 1 and the uppercase letter O because they look like the number 1 and 0.

Python String

A string is a series of characters.

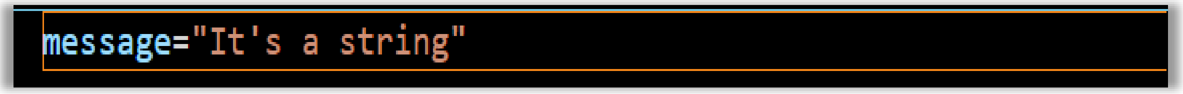
In python, anything inside quotes is a string. And you can use either single quotes or double quotes.

e.g.



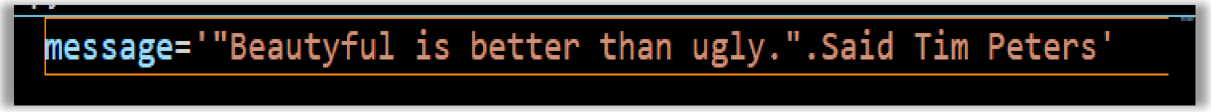
```
message='This is string in Python'
message="This is also a string"
```

If a string contains a single quote, you should place it in double-quotes like this:



```python
message="It's a string"
```

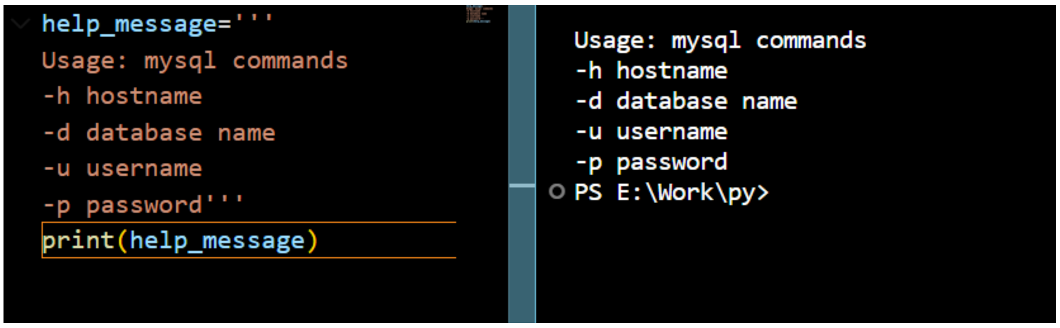
And when a string contains double quotes, you can use the single quotes:



```python
message='"Beautyful is better than ugly.".Said Tim Peters'
```

Creating multi-line strings:

To span a string multiple lines, you use triple-quotes “””””……””””” or “’…’”.



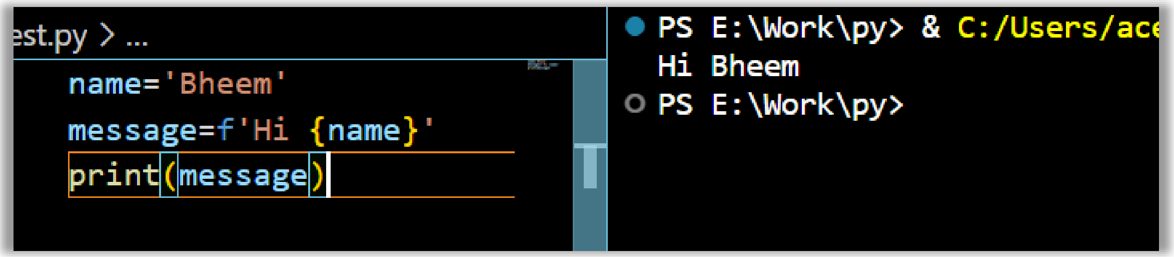
```python
help_message='''
Usage: mysql commands
-h hostname
-d database name
-u username
-p password'''
print(help_message)
```

```
Usage: mysql commands
-h hostname
-d database name
-u username
-p password
PS E:\Work\py>
```

Using variables in Python strings with the f-string:

Sometimes, you want to use the value of the variables in a string.

e.g. you may want to use the value of the name variable inside the message string variable:



```
est.py > ...
    name='Bheem'
    message=f'Hi {name}'
    print(message)
```

```
PS E:\Work\py> & C:/Users/ace
Hi Bheem
PS E:\Work\py>
```

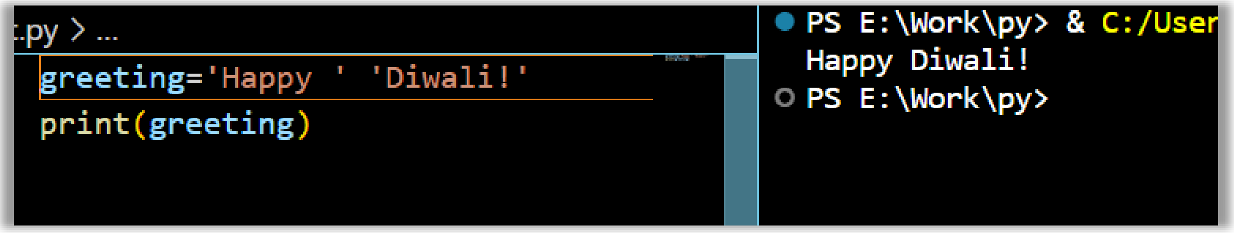
Python will replace the {name} by the value of the name variable.

This message is a format string, or f-string in short.

Concatenation Python strings:

When you place the string literals next to each other, Python automatically concatenates them into one string.

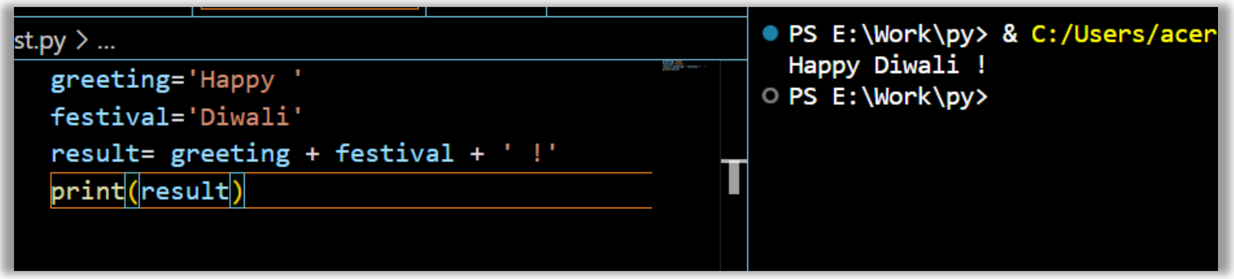
e.g.



```
.py > ...
greeting='Happy ' 'Diwali!'
print(greeting)
```

```
PS E:\Work\py> & C:/User
Happy Diwali!
PS E:\Work\py>
```

To concatenate two string variables, you use the operator +:



```
st.py > ...
    greeting='Happy '
    festival='Diwali'
    result= greeting + festival + ' !'
    print(result)
```

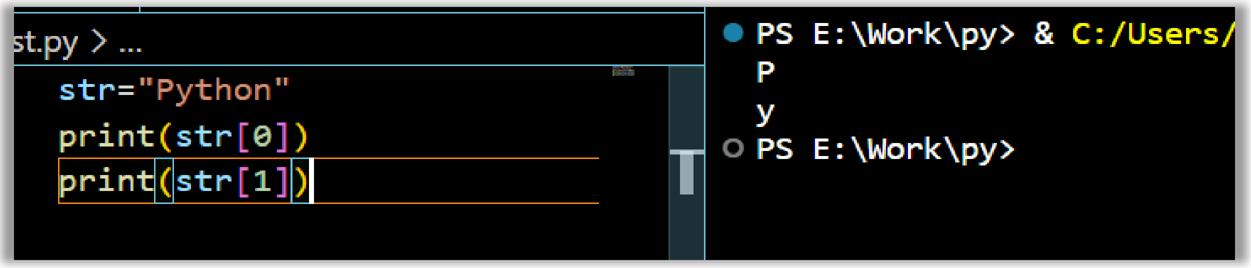
```
PS E:\Work\py> & C:/Users/acer
Happy Diwali !
PS E:\Work\py>
```

Accessing String Elements

Since a string is a sequence of characters, you can access its elements using an index.

The first character in the string has an index zero.

e.g.

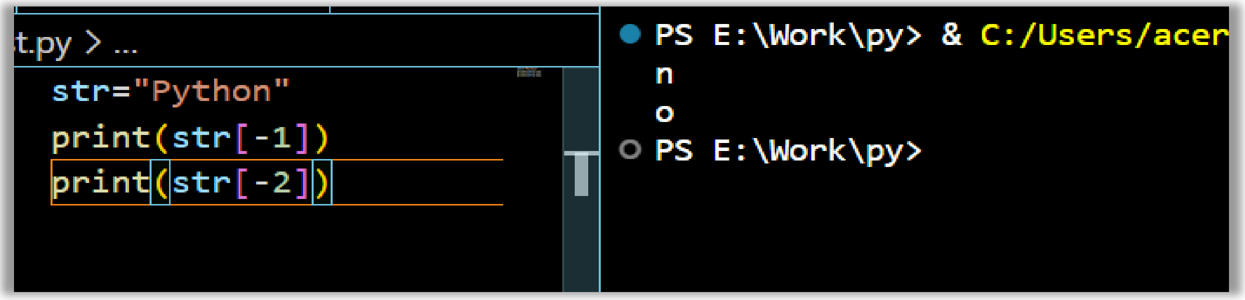


```
st.py > ...
    str="Python"
    print(str[0])
    print(str[1])
```

```
PS E:\Work\py> & C:/Users/
P
y
PS E:\Work\py>
```

Access the first and second characters of the string by using the square brackets [] and indexes.

If you use a negative index, Python returns the characters string from the end of the string.

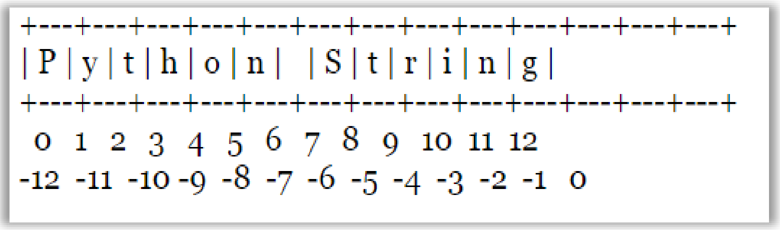


```
t.py > ...
    str="Python"
    print(str[-1])
    print(str[-2])
```

```
PS E:\Work\py> & C:/Users/acer
n
o
PS E:\Work\py>
```

the following illustrates the indexes of the string

"Python String":

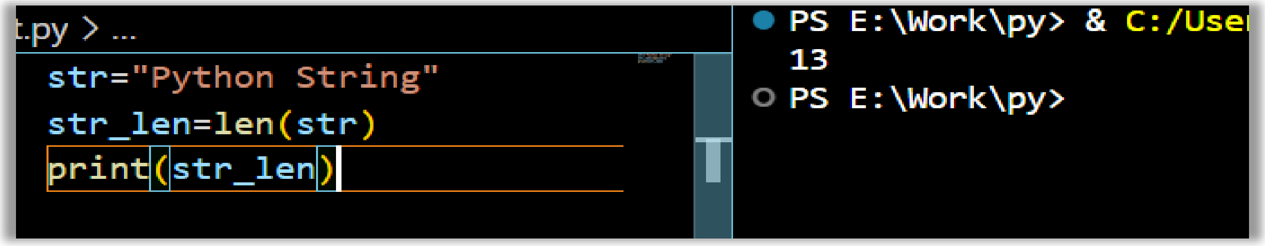


```
+---+---+---+---+---+---+---+---+---+---+---+---+---+
| P | y | t | h | o | n |   | S | t | r | i | n | g |
+---+---+---+---+---+---+---+---+---+---+---+---+---+
  0   1   2   3   4   5   6   7   8   9  10  11  12
-12 -11 -10  -9  -8  -7  -6  -5  -4  -3  -2  -1   0
```

Getting the length of a string:

To get the length of a string, you use the len() function.

e.g.



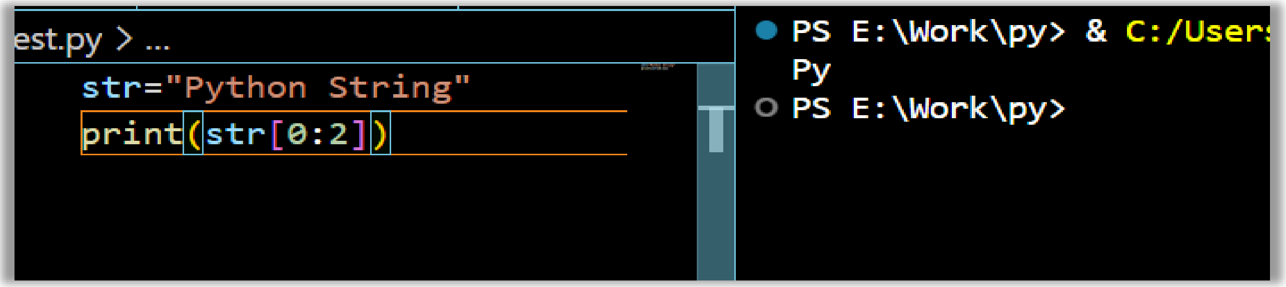
```
t.py > ...
  str="Python String"
  str_len=len(str)
  print(str_len)
```

```
● PS E:\Work\py> & C:/Use
  13
○ PS E:\Work\py>
```

Slicing string:

Slicing allows you to get a substring from a string.

e.g.



```
est.py > ...
    str="Python String"
    print(str[0:2])
```

```
● PS E:\Work\py> & C:/Users
  Py
○ PS E:\Work\py>
```

The str[0:2] returns a substring that includes the character from the index 0 (included) to 2 (excluded).

Syntax e.g.

string[start:end]

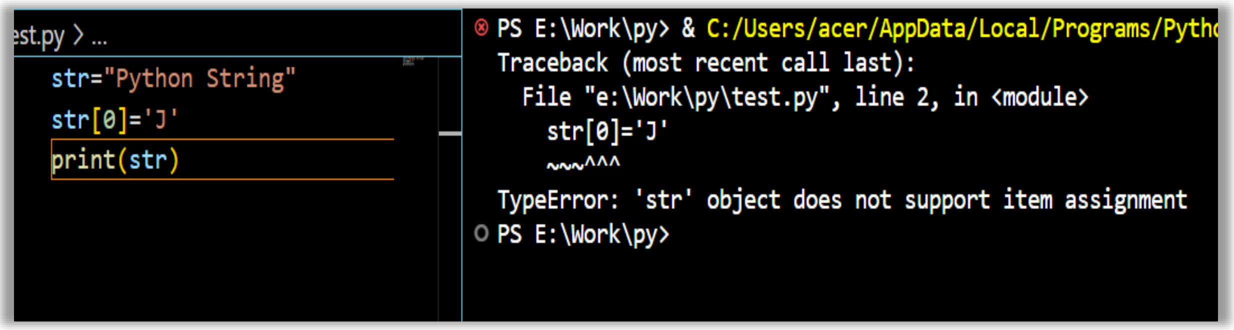
the substring always includes the character at the start and excludes the string at the end.

The start and end are optional. If you omit the start, it defaults to zero. If you omit the end, it defaults to the string’s length.

Python strings are immutable:

Python strings are immutable. It means that you cannot change the string.

e.g. you’ll get an error if you update one or more characters in a string:

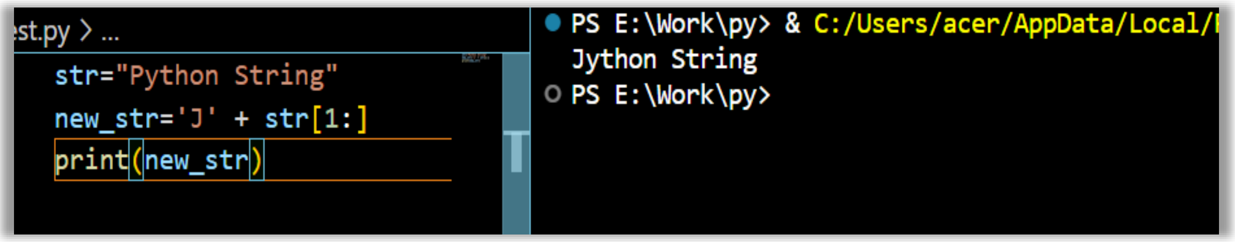


```
est.py > ...
    str="Python String"
    str[0]='J'
    print(str)
```

```
⊗ PS E:\Work\py> & C:/Users/acer/AppData/Local/Programs/Pytho
  Traceback (most recent call last):
    File "e:\Work\py\test.py", line 2, in <module>
      str[0]='J'
      ~~~^^^
  TypeError: 'str' object does not support item assignment
○ PS E:\Work\py>
```

When want to modify a string, you need to create a new one from the existing string.

e.g.



```
st.py > ...
    str="Python String"
    new_str='J' + str[1:]
    print(new_str)
```

```
● PS E:\Work\py> & C:/Users/acer/AppData/Local/
  Jython String
○ PS E:\Work\py>
```

Python Numbers

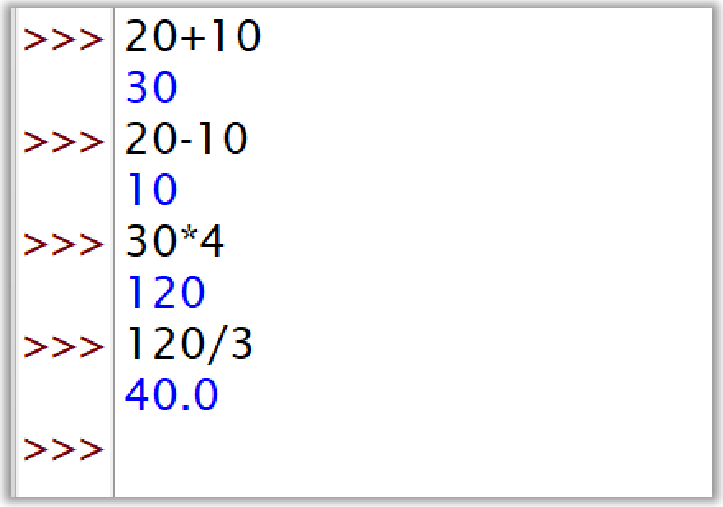
Python supports integers, floats, and complex numbers.

Integers:

The integers are numbers such as -1, 0, 1, 2, 3, … and they have type int.

You can use Math operators like +, -, *, and / to form expressions that include integers.

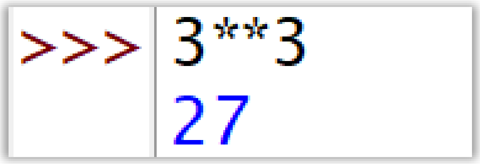
e.g.



```
>>> 20+10
30
>>> 20-10
10
>>> 30*4
120
>>> 120/3
40.0
>>>
```

To calculate exponents, you use two multiplication symbol (**).

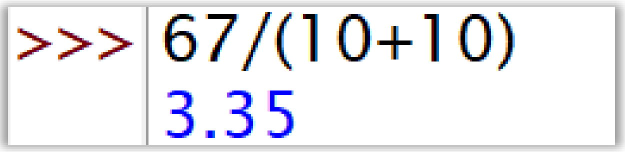
e.g.



```
>>> 3**3
27
```

To modify the order of operations, you use the parentheses().

e.g.

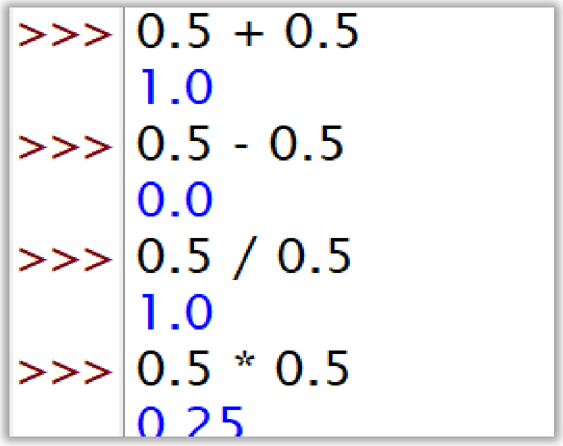


```
>>> 67/(10+10)
3.35
```

Floats:

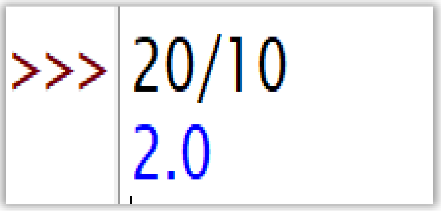
Any number with a decimal point is a floating-point number. The term float means that the decimal point can appear at any position in a number.

e.g.



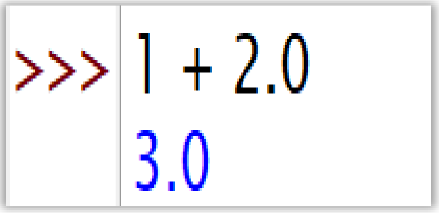
```
>>> 0.5 + 0.5
1.0
>>> 0.5 - 0.5
0.0
>>> 0.5 / 0.5
1.0
>>> 0.5 * 0.5
0.25
```

The division of two integer always returns a float:



```
>>> 20/10
2.0
```

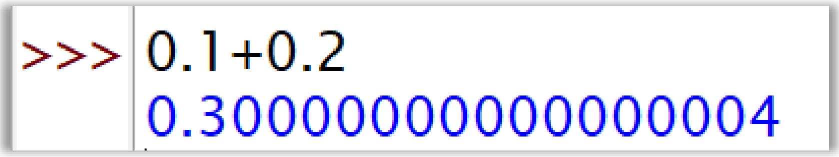
If you mix an integer and a float in any arithmetic operation, the result is a float:



```
>>> 1 + 2.0
3.0
```

Due to internal representation floats, Python will try to represent the result as precisely as a possible. However, you may get the result that you would not expect.

e.g.



```
>>> 0.1+0.2
0.30000000000000004
```

Underscores in numbers:

When a number is large, it'll become difficult to read.

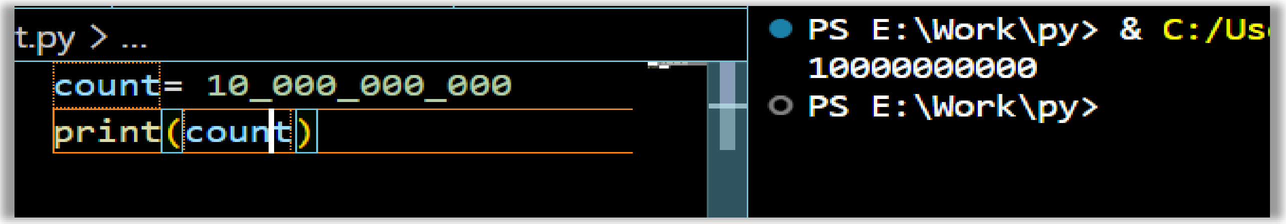
e.g.

count = 10000000000

To make the long numbers more readable, you can group digits using underscore, like this:

count = 10_000_000_000

when storing these values, Python just ignore the underscores. It does so when displaying the numbers with underscores on screen:



```
t.py > ...
count= 10_000_000_000
print(count)
```

```
PS E:\Work\py> & C:/Us
10000000000
PS E:\Work\py>
```

the underscores also work for both integers and floats.

Python Boolean

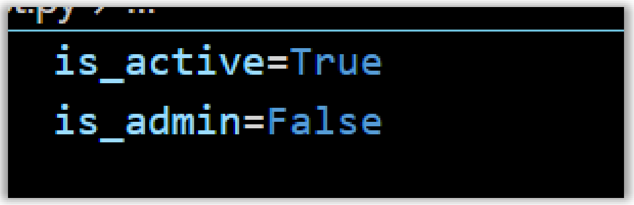
To represent true and false, Python provides you with a=the Boolean data type. The Boolean value has a technical name as bool.

The Boolean data type has two vales.

True and False:

Note that the boolean values True and False start with the capital letters (T) and (F).

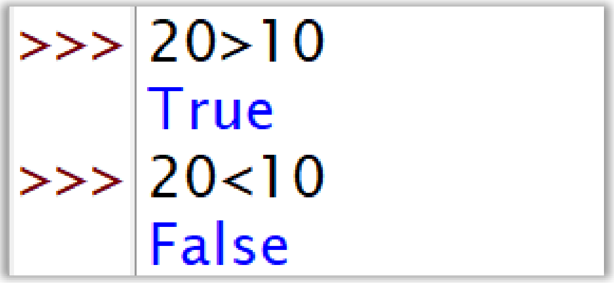
e.g.



```
is_active=True
is_admin=False
```

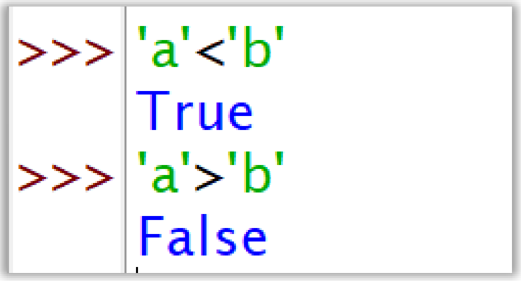
When you compare two numbers, Python returns the result as a boolean value.

e.g.



```
>>> 20>10
    True
>>> 20<10
    False
```

Also, comparing two strings results in a Boolean value:

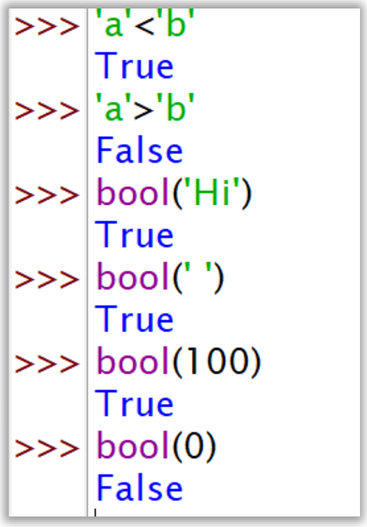


```
>>> 'a'<'b'
True
>>> 'a'>'b'
False
```

The bool() function:

To fine out if a value is True or False, you use the bool() function.

e.g.:



```
>>> 'a'<'b'
True
>>> 'a'>'b'
False
>>> bool('Hi')
True
>>> bool(' ')
True
>>> bool(100)
True
>>> bool(0)
False
```

Falsy and Truthy values:

When a value evaluates to True, it's truthy. And is a value evaluates to False, it's falsy.

The following are falsy values in Python:

- The number zero (0)

- An empty string ‘ ‘ .
- False.
- None.
- An empty list [].
- An empty tuple().
- An empty dictionary {}.

The truthy values are the other values that aren’t falsy.

Python Constants

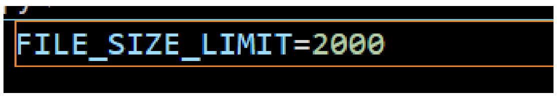
Sometimes, you may want to store values in variables. But you don’t want to change these values throughout the execution of the program.

The constants like variables but their values don’t change during the program execution.

The bad news is that Python doesn’t support constants,

To work around this, you use all capital letters to name a variable to indicate that the variable should be treated as a constants.

e.g.



```
FILE_SIZE_LIMIT=2000
```

When encountering variables like these, you should not change their values.

These variables are constant by convention, not by rules.

Comparison Operators

Python has six comparison operators, which are as follow:

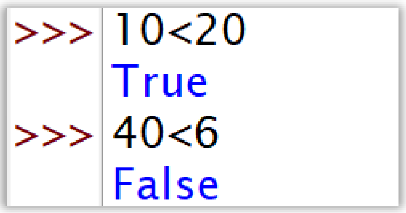
- Less than (<)
- Less than or equal to (<=)
- Greater than (>)
- Greater than or equal to (>=)
- Equal to (==)
- Not equal to (!=)

These comparison operators compare two values and return a Boolean value, either True or False.

And you can use these comparison operators to compare both numbers and string.

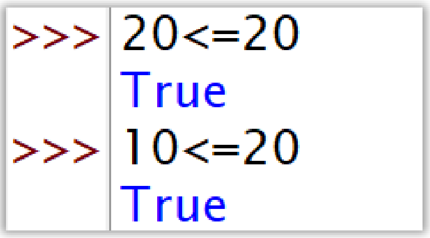
Less than (<):

e.g.



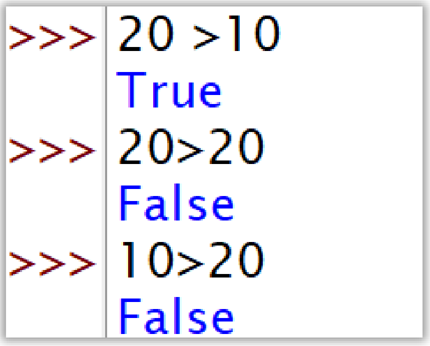
```
>>> 10<20
True
>>> 40<6
False
```

Less than or equal to (<=):



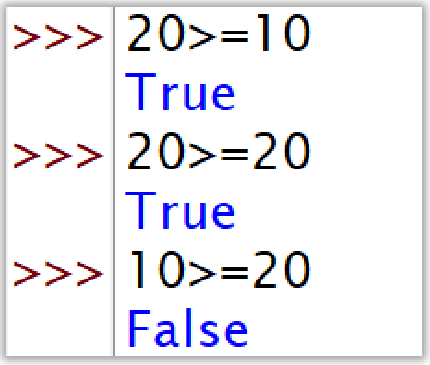
```
>>> 20<=20
True
>>> 10<=20
True
```

Greater than (>):



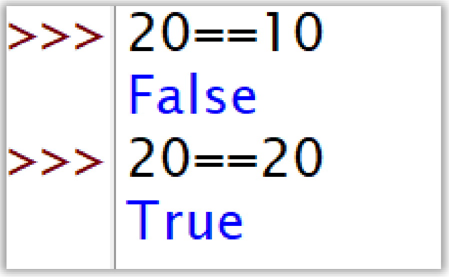
```
>>> 20 >10
True
>>> 20>20
False
>>> 10>20
False
```

Greater than or equal to (>=):



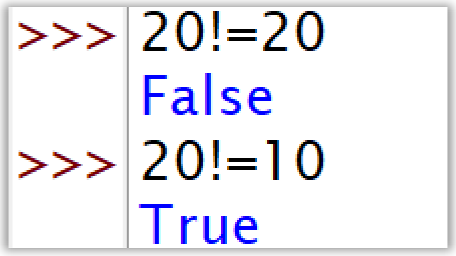
```
>>> 20>=10
True
>>> 20>=20
True
>>> 10>=20
False
```

Equal to (==):



```
>>> 20==10
False
>>> 20==20
True
```

Not equal to (!=):



```
>>> 20!=20
False
>>> 20!=10
True
```

Logical Operators

Python has three logical operators:

- and
- or
- Not

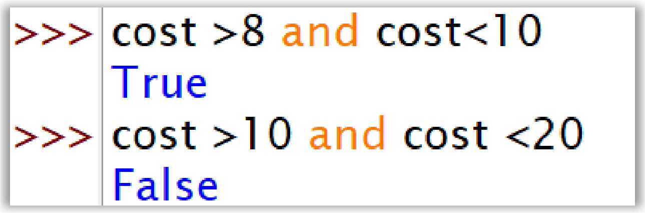
The and operator:

The and operator checks whether two conditions are both simultaneously.

a and b

It returns True if both conditions are True. And it returns False if either the condition a or b is False.

e.g.



```
>>> cost >8 and cost<10
True
>>> cost >10 and cost <20
False
```

The following table illustrates the result of the and operator when combining two conditions:

a	b	a and b
True	True	True
True	False	False
False	False	False
False	True	False

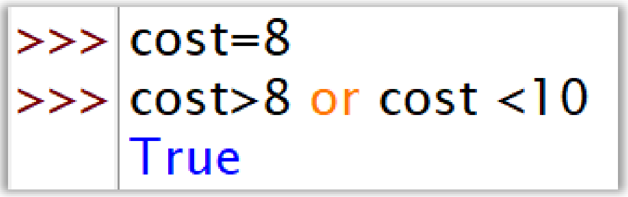
The or operator:

The or operator checks multiple conditions. But it returns True when either or both of individual conditions are True:

a and b

The or operator returns False only when both conditions are False.

e.g.



```
>>> cost=8
>>> cost>8 or cost <10
    True
```

The following table illustrates the result of the or operator when combining two conditions:

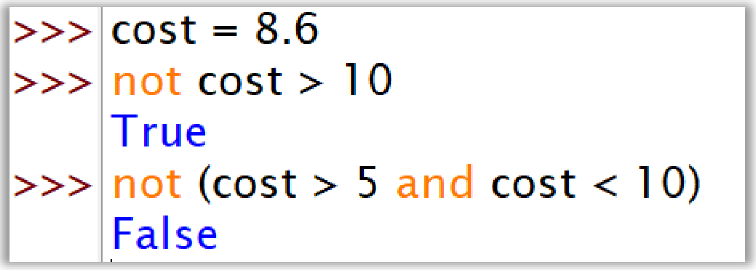
a	b	a or b
True	True	True
True	False	True
False	False	True
False	True	False

The not operator:

The not operator applies to one condition.

And it reverses the result of that condition, True become False and False become True.

e.g.



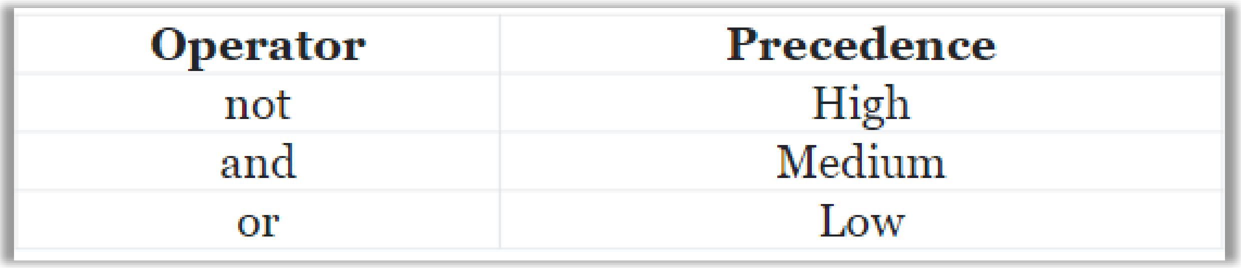
```
>>> cost = 8.6
>>> not cost > 10
True
>>> not (cost > 5 and cost < 10)
False
```

The following table illustrates the result of the not operator.

a	not a
True	False
False	True

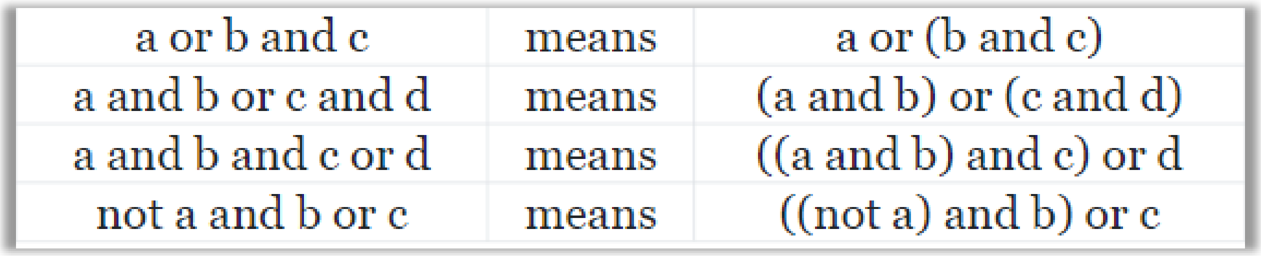
Precedence of Logical Operators:

When you mix the logical operators in an expression, Python will evaluate them in the order which is called the operator precedence.



Operator	Precedence
not	High
and	Medium
or	Low

In case an expression has several logical operators with the same precedence, Python will evaluate them from left to right:



a or b and c	means	a or (b and c)
a and b or c and d	means	(a and b) or (c and d)
a and b and c or d	means	((a and b) and c) or d
not a and b or c	means	((not a) and b) or c

If statement

If statement is use to execute a block of code based on a specified condition.

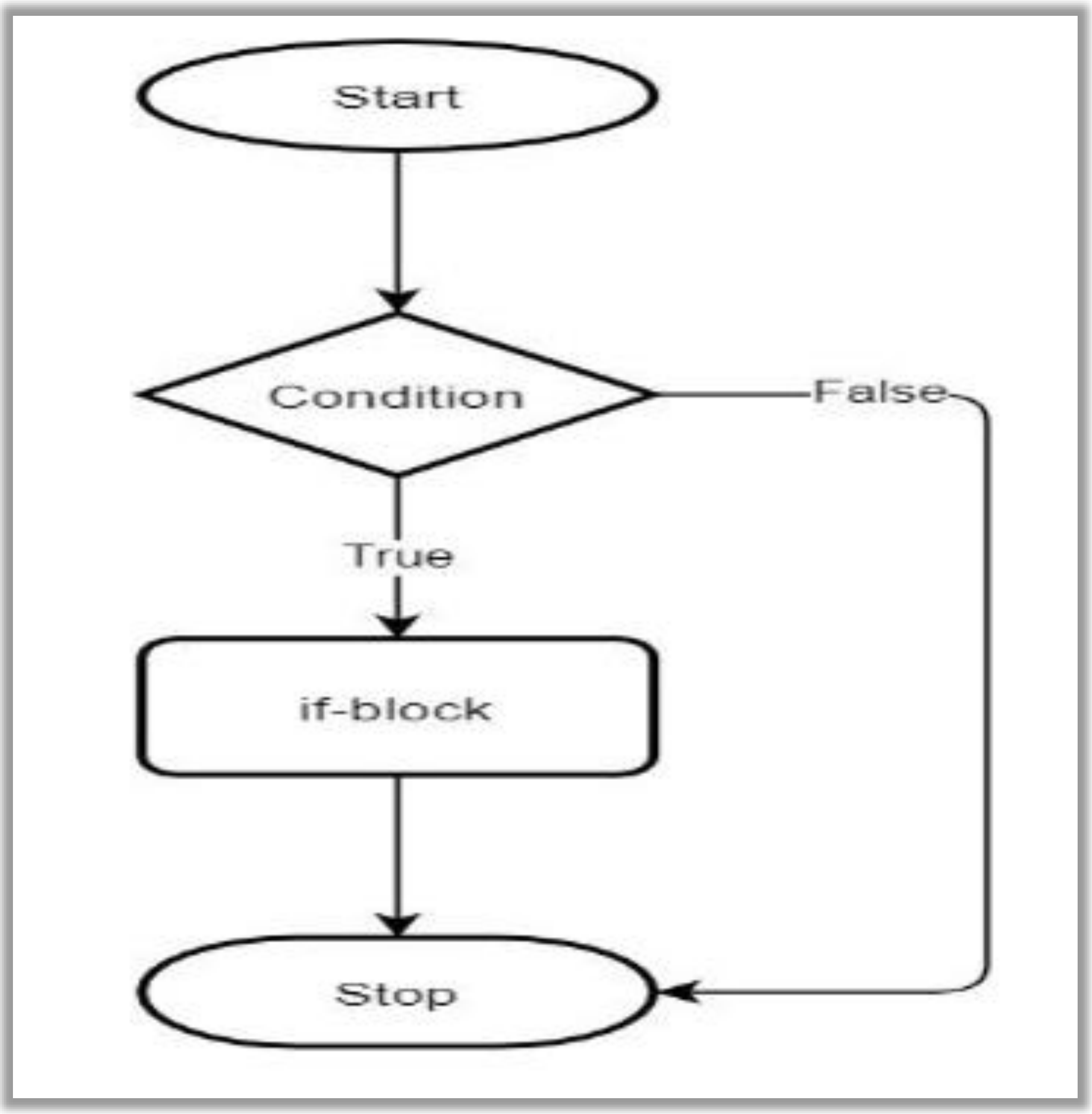
The syntax of if statement is as follow:

```
if condition:
    if-block
```

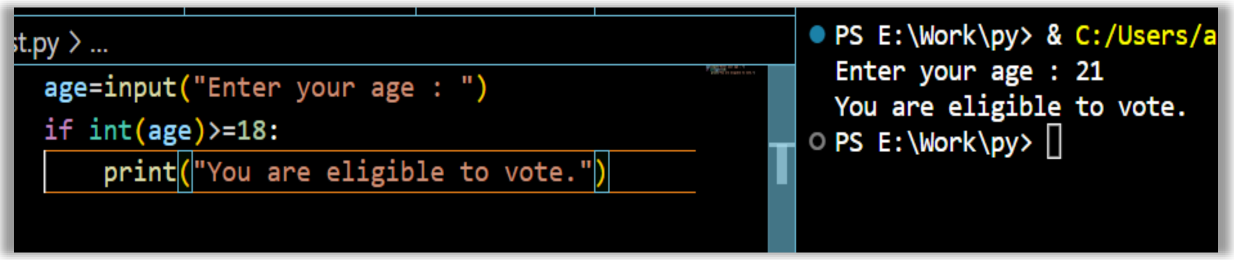
The if-statement checks the condition first.

If the condition evaluates to True, it executes the statements in the if-block. Otherwise, it ignore the statements.

Note that colon (:) that follow the condition is very important.



e.g.

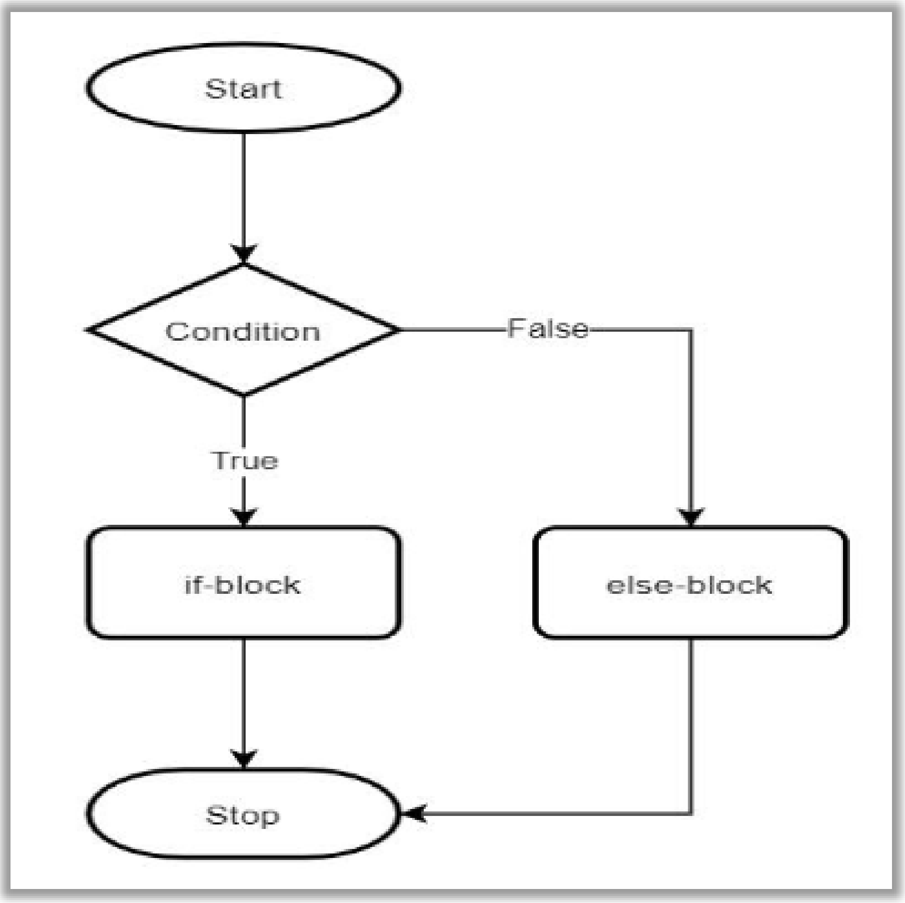


```
st.py > ...
  age=input("Enter your age : ")
  if int(age)>=18:
      print("You are eligible to vote.")
```

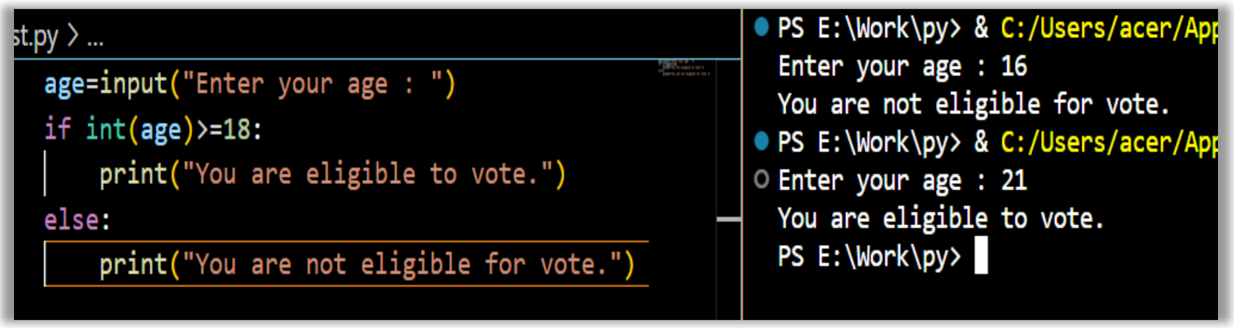
```
● PS E:\Work\py> & C:/Users/a
  Enter your age : 21
  You are eligible to vote.
○ PS E:\Work\py> 
```

If…else statement

Typically, you want to perform an action when a condition is True and another action when the condition is False. To do so, you use the if…else statement.



e.g.

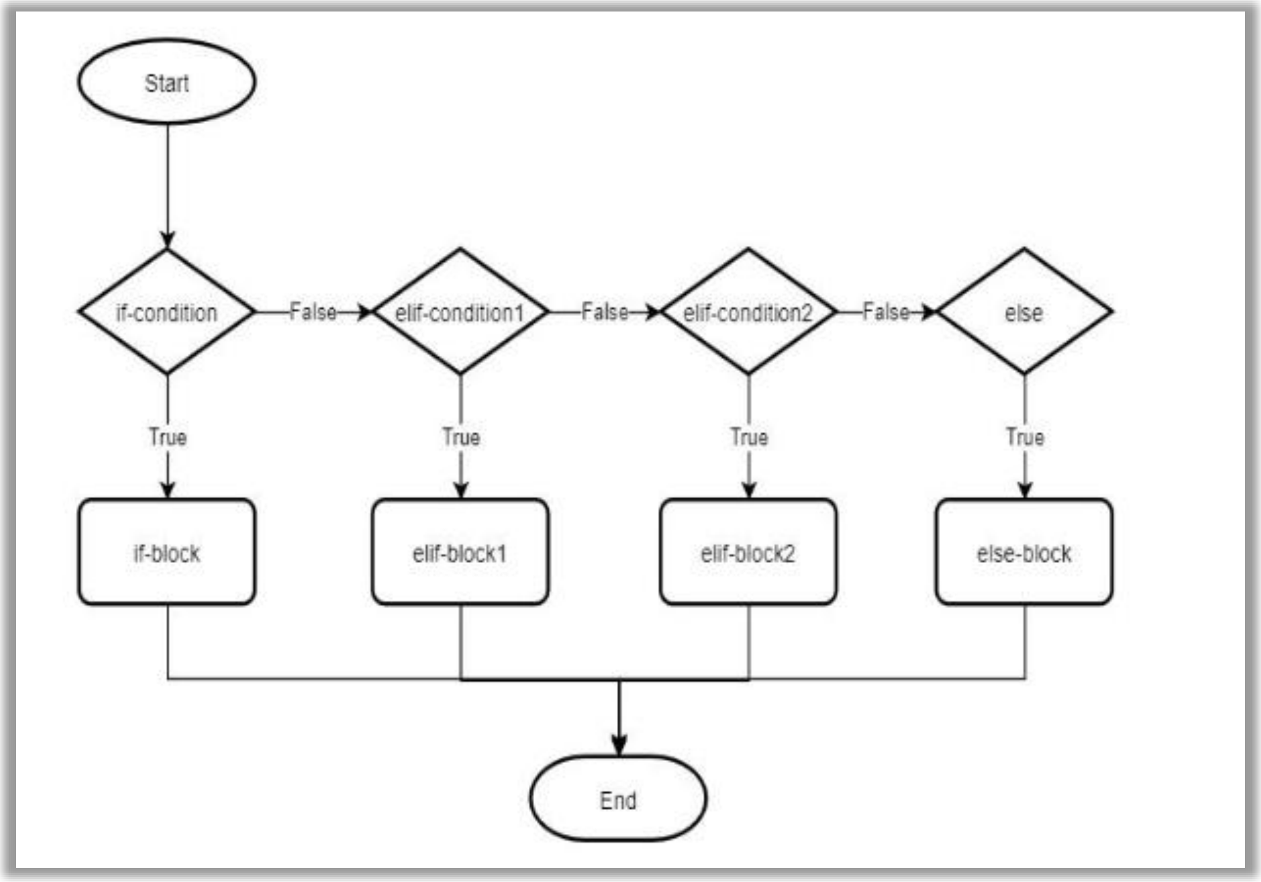


```
st.py > ...
age=input("Enter your age : ")
if int(age)>=18:
    print("You are eligible to vote.")
else:
    print("You are not eligible for vote.")
```

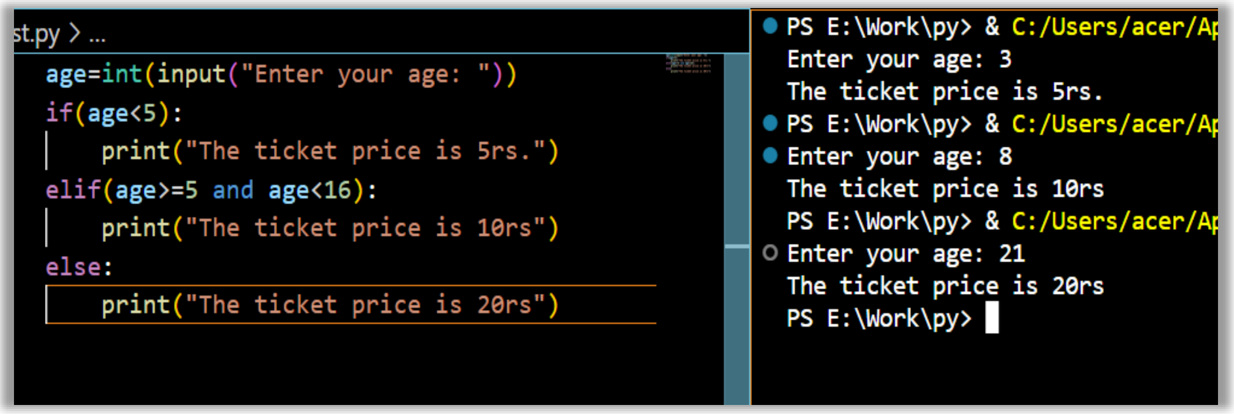
```
PS E:\Work\py> & C:/Users/acer/App
Enter your age : 16
You are not eligible for vote.
PS E:\Work\py> & C:/Users/acer/App
Enter your age : 21
You are eligible to vote.
PS E:\Work\py>
```

If…elif…else statement

If you want to check multiple conditions and perform an action accordingly, you can use if…elif…else statement.



e.g.



```
st.py > ...
age=int(input("Enter your age: "))
if(age<5):
    print("The ticket price is 5rs.")
elif(age>=5 and age<16):
    print("The ticket price is 10rs")
else:
    print("The ticket price is 20rs")
```

```
PS E:\Work\py> & C:/Users/acer/Ap
Enter your age: 3
The ticket price is 5rs.
PS E:\Work\py> & C:/Users/acer/Ap
Enter your age: 8
The ticket price is 10rs
PS E:\Work\py> & C:/Users/acer/Ap
Enter your age: 21
The ticket price is 20rs
PS E:\Work\py>
```

Ternary Operator

The following syntax is called a ternary operator:

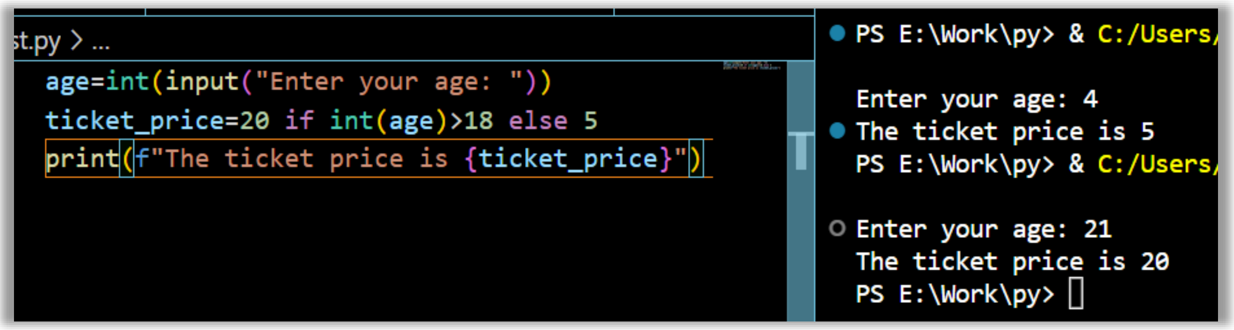
value_if_true if condition else value_if_false

The ternary operator evaluate the condition.

If the result is Ture, it returns the value_if_true.

Otherwise, it returns the value_if_false.

e.g.



```
st.py > ...
age=int(input("Enter your age: "))
ticket_price=20 if int(age)>18 else 5
print(f"The ticket price is {ticket_price}")
```

```
PS E:\Work\py> & C:/Users/

Enter your age: 4
The ticket price is 5
PS E:\Work\py> & C:/Users/

Enter your age: 21
The ticket price is 20
PS E:\Work\py>
```

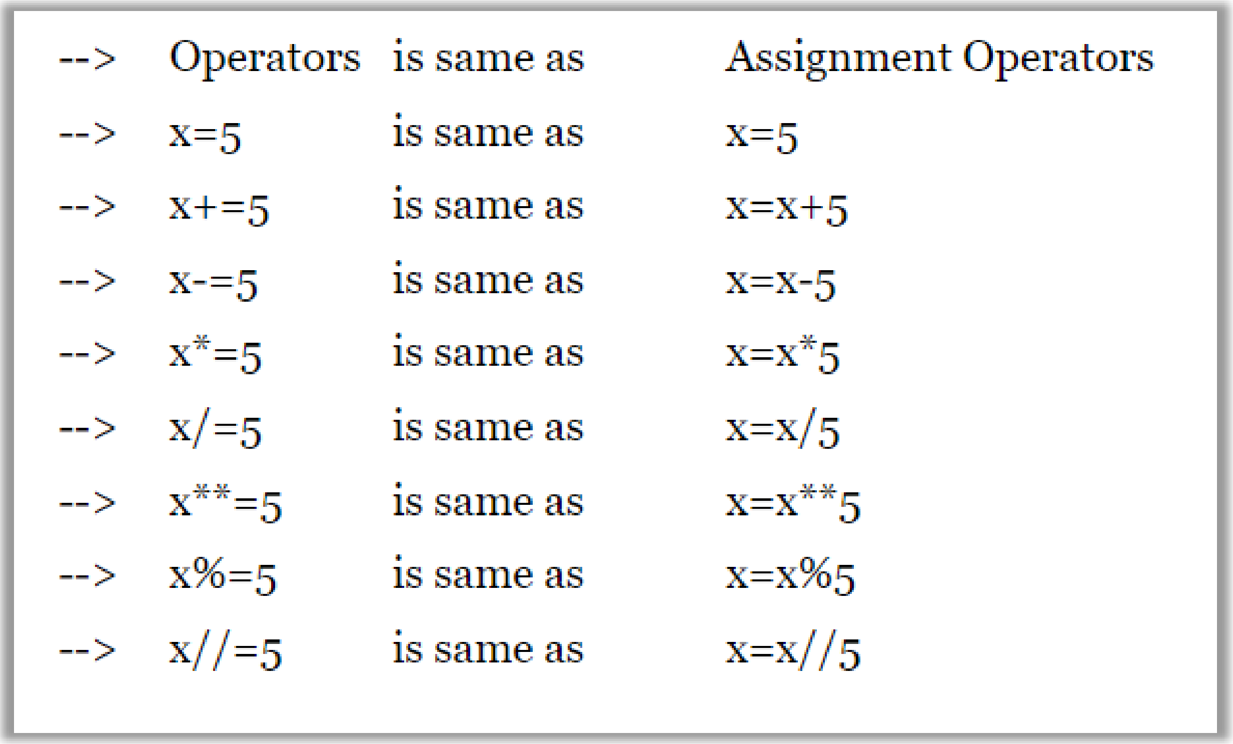
Python Operator

Operators are symbols that perform operations on operands.

(+, -, *, %, **, //, /)

Operands can be variables, strings, numbers and, Boolean etc.

(a, b, x, y, Myvar, Mystring, Mybool, c, d, p, q, r).



-->	Operators	is same as	Assignment Operators
-->	x=5	is same as	x=5
-->	x+=5	is same as	x=x+5
-->	x-=5	is same as	x=x-5
-->	x*=5	is same as	x=x*5
-->	x/=5	is same as	x=x/5
-->	x**=5	is same as	x=x**5
-->	x%=5	is same as	x=x%5
-->	x//=5	is same as	x=x//5

Python Identity Operator

Identity operator are used to compare two values to determine if they point to same object.

In Python we are having two identity operators:

- is
- is not

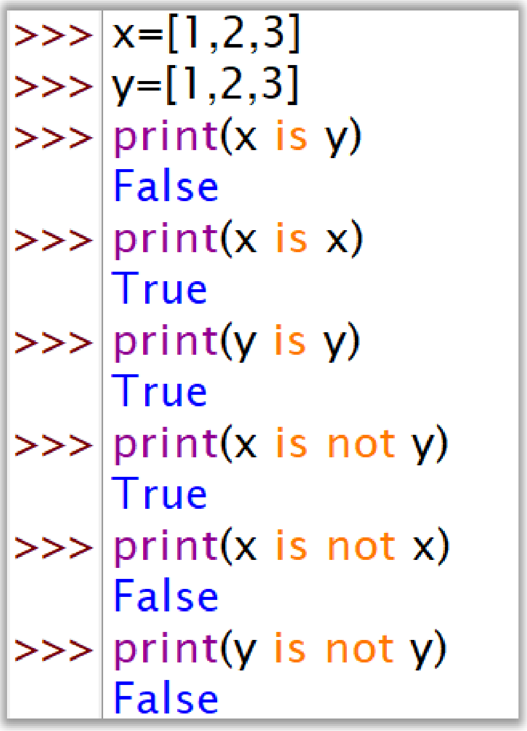
is operator:

The **is operator** returns True if both operands point to the same object.

Is not operator:

The **is not operator** returns True if both operands do not point the same object.

e.g.



```
>>> x=[1,2,3]
>>> y=[1,2,3]
>>> print(x is y)
    False
>>> print(x is x)
    True
>>> print(y is y)
    True
>>> print(x is not y)
    True
>>> print(x is not x)
    False
>>> print(y is not y)
    False
```

Python Membership Operator

Membership Operators are used to check if a sequence is present in an object like string, lists etc.

In python we are having two Membership Operators:

- in
- not in

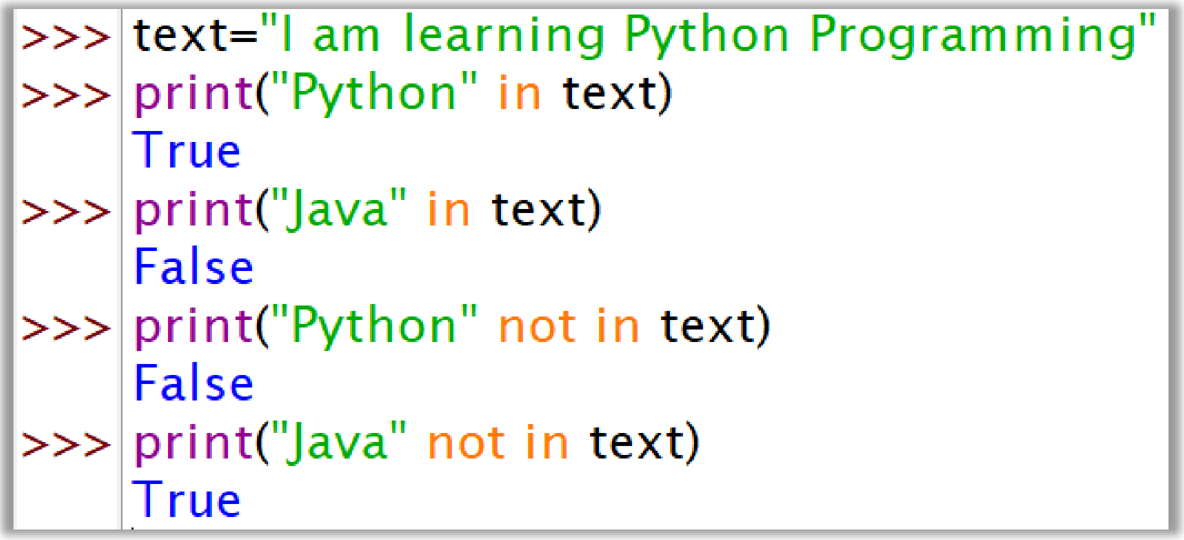
in operator:

The in operator returns True if a sequence is present in an Object.

not in operator:

The not in operator returns True if a sequence or Value is not present in an object.

e.g.



```
>>> text="I am learning Python Programming"
>>> print("Python" in text)
True
>>> print("Java" in text)
False
>>> print("Python" not in text)
False
>>> print("Java" not in text)
True
```

Python Number Methods

Rounding a Number:

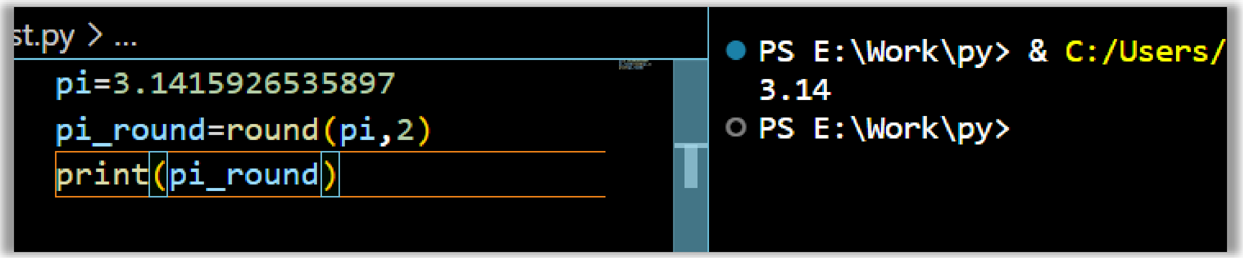
The round() function rounds a floating point number to a specified decimal point.

Syntax: round(number,ndigits)

Number- the number to be rounded off.

Ndigits- the number of decimal places.

e.g.



```
st.py > ...
  pi=3.1415926535897
  pi_round=round(pi,2)
  print(pi_round)
```

```
PS E:\Work\py> & C:/Users/
3.14
PS E:\Work\py>
```

Raising a number to a power:

The pow() function is used to raise a number to a specified power.

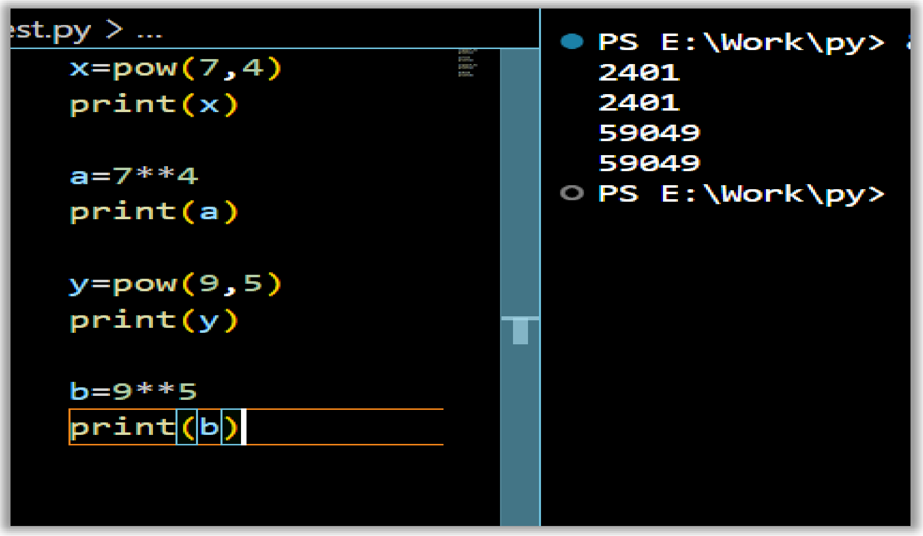
Syntax:

pow(base,exp)

Base- the base number

Exp- the exponent.

e.g.



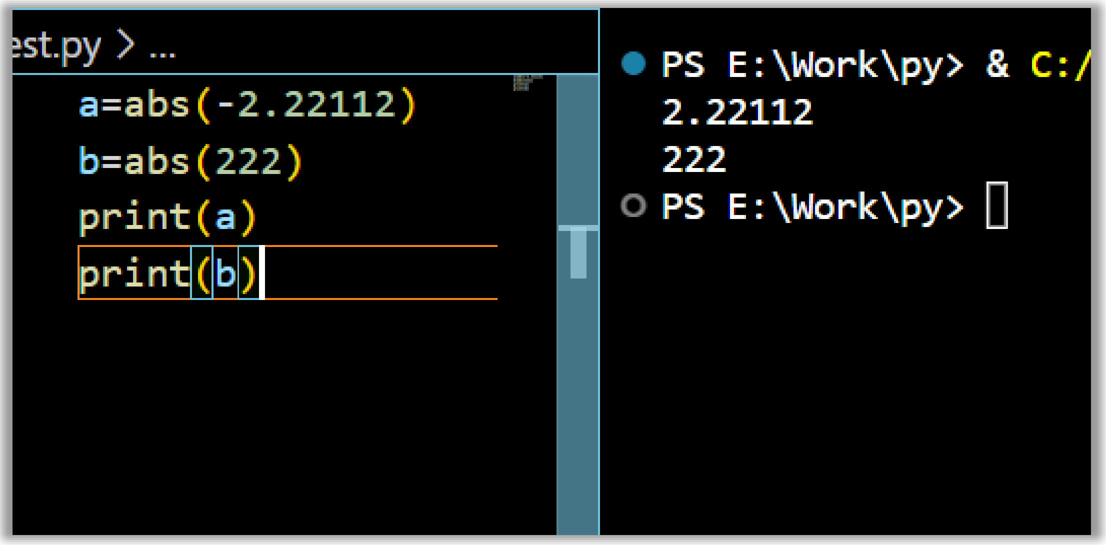
```
est.py > ...
x=pow(7,4)
print(x)

a=7**4
print(a)

y=pow(9,5)
print(y)

b=9**5
print(b)
```

```
PS E:\Work\py> &
2401
2401
59049
59049
PS E:\Work\py>
```

Abs Method:

The abs() function make any negative number to positive, while positive numbers are unaffected.

e.g.



```
est.py > ...
a=abs(-2.22112)
b=abs(222)
print(a)
print(b)
```

```
PS E:\Work\py> & C:/
2.22112
222
PS E:\Work\py>
```

Divmode Method:

The divmode() method in Python takes two numbers and returns a pair of numbers consisting of their quotient and remainder.

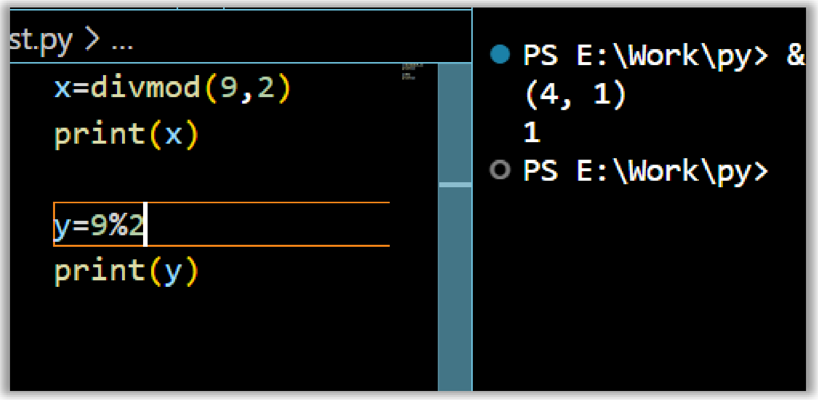
Syntax:

divmode(x,y)

where x is numerators and y id denominator.

Both x and y must not be a complex number.

e.g.



```
st.py > ...
   x=divmod(9,2)
   print(x)

   y=9%2
   print(y)
```

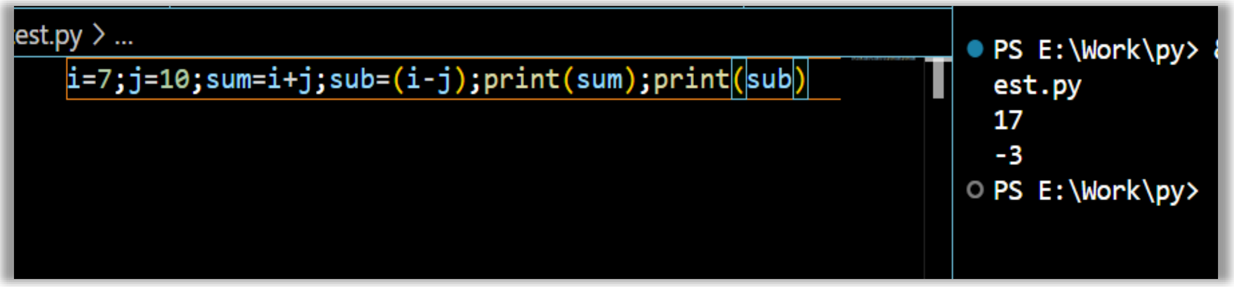
```
● PS E:\Work\py> &
  (4, 1)
  1
○ PS E:\Work\py>
```

Types of Executing Statements

There are two ways to execute statements in Python:

- Executing Multiple statements in one line.
- Executing One statement in multiple line.

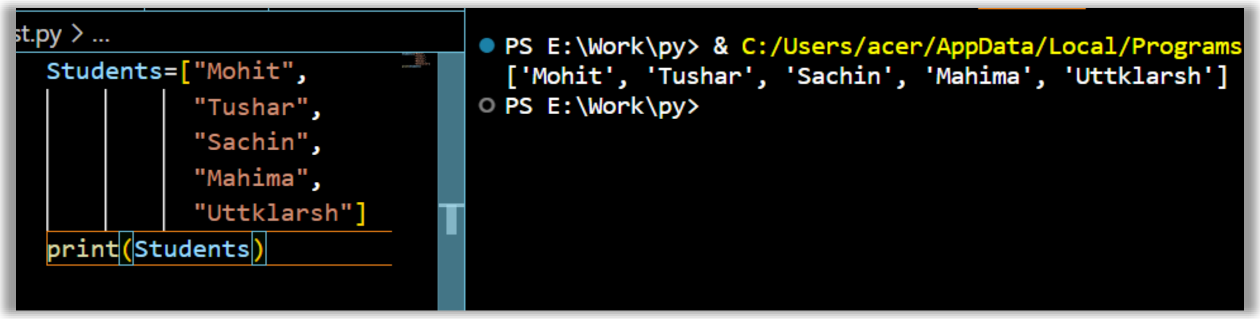
Executing Multiple statement in one line:



```
est.py > ...
i=7;j=10;sum=i+j;sub=(i-j);print(sum);print(sub)
```

```
PS E:\Work\py> &
est.py
17
-3
PS E:\Work\py>
```

Executing one statement in multiple lines:



```
st.py > ...
Students=["Mohit",
          "Tushar",
          "Sachin",
          "Mahima",
          "Uttklarsh"]
print(Students)
```

```
PS E:\Work\py> & C:/Users/acer/AppData/Local/Programs
['Mohit', 'Tushar', 'Sachin', 'Mahima', 'Uttklarsh']
PS E:\Work\py>
```

Containers

Python Containers:

Containers are objects that contain objects.

What is an Object?

In Python, everything is an object.

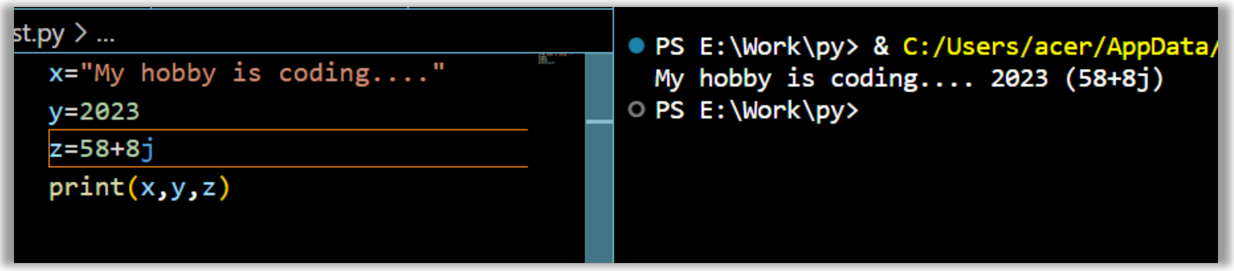
Even the simplest strings and numbers are considered as an object.

Containers:

Containers are also called as collection of objects that contain objects.

e.g.

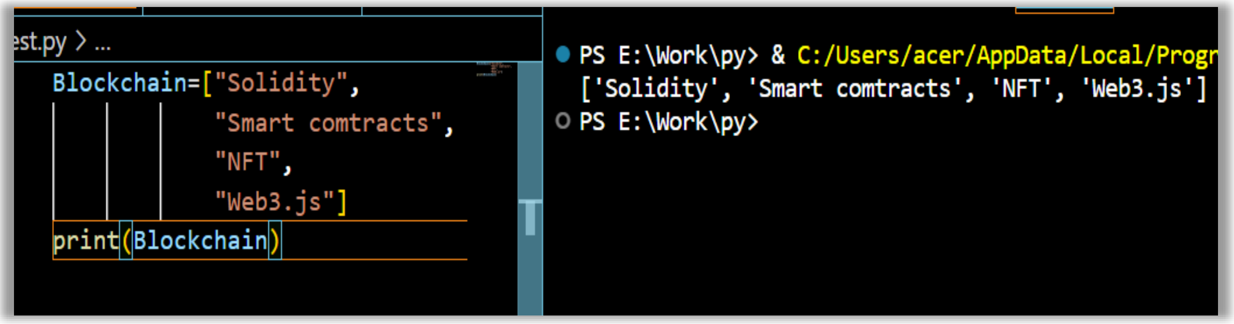
The code without using a list container.



```
st.py > ...
    x="My hobby is coding...."
    y=2023
    z=58+8j
    print(x,y,z)
```

```
PS E:\Work\py> & C:/Users/acer/AppData/
My hobby is coding.... 2023 (58+8j)
PS E:\Work\py>
```

The code with using a list container.



```
est.py > ...
    Blockchain=["Solidity",
                "Smart comtracts",
                "NFT",
                "Web3.js"]
    print(Blockchain)
```

```
PS E:\Work\py> & C:/Users/acer/AppData/Local/Progr
['Solidity', 'Smart comtracts', 'NFT', 'Web3.js']
PS E:\Work\py>
```

Data Types

Data types represents the type of value storing in a variable.

Types of Data types:

- Mutable Data Type.
- Immutable Data Type.

Mutable definition:

Mutable is when something is changeable or has the ability to change.

In Python, mutable is the ability of object to chage their values.

Immutable definition:

Immutable is when no change is possible over time.

In python, if the value of an object cannot br changed over time, then it is known as immutable.

List of Mutable data types:

1. List
2. Sets
3. Dictionaries
4. All user-defined classes.

List of Immutable data types:

1. Numbers(Integer, Float, Decimal, Boolean)
2. Strings
3. Tuples
4. Frozen sets
5. All user-defined classes.

Type Casting

It is a method used to change the variable/ values declared in a certain type into a different data type to match the operation required to be performed by the code snippet.

Types of Type casting:

1. int()
2. float()
3. str()
4. bool()
5. complex

Lists

Python list is an ordered container.

A list is created by using Square Brackets([]).

The object are placed inside those square brackets are separated by comma(,).

A list may contain mix data types.

Adding items from a list:

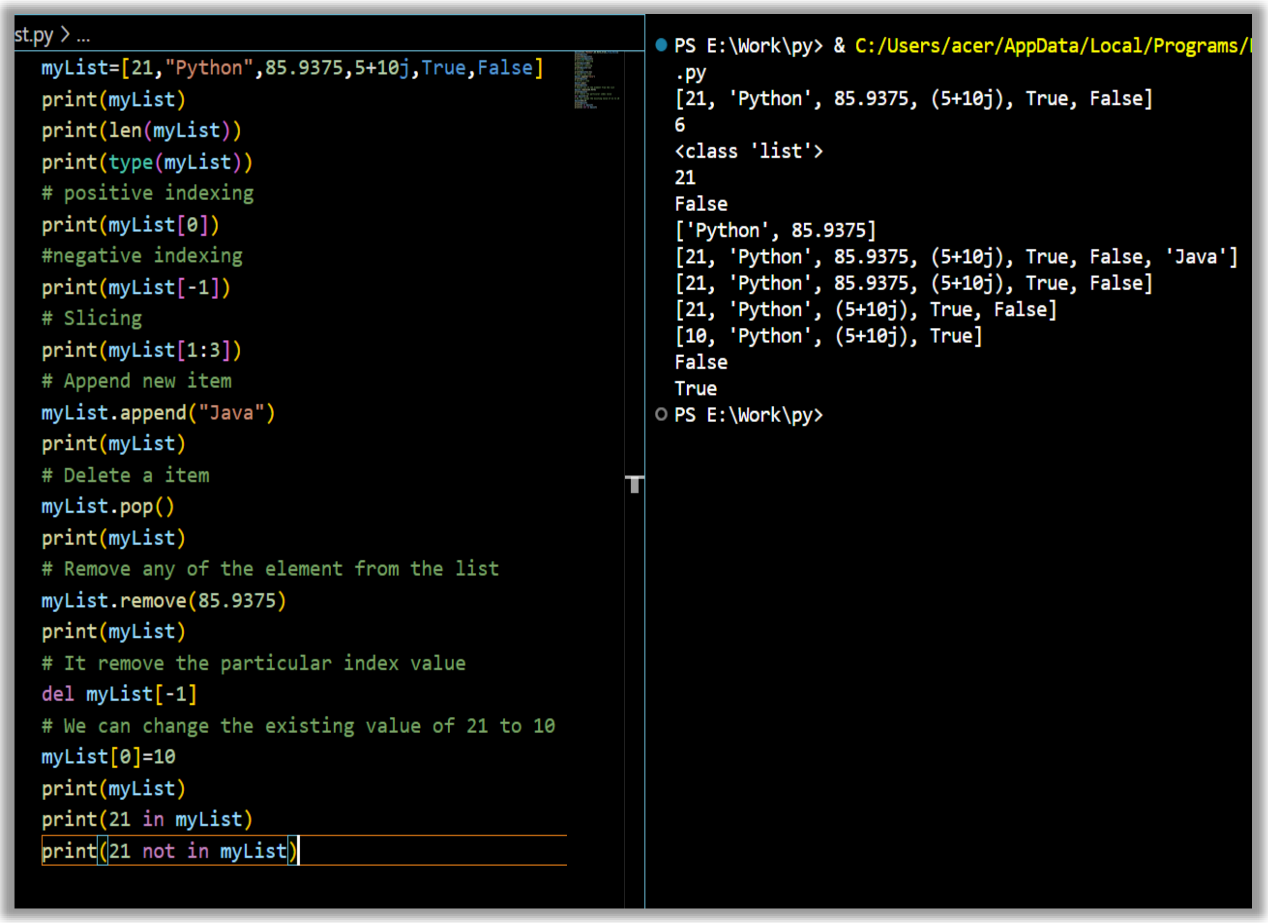
The append() method adds an item to the end of the list.

Deleting items from a list:

The pop() method removes the last item from a list.

Whereas the remove() method removes the specified item value.

e.g.



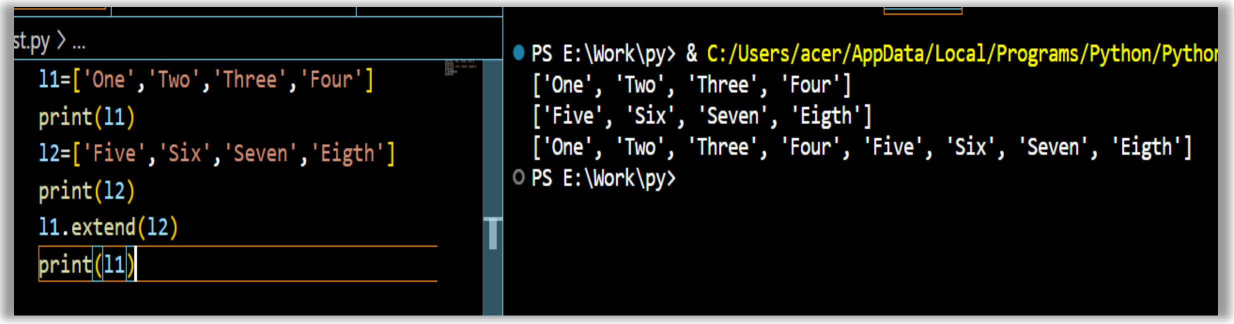
```
st.py > ...
myList=[21,"Python",85.9375,5+10j,True,False]
print(myList)
print(len(myList))
print(type(myList))
# positive indexing
print(myList[0])
#negative indexing
print(myList[-1])
# Slicing
print(myList[1:3])
# Append new item
myList.append("Java")
print(myList)
# Delete a item
myList.pop()
print(myList)
# Remove any of the element from the list
myList.remove(85.9375)
print(myList)
# It remove the particular index value
del myList[-1]
# We can change the existing value of 21 to 10
myList[0]=10
print(myList)
print(21 in myList)
print(21 not in myList)
```

```
PS E:\Work\py> & C:/Users/acer/AppData/Local/Programs/
.py
[21, 'Python', 85.9375, (5+10j), True, False]
6
<class 'list'>
21
False
['Python', 85.9375]
[21, 'Python', 85.9375, (5+10j), True, False, 'Java']
[21, 'Python', 85.9375, (5+10j), True, False]
[21, 'Python', (5+10j), True, False]
[10, 'Python', (5+10j), True]
False
True
PS E:\Work\py>
```

Extending a list:

The extents() method adds all items from a list to another list.

e.g.



```
st.py > ...
    l1=['One','Two','Three','Four']
    print(l1)
    l2=['Five','Six','Seven','Eigth']
    print(l2)
    l1.extend(l2)
    print(l1)
```

```
PS E:\Work\py> & C:/Users/acer/AppData/Local/Programs/Python/Pytho
['One', 'Two', 'Three', 'Four']
['Five', 'Six', 'Seven', 'Eigth']
['One', 'Two', 'Three', 'Four', 'Five', 'Six', 'Seven', 'Eigth']
PS E:\Work\py>
```

Tuple

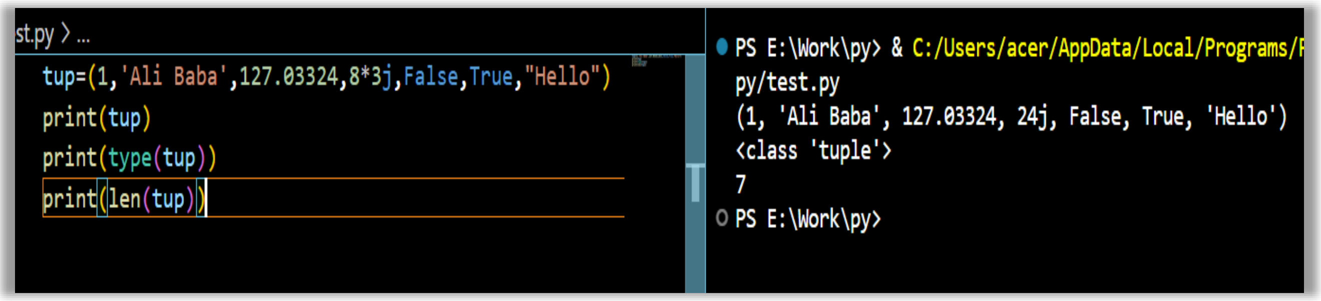
Tuple is an ordered container.

It's the same as list but the items of tuples cannot be changed.

Creating a Tuple:

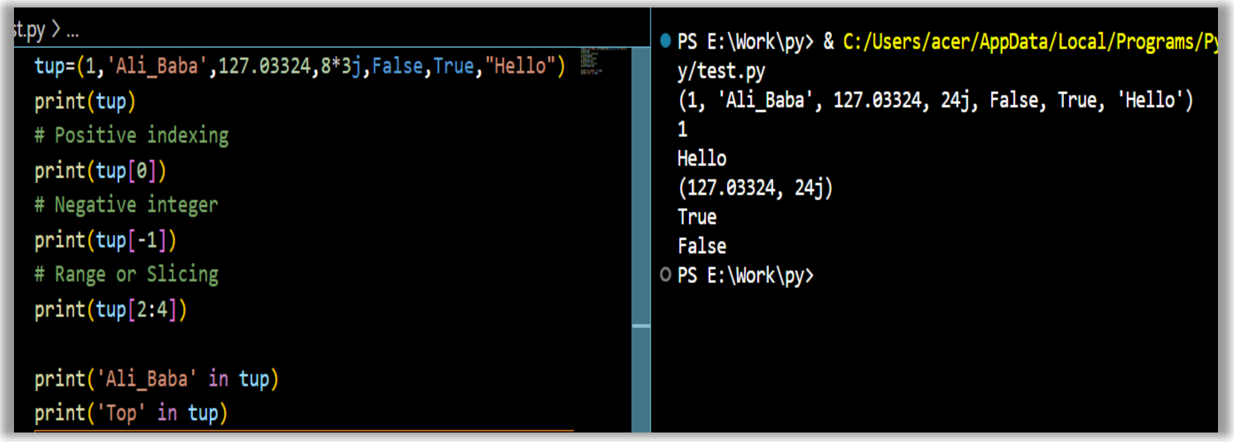
A tuple is created by using round brackets () by declaring variable tuple1, tuple2, etc. and start assigning values using equal to operator (=) to the declared variable within round brackets () and separated by comma(,).

The object may contain mixed data types.



```
st.py > ...
    tup=(1,'Ali Baba',127.03324,8*3j,False,True,"Hello")
    print(tup)
    print(type(tup))
    print(len(tup))
```

```
PS E:\Work\py> & C:/Users/acer/AppData/Local/Programs/P
py/test.py
(1, 'Ali Baba', 127.03324, 24j, False, True, 'Hello')
<class 'tuple'>
7
PS E:\Work\py>
```



```python
tup=(1,'Ali_Baba',127.03324,8*3j,False,True,"Hello")
print(tup)
# Positive indexing
print(tup[0])
# Negative integer
print(tup[-1])
# Range or Slicing
print(tup[2:4])

print('Ali_Baba' in tup)
print('Top' in tup)
```

```
PS E:\Work\py> & C:/Users/acer/AppData/Local/Programs/Py
y/test.py
(1, 'Ali_Baba', 127.03324, 24j, False, True, 'Hello')
1
Hello
(127.03324, 24j)
True
False
PS E:\Work\py>
```

Another way to create a tuple:

Another way to creating a tuple is use the tuple() constructor.

Set

A set is a container(collection of objects) that is unordered and also mutable.

Creating a set:

A set created by using curly brackets {} by declaring variable set1, set2, etc. and start assigning values using equal to operator(=) to the declared variable within curly brackets {} and separated by comma(,).

A set contain mixed data type.

Another way to creating a set:

Another way to create a set is to use the set() constructor.

Accessing Items:

Unlike lists and tuples, you cannot access the items of a set by using indexes.

It is because a set is unordered and not indexed.

However, we can use the for loop to access all the items one-by-one.

Adding items to a Set:

To add items to a set, we need to use add() method.

The add() method adds one item to a set.

Changing an Item of a Set:

The items of a set can be changed by using update() method.

Removing an Item of a Set:

To remove an item from a set, we need to use the remove() method.

You must specify the value of the item that you want to remove.

Even in Python for removing an Item in a set we can use discard() method also.

The difference between the remove() and discard() method is that the discard() method does not raise an error if the specified item is not present.

pop() method:

We can remove the last items from a set using pop() method, but we cannot determine which item will be removed because a set is unordered.

We can get the length of a set using len().

Checking if an item exists:

To check if an item exists in a set, we need to use 'in' operator. It returns True if the item is found, otherwise returns False.

If you want to check whether the item is not in the set then we use 'not in'.

It returns True, it the item is not found, otherwise False.

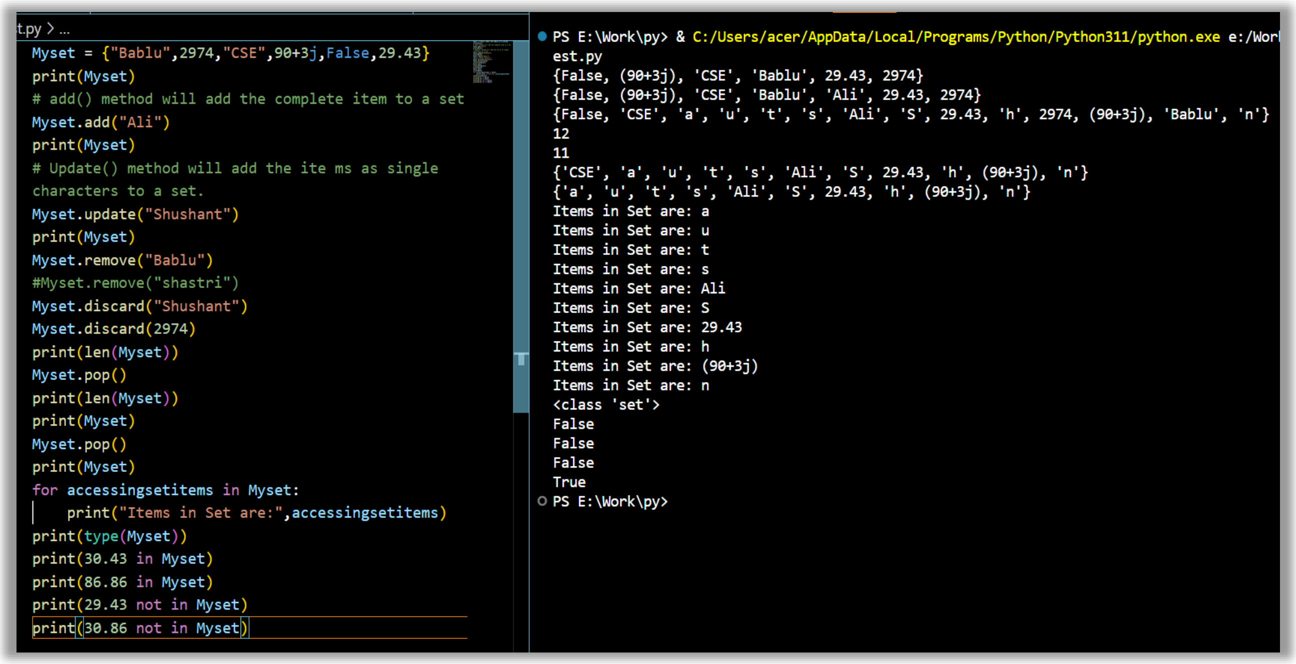
Combining two Sets:

To combine two sets, We can use the update() method. This method excludes or removes the duplicate items.

Difference of two sets:

To get the difference between two sets, we can use the subtraction operator(-).

e.g.



```
t.py > ...
Myset = {"Bablu",2974,"CSE",90+3j,False,29.43}
print(Myset)
# add() method will add the complete item to a set
Myset.add("Ali")
print(Myset)
# Update() method will add the ite ms as single
characters to a set.
Myset.update("Shushant")
print(Myset)
Myset.remove("Bablu")
#Myset.remove("shastri")
Myset.discard("Shushant")
Myset.discard(2974)
print(len(Myset))
Myset.pop()
print(len(Myset))
print(Myset)
Myset.pop()
print(Myset)
for accessingsetitems in Myset:
    print("Items in Set are:",accessingsetitems)
print(type(Myset))
print(30.43 in Myset)
print(86.86 in Myset)
print(29.43 not in Myset)
print(30.86 not in Myset)
```

```
PS E:\Work\py> & C:/Users/acer/AppData/Local/Programs/Python/Python311/python.exe e:/Wor
est.py
{False, (90+3j), 'CSE', 'Bablu', 29.43, 2974}
{False, (90+3j), 'CSE', 'Bablu', 'Ali', 29.43, 2974}
{False, 'CSE', 'a', 'u', 't', 's', 'Ali', 'S', 29.43, 'h', 2974, (90+3j), 'Bablu', 'n'}
12
11
{'CSE', 'a', 'u', 't', 's', 'Ali', 'S', 29.43, 'h', (90+3j), 'n'}
{'a', 'u', 't', 's', 'Ali', 'S', 29.43, 'h', (90+3j), 'n'}
Items in Set are: a
Items in Set are: u
Items in Set are: t
Items in Set are: s
Items in Set are: Ali
Items in Set are: S
Items in Set are: 29.43
Items in Set are: h
Items in Set are: (90+3j)
Items in Set are: n
<class 'set'>
False
False
False
True
PS E:\Work\py>
```

Dictionary

Python Dictionary:

A Dictionary is an Unordered and mutable collection of items.

A dictionary is written within curly brackets {}.

Here in dictionary, each item in a dictionary contains

key name : item value pair.

Accessing Items:

To access an item, we need to specify the "Key name" of an item inside square brackets. If you try to access an item using a key name does not exist, an error will be raised.

get() method:

We can also use get() method for accessing the items in a dictionary.

Adding Items:

To add new items, specify a new index key name inside the square brackets and assign a value using the assignment operator(=).

Changing an Item's Value:

To change an item's value, refer to its key name using square brackets and use the assignment operator(=).

Removing an Item:

To remove an item, we need to use the pop() method.

The pop() method removes an item with the specified key name.

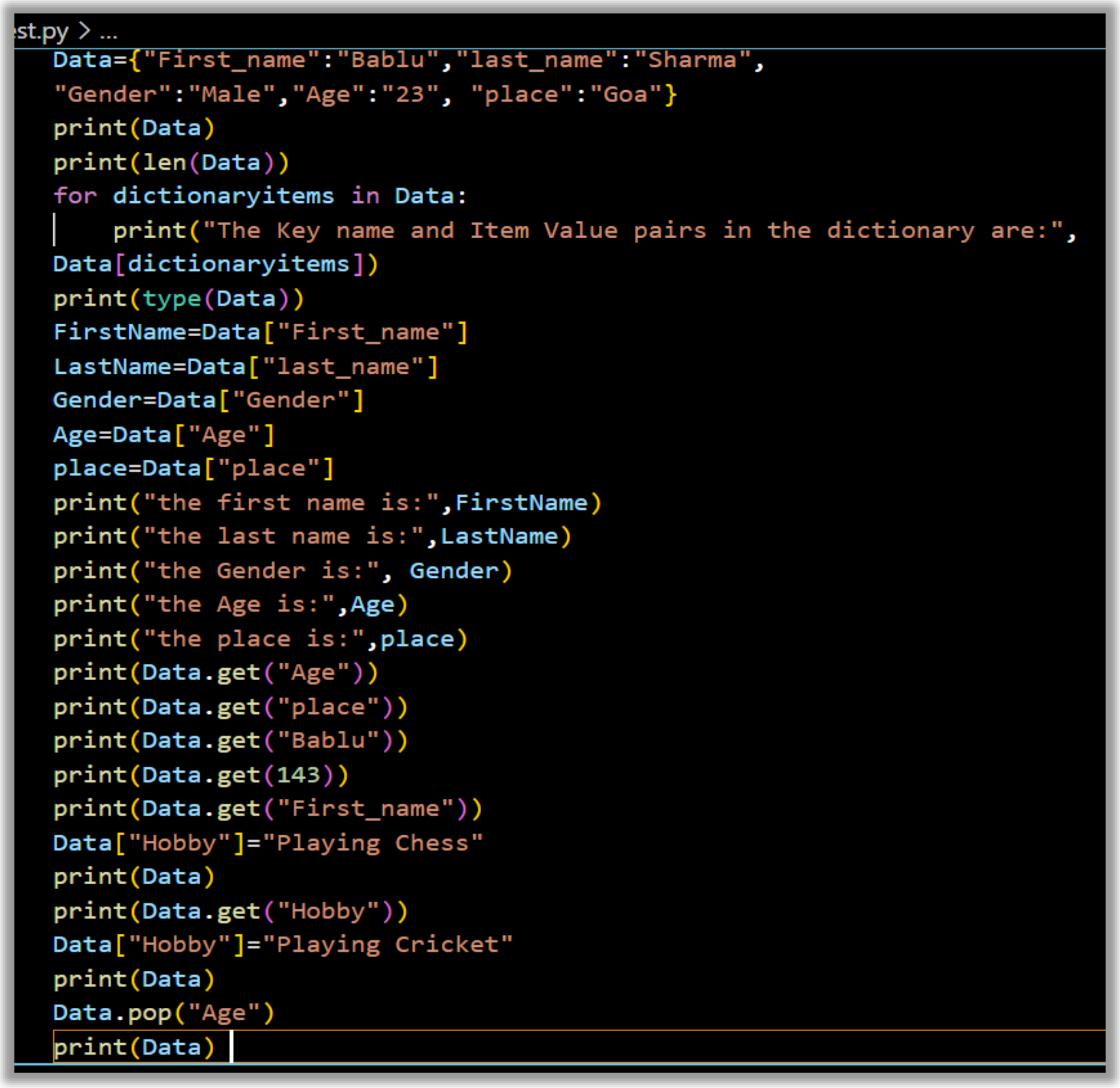
Instead of pop() method we can remove an item using del keyword also.

Getting the Length of a Dictionary:

To get the length or the number of items in a dictionary, we need to use len() method.

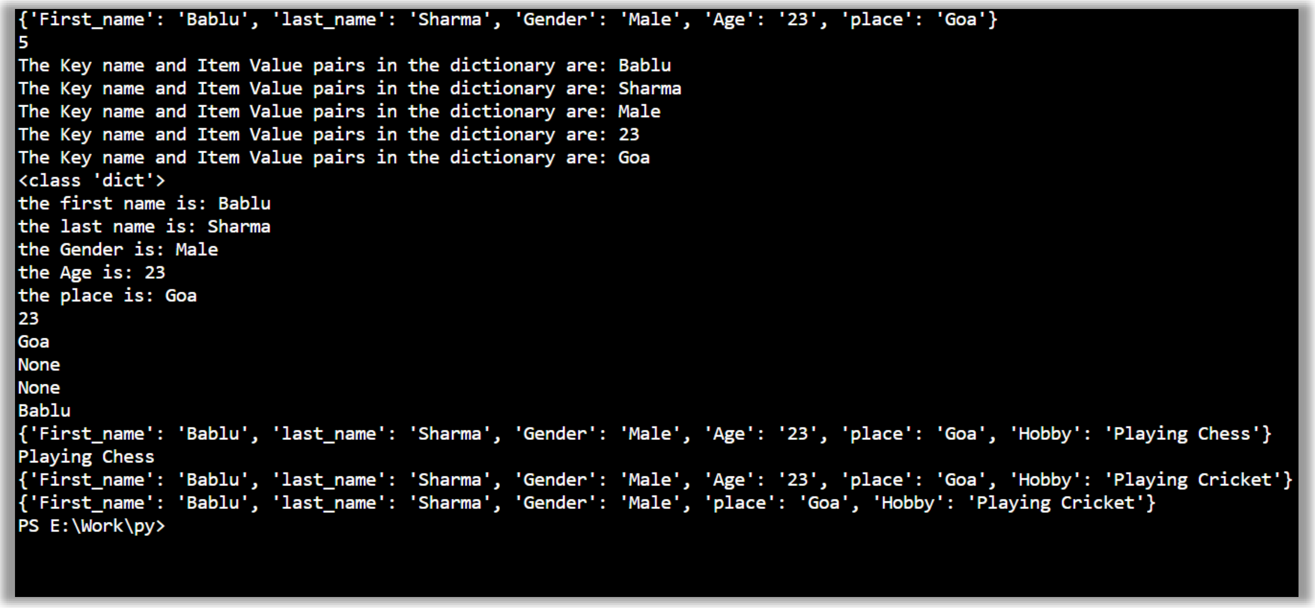
Checking if a Key Exists:

To check if a key exists in a dictionary, we need to use the membership operators like "in" and "not in" along with if statement.[Conditional statements]



```
st.py > ...
Data={"First_name":"Bablu","last_name":"Sharma",
"Gender":"Male","Age":"23", "place":"Goa"}
print(Data)
print(len(Data))
for dictionaryitems in Data:
    print("The Key name and Item Value pairs in the dictionary are:",
Data[dictionaryitems])
print(type(Data))
FirstName=Data["First_name"]
LastName=Data["last_name"]
Gender=Data["Gender"]
Age=Data["Age"]
place=Data["place"]
print("the first name is:",FirstName)
print("the last name is:",LastName)
print("the Gender is:", Gender)
print("the Age is:",Age)
print("the place is:",place)
print(Data.get("Age"))
print(Data.get("place"))
print(Data.get("Bablu"))
print(Data.get(143))
print(Data.get("First_name"))
Data["Hobby"]="Playing Chess"
print(Data)
print(Data.get("Hobby"))
Data["Hobby"]="Playing Cricket"
print(Data)
Data.pop("Age")
print(Data)
```

Output:


```
{'First_name': 'Bablu', 'last_name': 'Sharma', 'Gender': 'Male', 'Age': '23', 'place': 'Goa'}
5
The Key name and Item Value pairs in the dictionary are: Bablu
The Key name and Item Value pairs in the dictionary are: Sharma
The Key name and Item Value pairs in the dictionary are: Male
The Key name and Item Value pairs in the dictionary are: 23
The Key name and Item Value pairs in the dictionary are: Goa
<class 'dict'>
the first name is: Bablu
the last name is: Sharma
the Gender is: Male
the Age is: 23
the place is: Goa
23
Goa
None
None
Bablu
{'First_name': 'Bablu', 'last_name': 'Sharma', 'Gender': 'Male', 'Age': '23', 'place': 'Goa', 'Hobby': 'Playing Chess'}
Playing Chess
{'First_name': 'Bablu', 'last_name': 'Sharma', 'Gender': 'Male', 'Age': '23', 'place': 'Goa', 'Hobby': 'Playing Cricket'}
{'First_name': 'Bablu', 'last_name': 'Sharma', 'Gender': 'Male', 'place': 'Goa', 'Hobby': 'Playing Cricket'}
PS E:\Work\py>
```

List Comprehension

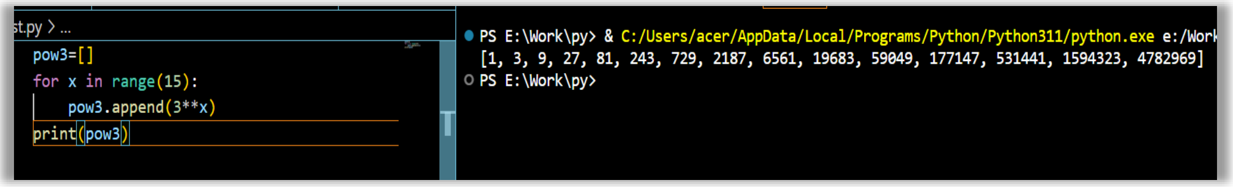
List Comprehension is a concise way of creating a new list from an existing list.

It consists of an expression and a 'for' loop statement enclosed within square brackets.

To illustrate List comprehension, we are taking an example of creating a list where each item is an increasing power of 3.

A list comprehension may have more for or if statements. An if statement can be used to filter out the elements for a new list.

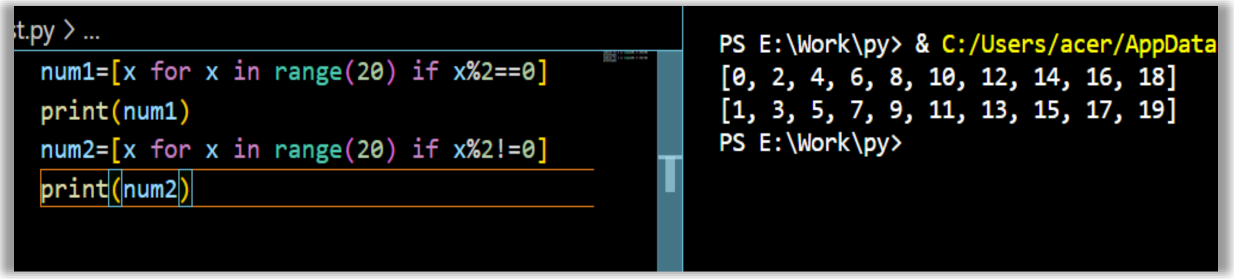
Program without using List comprehension:



```
st.py > ...
pow3=[]
for x in range(15):
    pow3.append(3**x)
print(pow3)
```

```
PS E:\Work\py> & C:/Users/acer/AppData/Local/Programs/Python/Python311/python.exe e:/Work
[1, 3, 9, 27, 81, 243, 729, 2187, 6561, 19683, 59049, 177147, 531441, 1594323, 4782969]
PS E:\Work\py>
```

Program with using List comprehension:



```
t.py > ...
num1=[x for x in range(20) if x%2==0]
print(num1)
num2=[x for x in range(20) if x%2!=0]
print(num2)
```

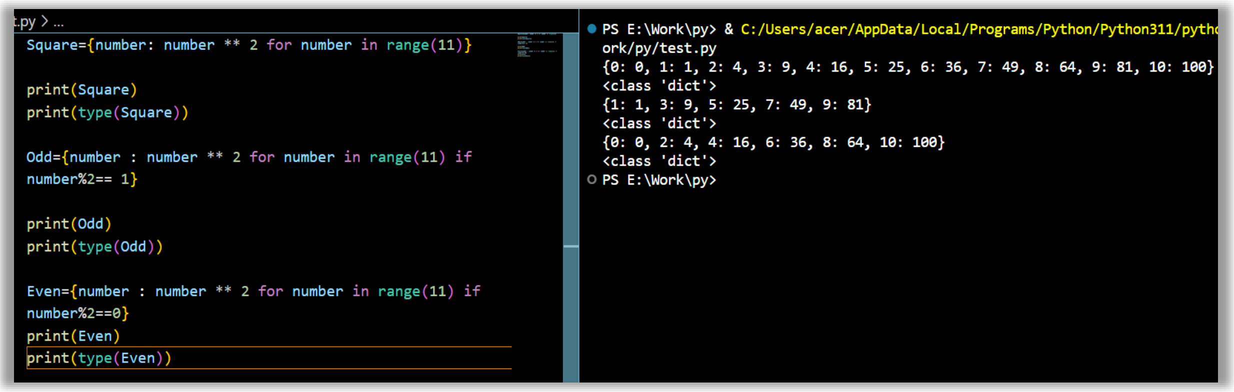
```
PS E:\Work\py> & C:/Users/acer/AppData
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
PS E:\Work\py>
```

Dictionary Comprehension

Dictionary comprehension is a concise way of creating a new dictionary from a python iterable.

It consists of a key-value pair expression and a for statement enclosed in curly brackets{}.

A dictionary comprehension may hold more than one conditional statement.



```
t.py > ...
Square={number: number ** 2 for number in range(11)}

print(Square)
print(type(Square))

Odd={number : number ** 2 for number in range(11) if
number%2== 1}

print(Odd)
print(type(Odd))

Even={number : number ** 2 for number in range(11) if
number%2==0}
print(Even)
print(type(Even))
```

```
PS E:\Work\py> & C:/Users/acer/AppData/Local/Programs/Python/Python311/pytho
ork/py/test.py
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81, 10: 100}
<class 'dict'>
{1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
<class 'dict'>
{0: 0, 2: 4, 4: 16, 6: 36, 8: 64, 10: 100}
<class 'dict'>
PS E:\Work\py>
```

Functions

A function is a group of statements that performs a particular task. This function's task is simply to print "Hello World!".

Creating a function:

To create a function, we need the following:

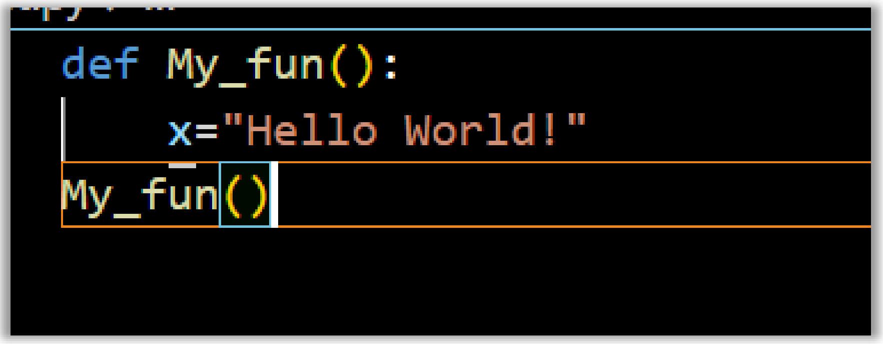
- ➢ The def Keyword.
- ➢ A function name
- ➢ Round brackets () and a colon (:)
- ➢ A function body that is a group of statements.

Calling a function:

To execute a function, it needs to be called.

To call a function, use its function name with parentheses(). My_fun()

e.g.



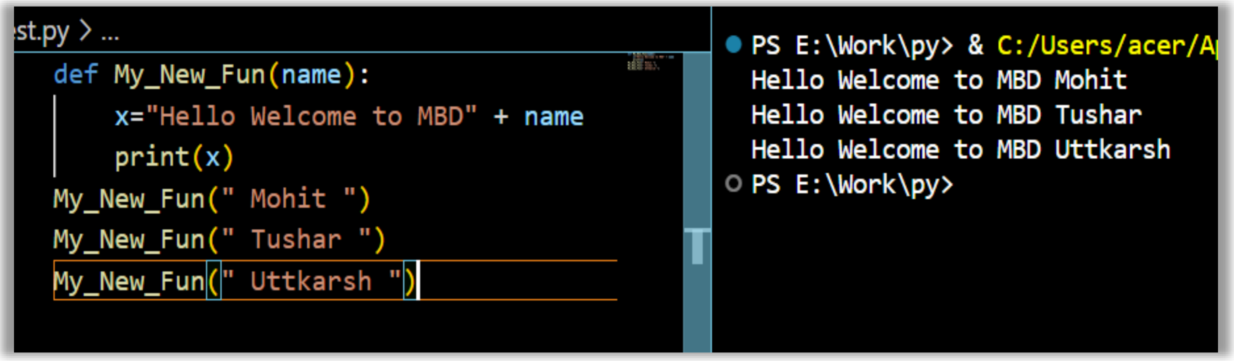
```
def My_fun():
    x="Hello World!"
My_fun()
```

Python Function Parameters/Arguments:

When calling a function, we can pass data or values using parameters/arguments.

A Parameter is a variable declared within the function.

e.g.



```
st.py > ...
def My_New_Fun(name):
    x="Hello Welcome to MBD" + name
    print(x)
My_New_Fun(" Mohit ")
My_New_Fun(" Tushar ")
My_New_Fun(" Uttkarsh ")
```

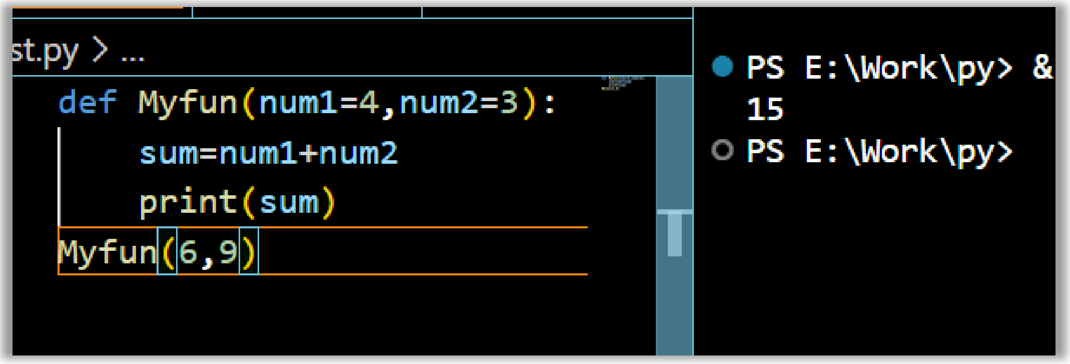
```
PS E:\Work\py> & C:/Users/acer/A
Hello Welcome to MBD Mohit
Hello Welcome to MBD Tushar
Hello Welcome to MBD Uttkarsh
PS E:\Work\py>
```

Python Function Multiple Parameters/Arguments:

To use multiple parameters, separate them using commas.

The arguments when calling the function should also be separated by commas.

e.g.



```
st.py > ...
def Myfun(num1=4,num2=3):
    sum=num1+num2
    print(sum)
Myfun(6,9)
```

```
PS E:\Work\py> &
15
PS E:\Work\py>
```

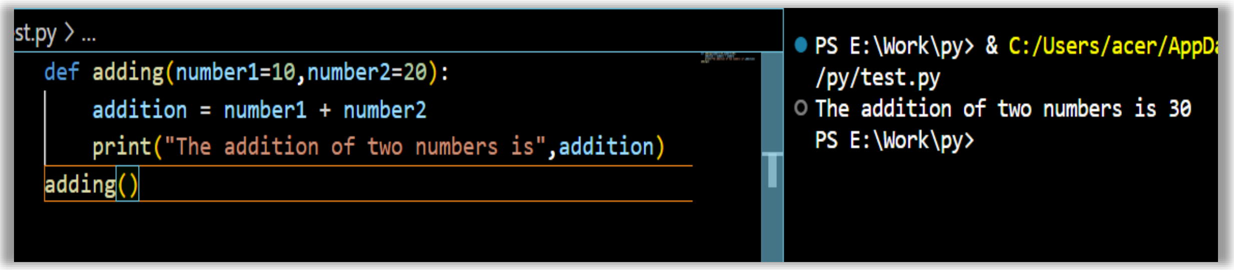
Python Function Default Parameters/Arguments:

A function can have default arguments.

It can be done using the assignment operator.(=)

If you don't pass the argument, then the default argument will be used instead.

e.g.



```
st.py > ...
def adding(number1=10,number2=20):
    addition = number1 + number2
    print("The addition of two numbers is",addition)
adding()
```

```
PS E:\Work\py> & C:/Users/acer/AppDa
/py/test.py
The addition of two numbers is 30
PS E:\Work\py>
```

Lambda Functions

Lambda Functions also called as Anonymous functions.

These lambda function does not have names.

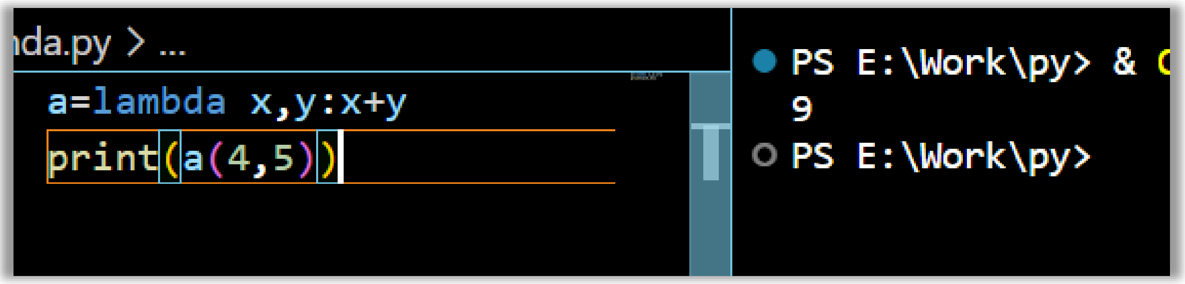
The body of lambda function can only have one single expression, but can have multiple arguments.

The result of the expression is automatically returned.

Syntax:

lambda arguments: expression

e.g.



```
ıda.py > ...
a=lambda x,y:x+y
print(a(4,5))
```

```
PS E:\Work\py> & C
9
PS E:\Work\py>
```

Loops

In Python, there are two types of loop:

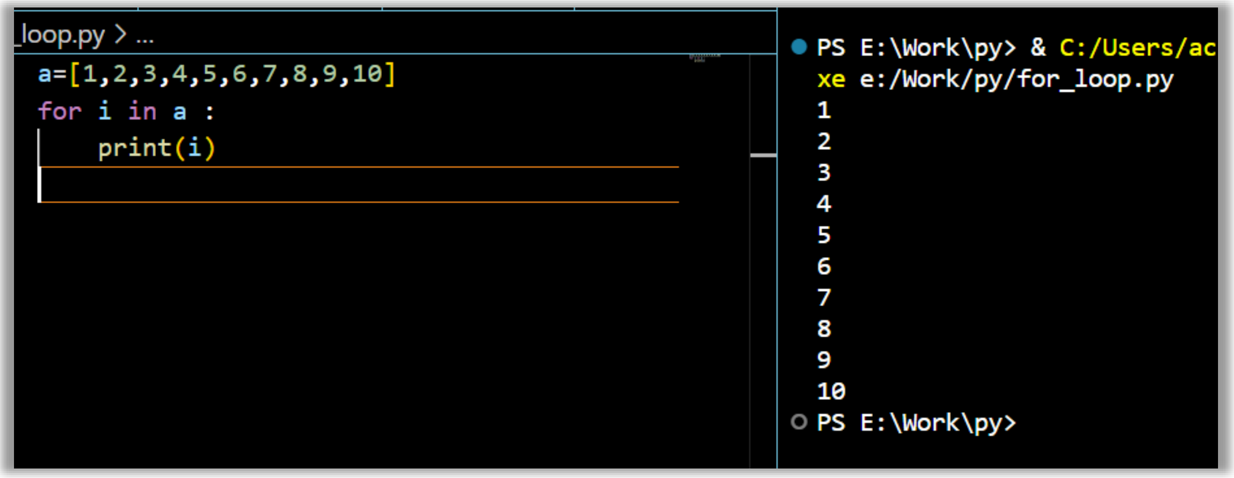
- The for loop.
- The while loop.

Python for Loop:

The for loop is used to loop through or iterate over a sequence or iterable objects.

Iterable objects are strings, lists, sets etc.

e.g.



```
_loop.py > ...
a=[1,2,3,4,5,6,7,8,9,10]
for i in a :
    print(i)
```

```
PS E:\Work\py> & C:/Users/ac
xe e:/Work/py/for_loop.py
1
2
3
4
5
6
7
8
9
10
PS E:\Work\py>
```

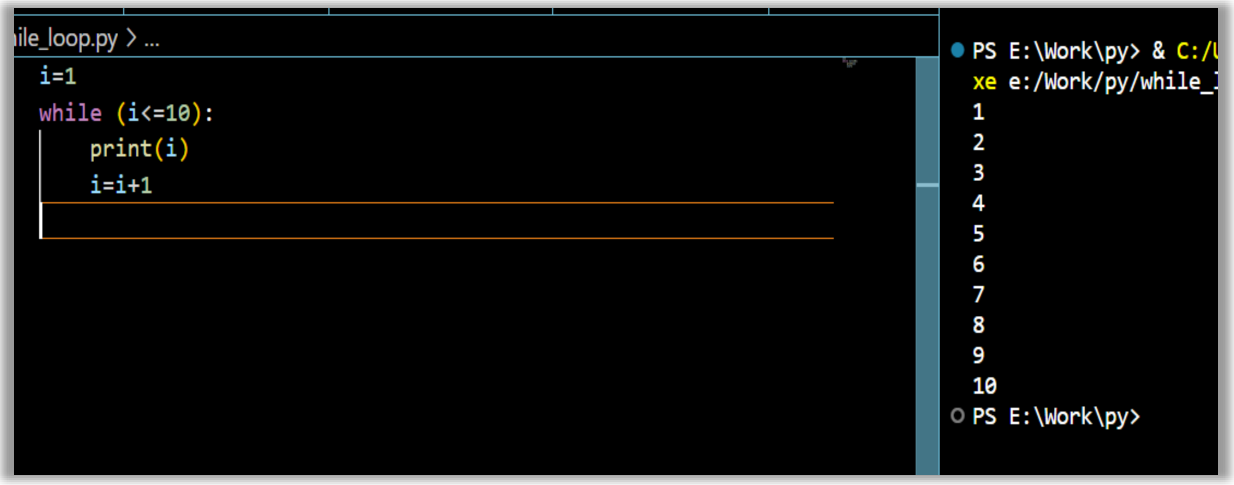
Python while loop:

The while loop executes a group of statements as long as the given expression is True.

The while Keyword is the while loop in Python.

Python While Loop is used to execute a block of statements repeatedly until a given condition is satisfied.

e.g.



```
ile_loop.py > ...
i=1
while (i<=10):
    print(i)
    i=i+1
```

```
PS E:\Work\py> & C:/
xe e:/Work/py/while_
1
2
3
4
5
6
7
8
9
10
PS E:\Work\py>
```

Break & Continue statement

The break and continue statement are commonly used in loops.

The Break statement:

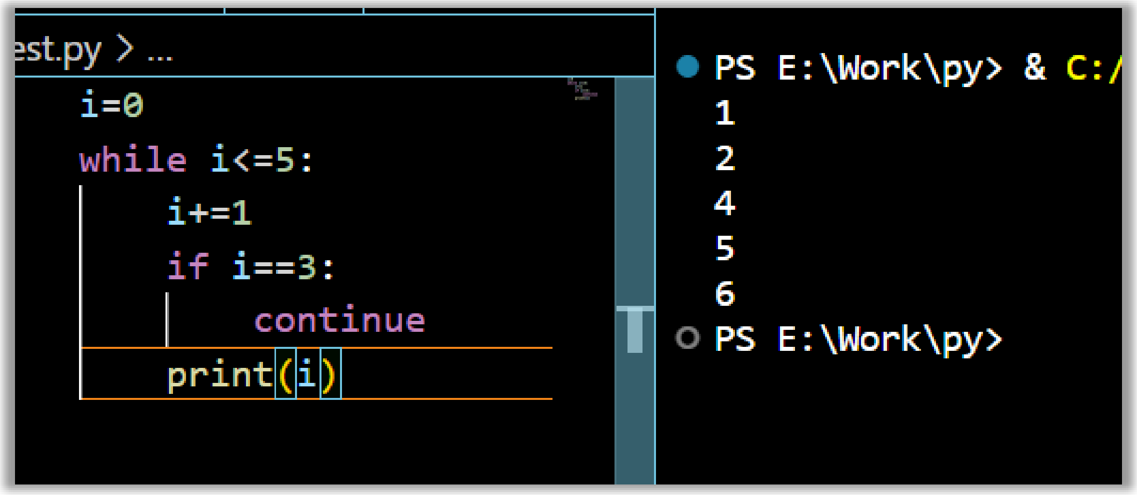
The break statement stop the execution of current loop.

The continue statement:

The continue statement terminates the execution of the current iteration of the loop.

And continue the execution of the loop with the next iteration.

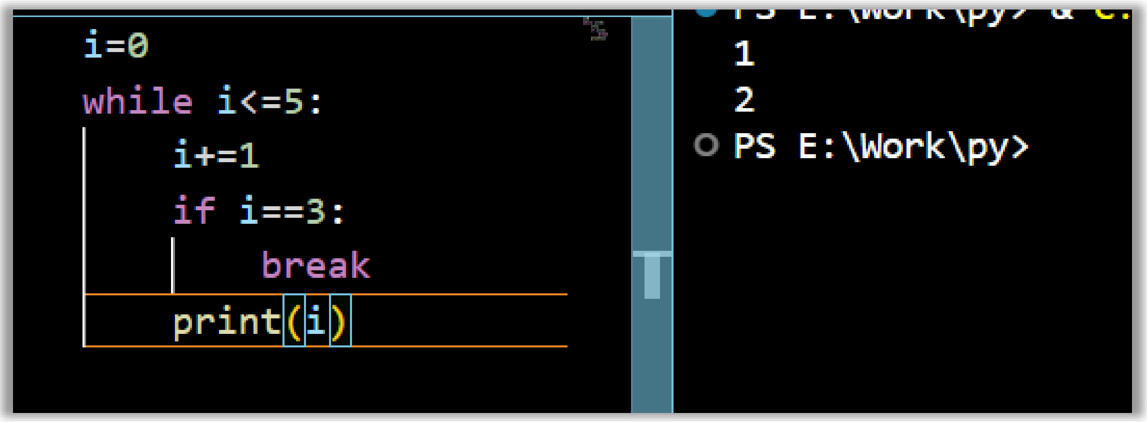
e.g. for continue statement.



```
est.py > ...
i=0
while i<=5:
    i+=1
    if i==3:
        continue
    print(i)
```

```
PS E:\Work\py> & C:/
1
2
4
5
6
PS E:\Work\py>
```

e.g. for break statement.



```
i=0
while i<=5:
    i+=1
    if i==3:
        break
    print(i)
```

```
1
2
PS E:\Work\py>
```

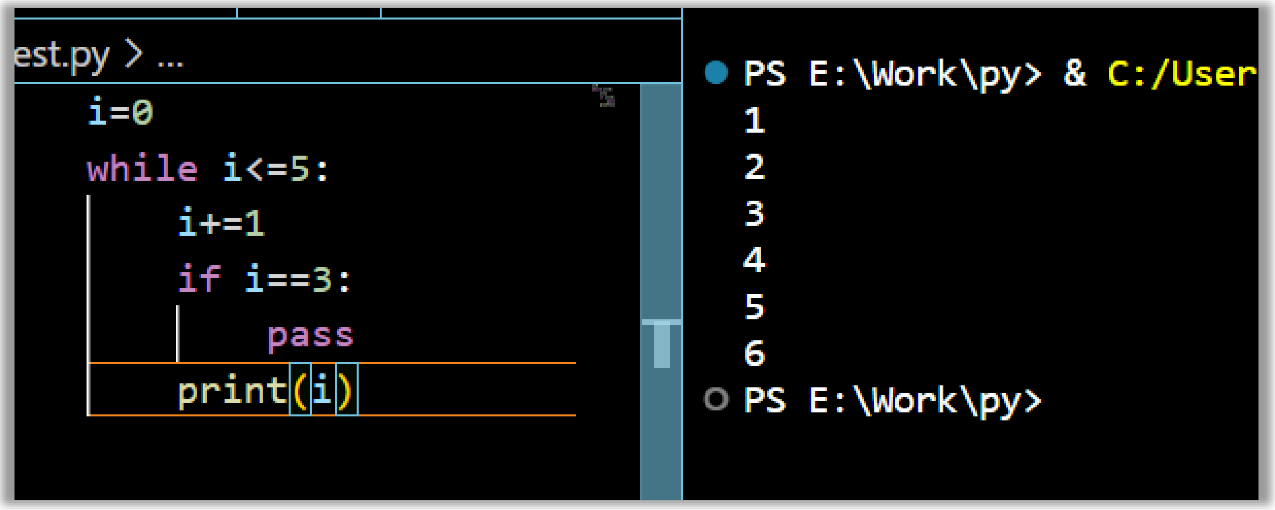
Pass statement

The pass statement is used as a placeholder when a statement is syntactically required.

when it is executed, nothing happens.

It can be used on if statements, loops, classes etc.

e.g.



```
est.py > ...
    i=0
    while i<=5:
        i+=1
        if i==3:
            pass
        print(i)
```

```
PS E:\Work\py> & C:/User
1
2
3
4
5
6
PS E:\Work\py>
```

Recursive Function

A Recursive function is defined as a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself.

Every recursive solution has two major cases, which are as follows:

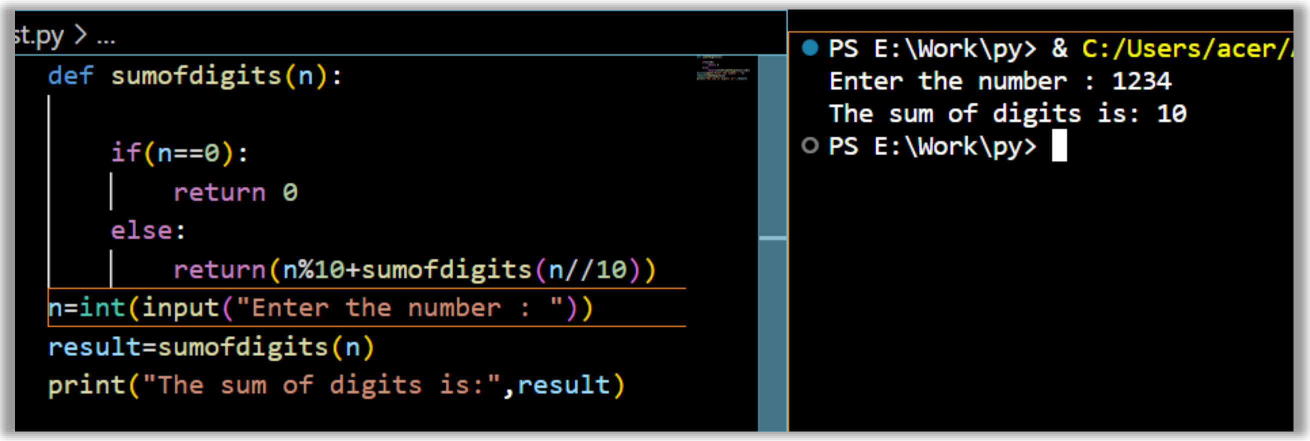
Base case:

In which the problem is simple enough to be solved directly without making any further calls to the same function.

Recursive case:

In which first problem at hand is divided into smaller sub parts.

e.g.



```
st.py > ...
def sumofdigits(n):

    if(n==0):
        return 0
    else:
        return(n%10+sumofdigits(n//10))
n=int(input("Enter the number : "))
result=sumofdigits(n)
print("The sum of digits is:",result)
```

```
PS E:\Work\py> & C:/Users/acer/
  Enter the number : 1234
  The sum of digits is: 10
PS E:\Work\py>
```

Python Keywords

Keywords are the reserved words which have predefined meaning and functionality.

Like other languages, Python also has some reserved words. These words hold some special meaning. Sometimes it may be a command, or a parameter etc.

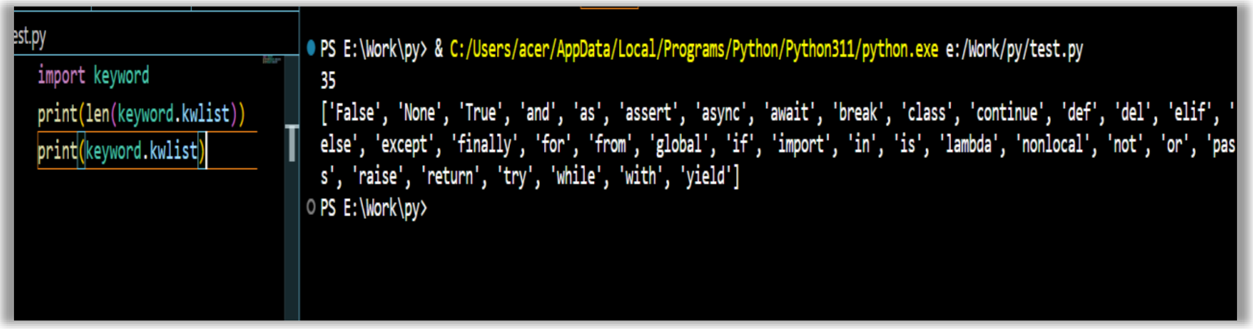
We cannot use keywords as variable names.

The Python Keywords are:

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

There are 35 keywords in python.

Program for printing Python Keywords:



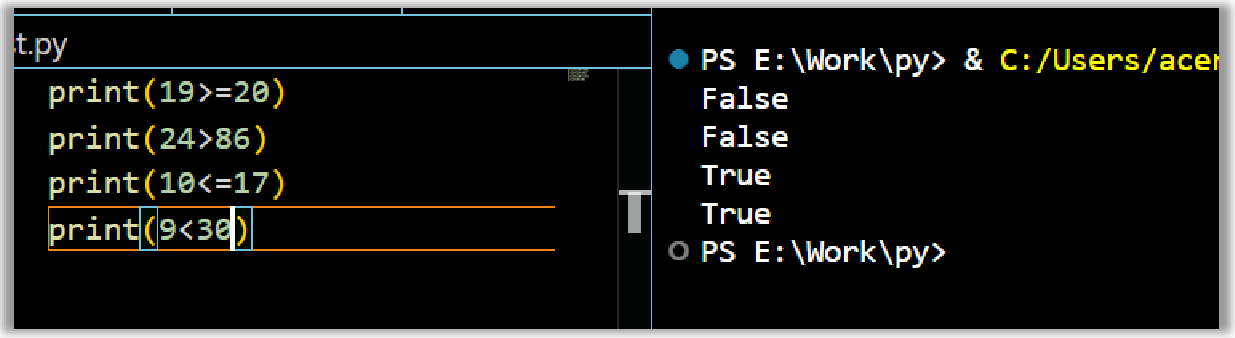
```
est.py
import keyword
print(len(keyword.kwlist))
print(keyword.kwlist)
```

```
PS E:\Work\py> & C:/Users/acer/AppData/Local/Programs/Python/Python311/python.exe e:/Work/py/test.py
35
['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break', 'class', 'continue', 'def', 'del', 'elif', '
else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pas
s', 'raise', 'return', 'try', 'while', 'with', 'yield']
PS E:\Work\py>
```

True & False Keywords

The True and False are the truth value in Python. Comparison operators returns True or False. Boolean Data Type also can hold them.

e.g.



```
t.py
print(19>=20)
print(24>86)
print(10<=17)
print(9<30)
```

```
PS E:\Work\py> & C:/Users/acer
False
False
True
True
PS E:\Work\py>
```

Random Function

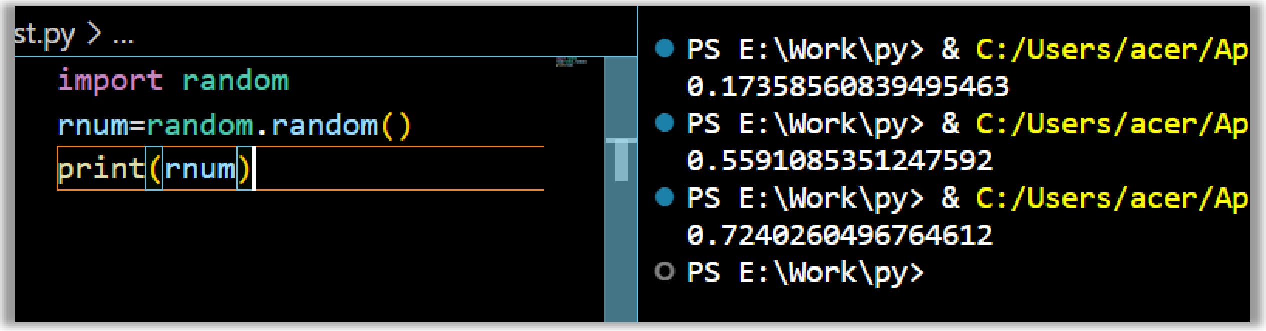
The random module lets us generate a random number:

To use the random module, we need to import it first.

Now, we can start generating a random number.

The random.random() method generates a random floating point in the range of 0.0 to 1.0.

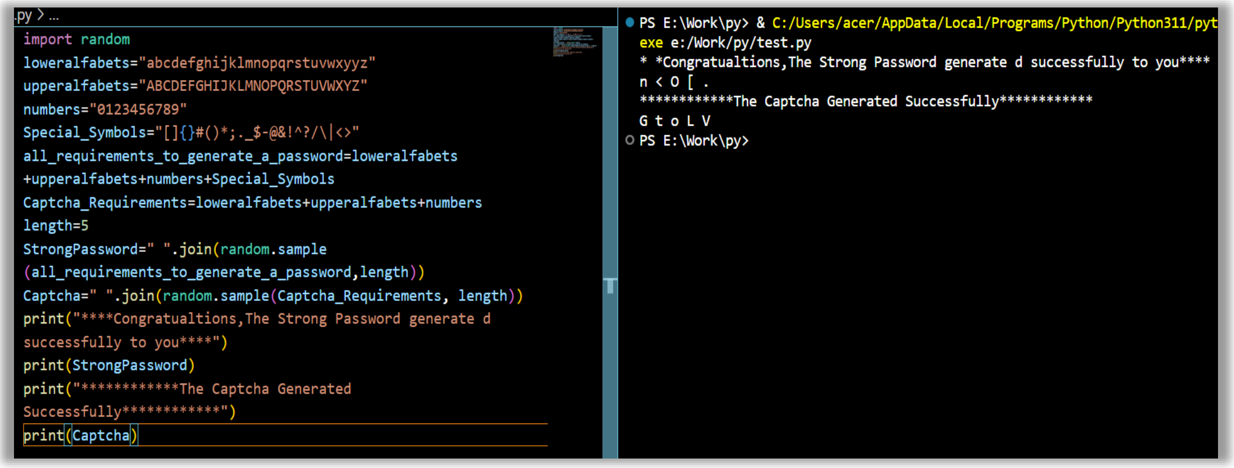
e.g.



```
st.py > ...
import random
rnum=random.random()
print(rnum)
```

```
PS E:\Work\py> & C:/Users/acer/Ap
0.17358560839495463
PS E:\Work\py> & C:/Users/acer/Ap
0.5591085351247592
PS E:\Work\py> & C:/Users/acer/Ap
0.7240260496764612
PS E:\Work\py>
```

Generating random password to the banking customers:



```
.py > ...
import random
loweralfabets="abcdefghijklmnopqrstuvwxyyz"
upperalfabets="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
numbers="0123456789"
Special_Symbols="[]{}#()*;._$-@&!^?/\|<>"
all_requirements_to_generate_a_password=loweralfabets
+upperalfabets+numbers+Special_Symbols
Captcha_Requirements=loweralfabets+upperalfabets+numbers
length=5
StrongPassword=" ".join(random.sample
(all_requirements_to_generate_a_password,length))
Captcha=" ".join(random.sample(Captcha_Requirements, length))
print("****Congratualtions,The Strong Password generate d
successfully to you****")
print(StrongPassword)
print("************The Captcha Generated
Successfully************")
print(Captcha)
```

```
PS E:\Work\py> & C:/Users/acer/AppData/Local/Programs/Python/Python311/pyt
exe e:/Work/py/test.py
* *Congratualtions,The Strong Password generate d successfully to you****
n < O [ .
************The Captcha Generated Successfully************
G t o L V
PS E:\Work\py>
```

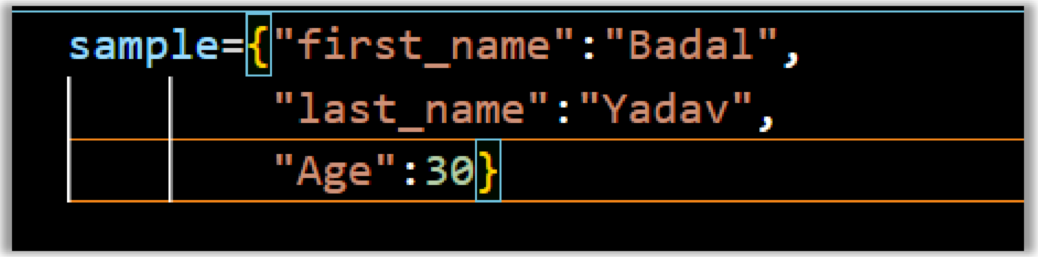
JSON

JSON stands for JavaScript Object Notation.

JSON contains data that are sent or received to and from a server.

JSON is simply a string, it follows a format similar to a Python dictionary.

e.g.



```
sample={"first_name":"Badal",
        "last_name":"Yadav",
        "Age":30}
```

Note:

The keys and string values in a JSON String should always use in double quotes and complete dictionary should be in single quotes.

JSON to dictionary:

Before we can individually access the data present in a JSON, we need to convert it to a Python dictionary first.

To do that, we need to import the json module.

import json

To convert from JSON to dictionary, use the json.loads() methods.

This method parses a JSON and returns a dictionary.

Dictionary to JSON:

To convert a dictionary to a JSON, we use the json.dumps() method.

Formatting a JSON:

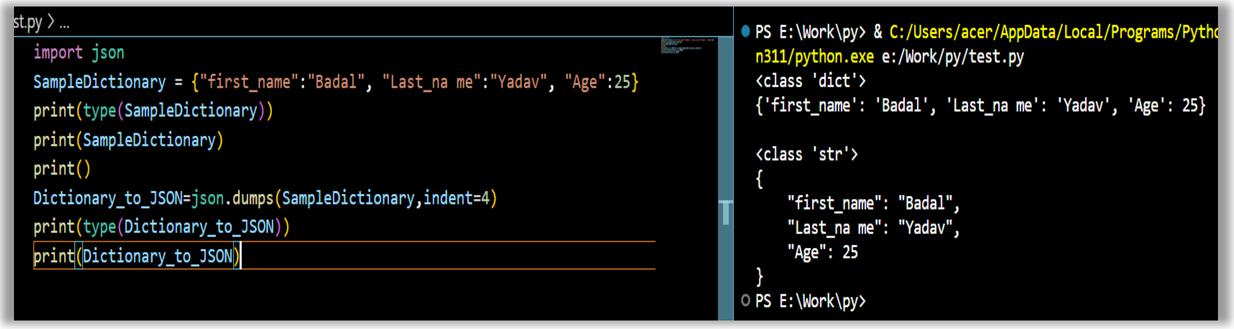
A JSON file can be formatted or be "prettified" using the indent parameter of the json.dumps() method.

Saving JSON Files:

Since JSON is simply just a string, then it can be saved just like a plain text.

When saving JSON files, make sure you can them with .json file extension ex:filename.json

e.g.



```
st.py > ...
import json
SampleDictionary = {"first_name":"Badal", "Last_na me":"Yadav", "Age":25}
print(type(SampleDictionary))
print(SampleDictionary)
print()
Dictionary_to_JSON=json.dumps(SampleDictionary,indent=4)
print(type(Dictionary_to_JSON))
print(Dictionary_to_JSON)
```

```
PS E:\Work\py> & C:/Users/acer/AppData/Local/Programs/Pytho
n311/python.exe e:/Work/py/test.py
<class 'dict'>
{'first_name': 'Badal', 'Last_na me': 'Yadav', 'Age': 25}

<class 'str'>
{
    "first_name": "Badal",
    "Last_na me": "Yadav",
    "Age": 25
}
PS E:\Work\py>
```

OOPS

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

Principals of OOPS:

- Class
- Object
- Polymorphism
- Encapsulation
- Inheritance
- Data Abstraction

Pillars of OOPS:

- Polymorphism
- Encapsulation
- Inheritance
- Data Abstraction

Class & Objects

Creating a Class:

Use the class keyword is used to define new user-define class in Python.

Syntax

```
class Classname:
    statement(s)
```

Instantiating a Class:

Now we can create an object from the class by instantiating it.

To instantiate a class, add round brackets() to its class name.

Syntax:

```
object=Classname()
```

After instantiating a class, we can now access the object's Properties.

Class Attributes:

A class can have attributes. For example, the student class can have attributes like Roll_Number, Name,and age, Batch.

Now that our class is ready, we can now instantiate it and provide values to its attributes.

The Process can also be called as "Creating an instance of the class".

An instance is simply the object created from a class.

Methods:

Methods are functions that can access the class attributes.

These methods should be defined(created) inside the class.

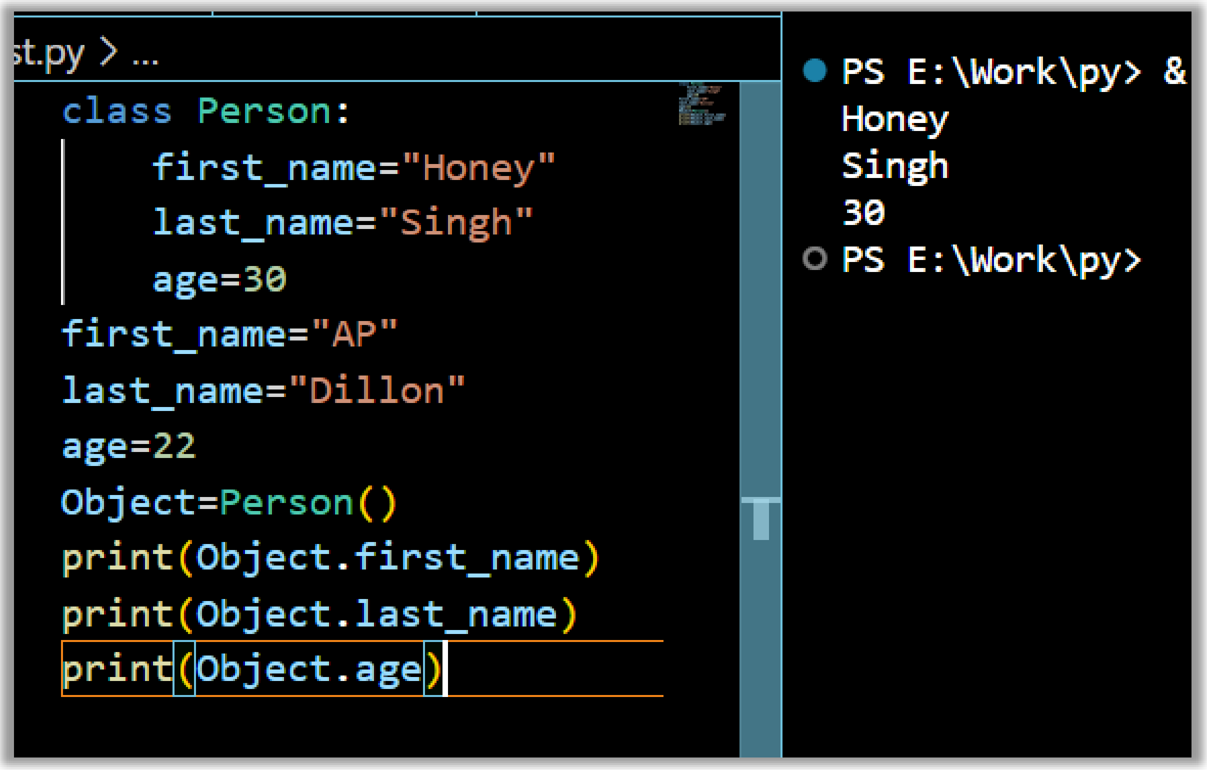
We must use self-parameter to access the class attributes.

You can also pass the arguments to a method.

__init__ Method:

The __init__ method is similar to constructors in C++ and Java. It is run as soon as an object of a class is instantiated. The method is useful to do any initialization you want to do with your object.

e.g.



```
st.py > ...
class Person:
    first_name="Honey"
    last_name="Singh"
    age=30
first_name="AP"
last_name="Dillon"
age=22
Object=Person()
print(Object.first_name)
print(Object.last_name)
print(Object.age)
```

```
PS E:\Work\py> &
Honey
Singh
30
PS E:\Work\py>
```

Object:

The object is an entity that has a state and behaviour associated with it. It may be any real-world object like a mouse, keyboard, chair, table, pen, etc. Integers, strings, floating-point numbers, even arrays, and dictionaries, are all objects.

Syntax of creating an object:

Obj=cat()

Inheritance

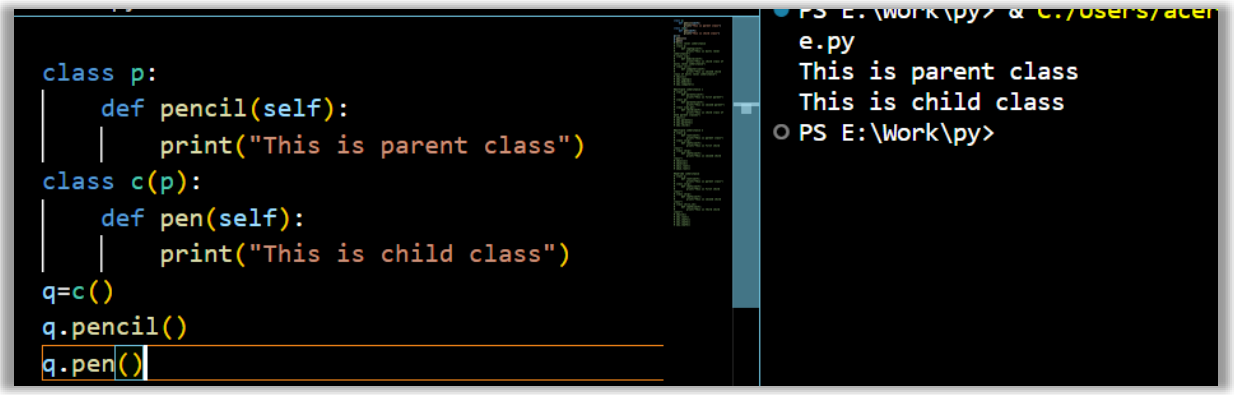
Inheritance is a feature that allows us to create a class [child class] that inherits the attributes or properties and methods of another class [Parent class].

Types of Inheritance:

1) Single Inheritance:

Single Inheritance enables a derived class to inherit properties from a single parent class, thus enabling code re-usability and the addition of new features to existing code. A--->B

e.g.



```python
class p:
    def pencil(self):
        print("This is parent class")
class c(p):
    def pen(self):
        print("This is child class")
q=c()
q.pencil()
q.pen()
```

```
PS E:\Work\py> & C:/Users/acer
e.py
This is parent class
This is child class
PS E:\Work\py>
```

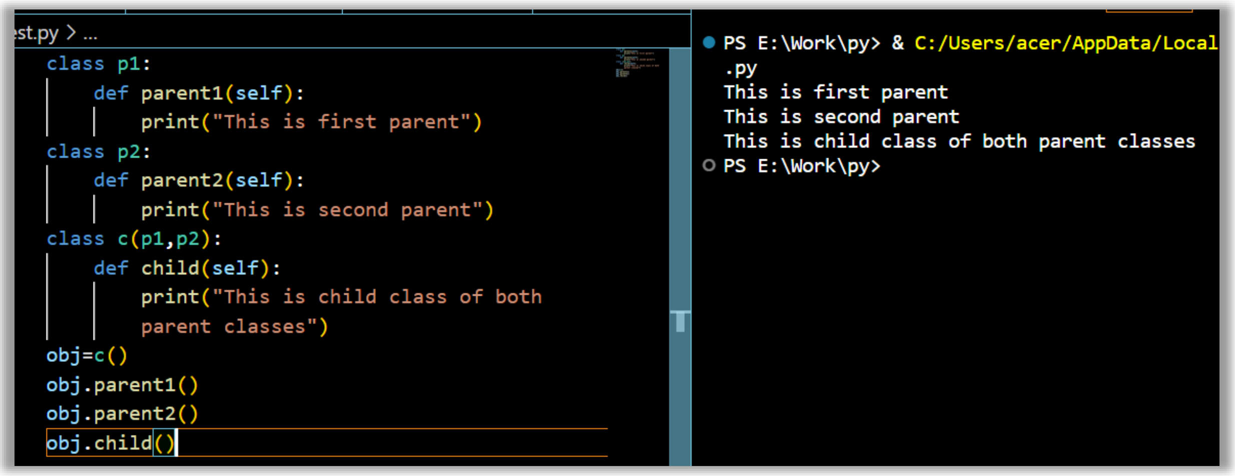
2) Multiple Inheritance:

When a class can be derived from more than one parent [Father, Mother] class this type of inheritance is called Multiple inheritance.

In Multiple inheritance, all the features of the parent class are inherited into the derived class.

A-->C and B-->C

e.g.



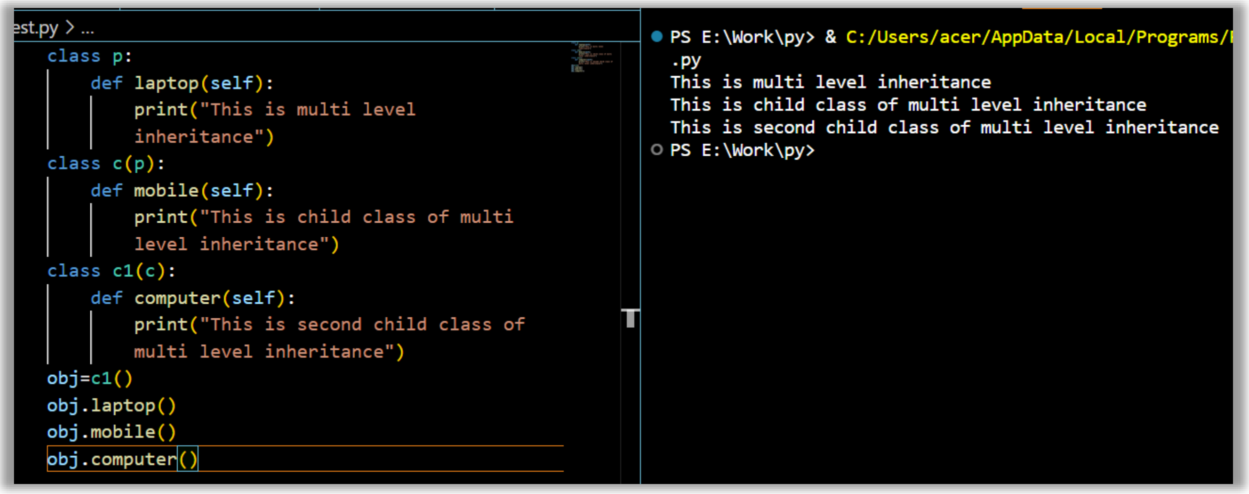
```
est.py > ...
class p1:
    def parent1(self):
        print("This is first parent")
class p2:
    def parent2(self):
        print("This is second parent")
class c(p1,p2):
    def child(self):
        print("This is child class of both
        parent classes")
obj=c()
obj.parent1()
obj.parent2()
obj.child()
```

```
PS E:\Work\py> & C:/Users/acer/AppData/Local
.py
This is first parent
This is second parent
This is child class of both parent classes
PS E:\Work\py>
```

3) Multi-Level Inheritance:

In this type of inheritance, the child class derives from a class [parent] which already derived from another class [grand parent].

In Multi-Level inheritance, we inherit the classes at multiple separate levels.



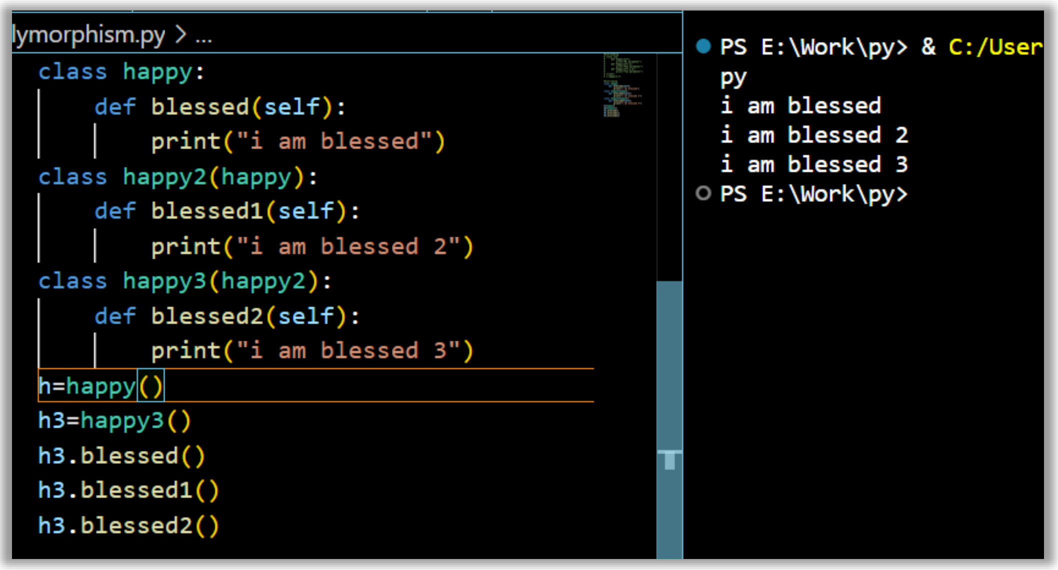
```
est.py > ...
class p:
    def laptop(self):
        print("This is multi level
        inheritance")
class c(p):
    def mobile(self):
        print("This is child class of multi
        level inheritance")
class c1(c):
    def computer(self):
        print("This is second child class of
        multi level inheritance")
obj=c1()
obj.laptop()
obj.mobile()
obj.computer()
```

```
PS E:\Work\py> & C:/Users/acer/AppData/Local/Programs/
.py
This is multi level inheritance
This is child class of multi level inheritance
This is second child class of multi level inheritance
PS E:\Work\py>
```

Polymorphism

Polymorphism simply means having many forms. For example, we need to determine if the given species of birds fly or not, using polymorphism we can do this using a single function.

Method overriding:

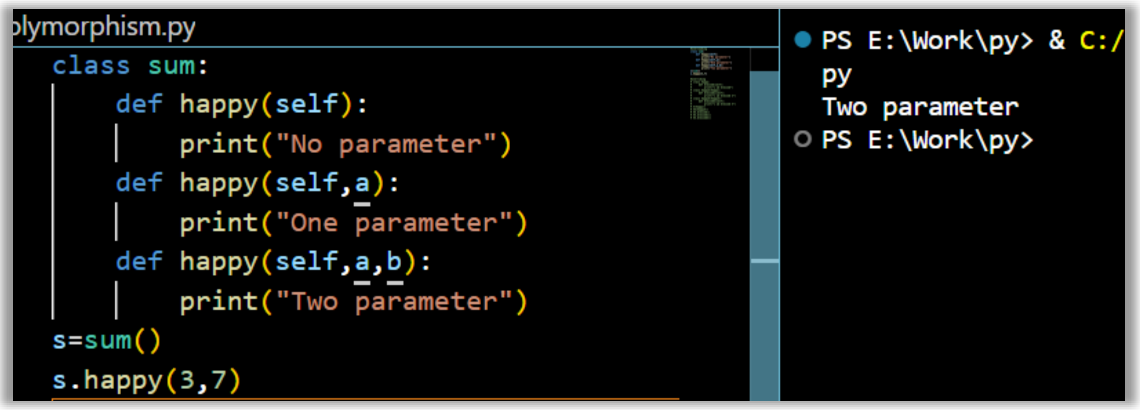


lymorphism.py > ...

```python
class happy:
    def blessed(self):
        print("i am blessed")
class happy2(happy):
    def blessed1(self):
        print("i am blessed 2")
class happy3(happy2):
    def blessed2(self):
        print("i am blessed 3")
h=happy()
h3=happy3()
h3.blessed()
h3.blessed1()
h3.blessed2()
```

```
PS E:\Work\py> & C:/User
py
i am blessed
i am blessed 2
i am blessed 3
PS E:\Work\py>
```

Method overloading:



olymorphism.py

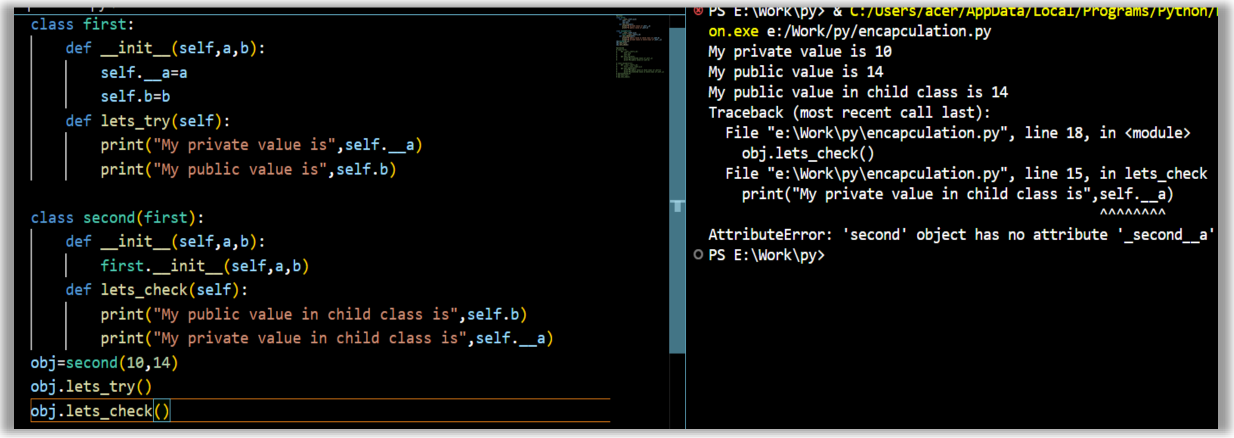
```python
class sum:
    def happy(self):
        print("No parameter")
    def happy(self,a):
        print("One parameter")
    def happy(self,a,b):
        print("Two parameter")
s=sum()
s.happy(3,7)
```

```
PS E:\Work\py> & C:/
py
Two parameter
PS E:\Work\py>
```

Encapsulation

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

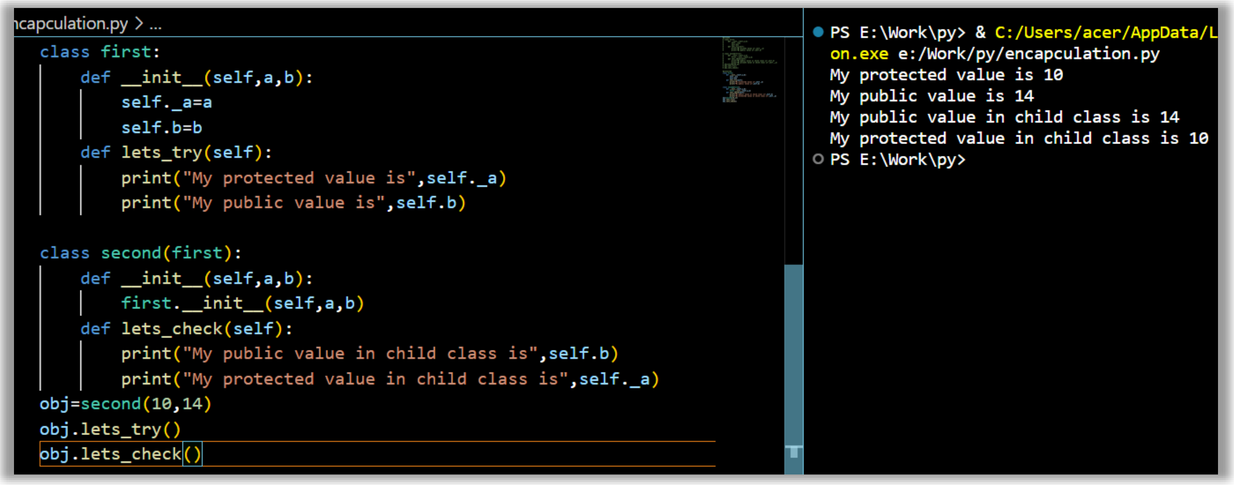
Private:



```python
class first:
    def __init__(self,a,b):
        self.__a=a
        self.b=b
    def lets_try(self):
        print("My private value is",self.__a)
        print("My public value is",self.b)

class second(first):
    def __init__(self,a,b):
        first.__init__(self,a,b)
    def lets_check(self):
        print("My public value in child class is",self.b)
        print("My private value in child class is",self.__a)
obj=second(10,14)
obj.lets_try()
obj.lets_check()
```

```
PS E:\Work\py> & C:/Users/acer/AppData/Local/Programs/Python/
on.exe e:/Work/py/encapculation.py
My private value is 10
My public value is 14
My public value in child class is 14
Traceback (most recent call last):
  File "e:\Work\py\encapculation.py", line 18, in <module>
    obj.lets_check()
  File "e:\Work\py\encapculation.py", line 15, in lets_check
    print("My private value in child class is",self.__a)
                                               ^^^^^^^^
AttributeError: 'second' object has no attribute '_second__a'
PS E:\Work\py>
```

Protected:



ncapculation.py > ...

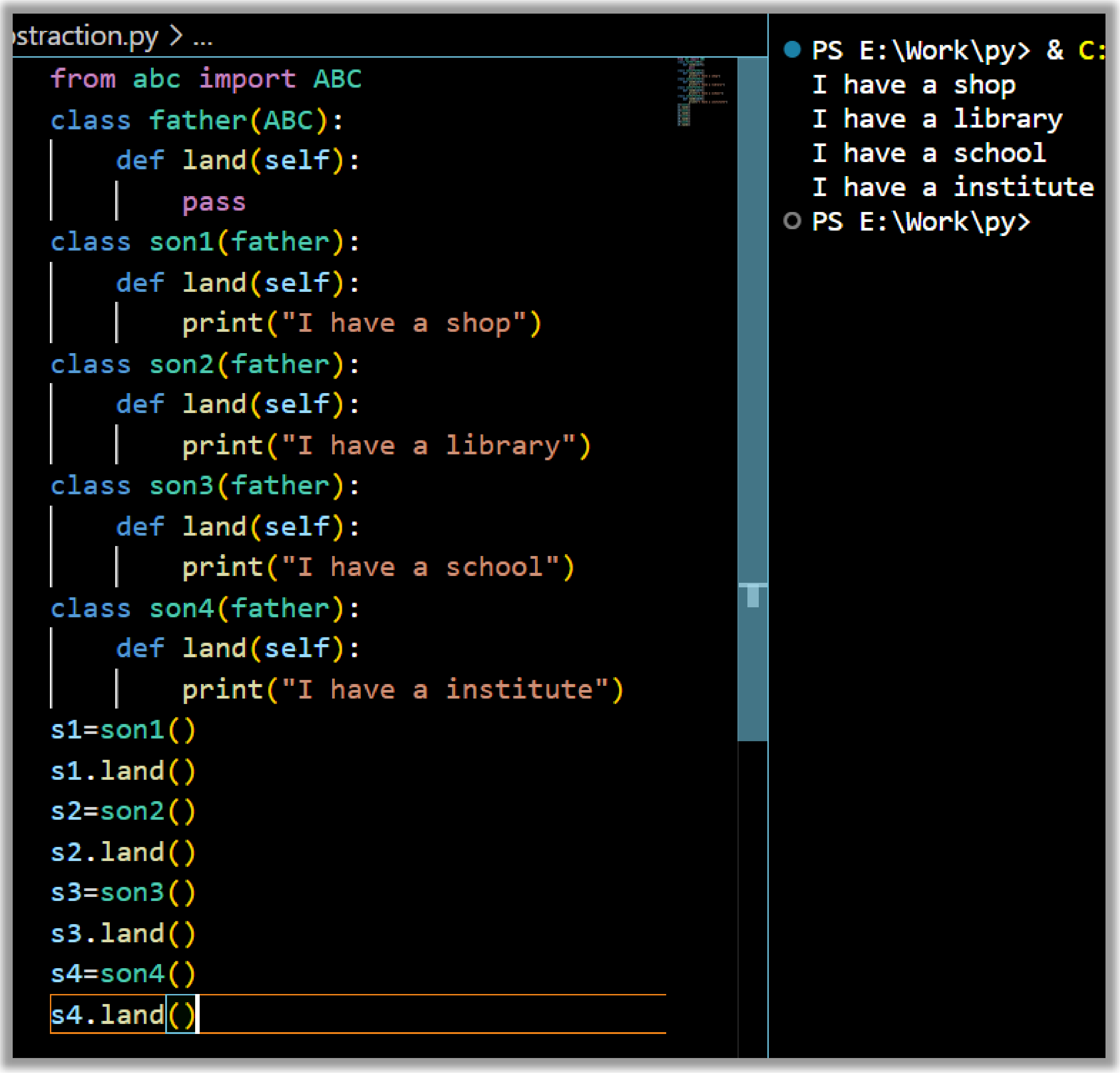
```python
class first:
    def __init__(self,a,b):
        self._a=a
        self.b=b
    def lets_try(self):
        print("My protected value is",self._a)
        print("My public value is",self.b)

class second(first):
    def __init__(self,a,b):
        first.__init__(self,a,b)
    def lets_check(self):
        print("My public value in child class is",self.b)
        print("My protected value in child class is",self._a)
obj=second(10,14)
obj.lets_try()
obj.lets_check()
```

```
● PS E:\Work\py> & C:/Users/acer/AppData/L
  on.exe e:/Work/py/encapculation.py
  My protected value is 10
  My public value is 14
  My public value in child class is 14
  My protected value in child class is 10
○ PS E:\Work\py>
```

Data Abstraction

It hides unnecessary code details from the user. Also, when we do not want to give out sensitive parts of our code implementation and this is where data abstraction came.

Data Abstraction in Python can be achieved by creating abstract classes.



```
straction.py > ...
from abc import ABC
class father(ABC):
    def land(self):
        pass
class son1(father):
    def land(self):
        print("I have a shop")
class son2(father):
    def land(self):
        print("I have a library")
class son3(father):
    def land(self):
        print("I have a school")
class son4(father):
    def land(self):
        print("I have a institute")
s1=son1()
s1.land()
s2=son2()
s2.land()
s3=son3()
s3.land()
s4=son4()
s4.land()
```

```
PS E:\Work\py> & C:
I have a shop
I have a library
I have a school
I have a institute
PS E:\Work\py>
```

Exception Handling

In any Programming language, there are two types of errors are possible.

Syntax Errors

Runtime Errors

- **Syntax Errors:**

The errors which occurs because of invalid syntax are called as Syntax Errors.

- **Runtime Errors:**

Runtime errors are also known as Exceptions.

While executing the Program if something goes wrong because of end user input or programming logic or memory problems etc.

Then we will get these runtime errors.

What is an Exception?

An unwanted and unexpected event that disturbs normal flow of Program is called as an Exception.

e.g.

- ZeroDivisionError
- TypeError
- FileNotFountError
- ValueError
- EOFError
- NameError

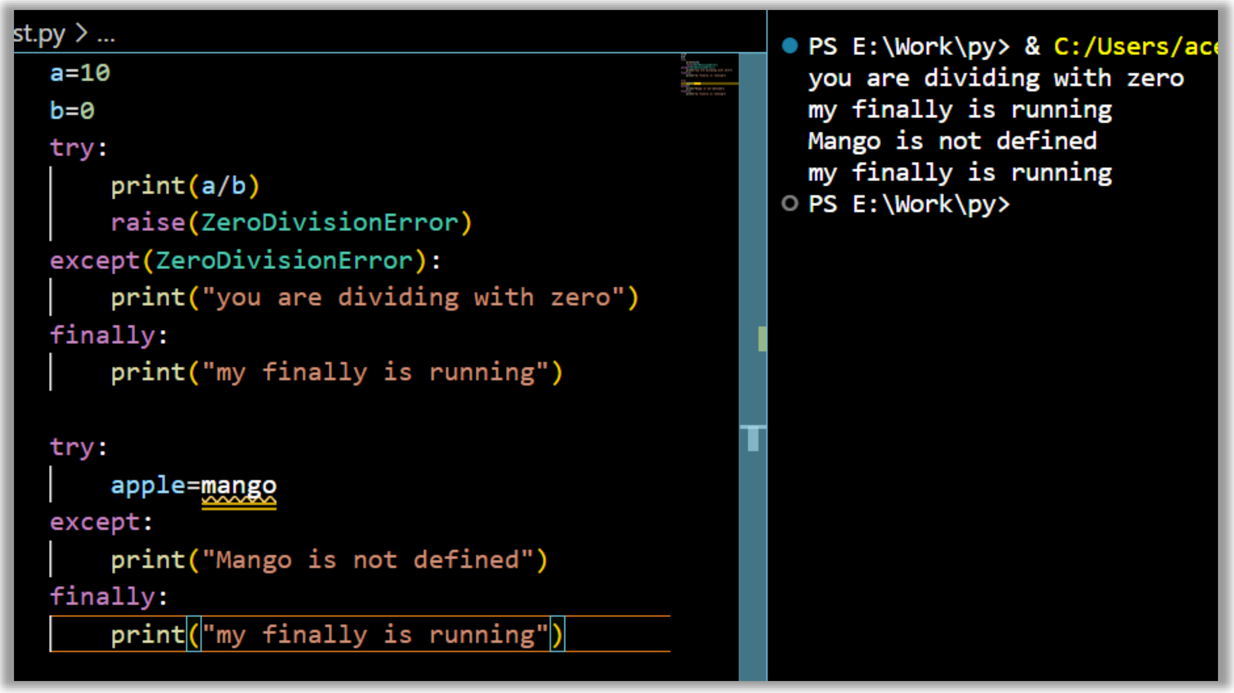
It is highly recommended to handle exception.

The main objective of exception handling is graceful termination of the program (i.e, we should not block our resources and we should not miss anything).

Exception Handling does not mean repairing exception.

We have to define alternative way to continue rest of the program normally.

e.g.



```
st.py > ...
a=10
b=0
try:
    print(a/b)
    raise(ZeroDivisionError)
except(ZeroDivisionError):
    print("you are dividing with zero")
finally:
    print("my finally is running")

try:
    apple=mango
except:
    print("Mango is not defined")
finally:
    print("my finally is running")
```

```
PS E:\Work\py> & C:/Users/ac
you are dividing with zero
my finally is running
Mango is not defined
my finally is running
PS E:\Work\py>
```

User-Defined Exceptions:

It is also known as Programmatic exceptions. Customized exceptions

Sometimes we have to define and raise our own exceptions in the program explicitly to indicate that some thing went wrong.

Such type of exceptions are known as User Defined Exceptions.

try block:

In the try block, we can write some code, which may raise some exceptions.

raise keyword:

This raise keyword is going the raise the exception which is already handled.

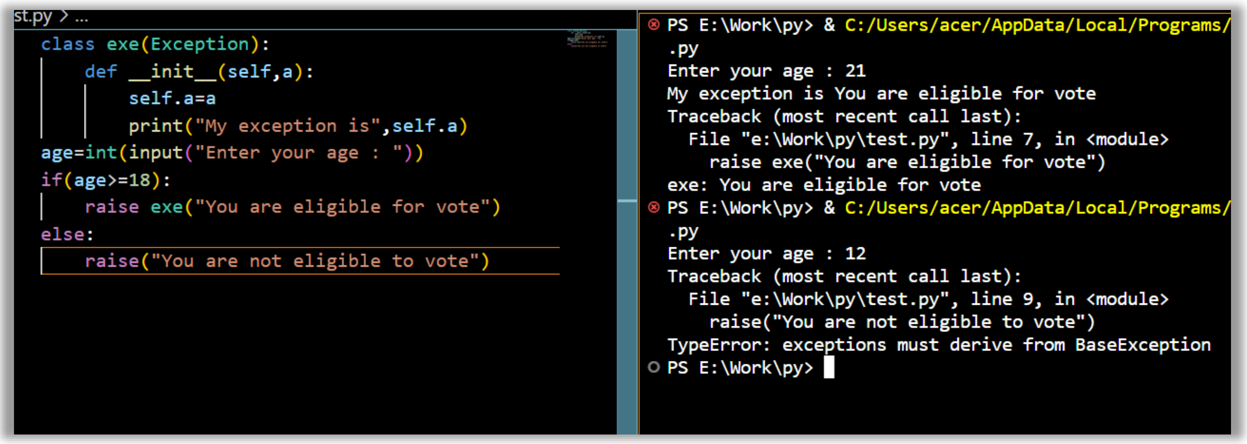
except block:

This block is used to handle the exceptions.

finally block:

The finally block is executed even there is an unhandled exception in the program.

e.g.



```
st.py > ...
class exe(Exception):
    def __init__(self,a):
        self.a=a
        print("My exception is",self.a)
age=int(input("Enter your age : "))
if(age>=18):
    raise exe("You are eligible for vote")
else:
    raise("You are not eligible to vote")
```

```
PS E:\Work\py> & C:/Users/acer/AppData/Local/Programs/
.py
Enter your age : 21
My exception is You are eligible for vote
Traceback (most recent call last):
  File "e:\Work\py\test.py", line 7, in <module>
    raise exe("You are eligible for vote")
exe: You are eligible for vote
PS E:\Work\py> & C:/Users/acer/AppData/Local/Programs/
.py
Enter your age : 12
Traceback (most recent call last):
  File "e:\Work\py\test.py", line 9, in <module>
    raise("You are not eligible to vote")
TypeError: exceptions must derive from BaseException
PS E:\Work\py>
```

Unchecked vs Checked Exceptions:

Unchecked Exceptions:

The Unchecked Exceptions that are not checked at compile time.

These exceptions can handle at runtime only.

These exceptions are not handled by compiler not even compiler is going mention or specify this type of exceptions.

Checked Exceptions:

The Checked Exceptions that are checked at compile time.

If some code within the method throws a checked exception.

Then the method must either handle the exception or it must specify the user the exception is occurred using except block.

File Handling

The main advantage of implementing the files is in order to store the output.

In the concepts like basic operations built-in methods.

We learn till now does not have the option of saving the output data whereas in the files we can save the output data.

Definition of File:

A file is a collection of data stored on a secondary storage device like hard disk.

Types of files:

- Text file [This file we can read]
- Binary file [This file we cannot read]

Rules for Handling files:

Rule:1

To perform reading operation or writing operation on a file first of all we need to create a file then only we have to open a file in any of the access modes like write, read and append.

Syntax:

filepointer=open(newfile,accessmode).

Rule:2

The user's primary responsibility is to close the opened file that means every open file should be closed after its use like after completion of reading and writing operation in any of the access modes (r, w, a).

Syntax:

filepointer.close()

How to create a file:

In order to create a file we need to use built in function called open function. [open()]

Next in order to create a file we need to create one file pointer called as file object in python whereas in C programming we call this as pointer.

Syntax for creating a file:

file_object=open("filename.txt","access_mode")

Access modes:

r=read mode

Rules for accessing read mode:

- It is meant only for reading purpose.

- The file object points at the beginning of the file.
- To open a file in read mode the file must exist.

Syntax:

file_object=open("Myfile.txt","r")

w=write mode

Rules for accessing write mode:

- It is meant only for writing purpose.
- The file object points at the beginning of the file.
- If the file already exists then the file is opened and file pointer points at the beginning of the file and the existing data of the file will be over written.
- If the file does not exists then new file will be created with the given file name.

Syntax:

file_object=open("newfilename.txt","w")
a=append mode

Rules for accessing append mode:

- It is meant only for appending and writing the data.
- The file object points at the end of the content of the file.

- If the file does not exists then new file will be created and file pointer points at the beginning of the file.

Syntax:

file_object=open("newfilename.txt","a")

for text files the access modes available are as follows:

r for reading, w for writing, a for appending/writing, r+ for both reading and writing, w+ for both reading and writing, a+ for reading and appending.

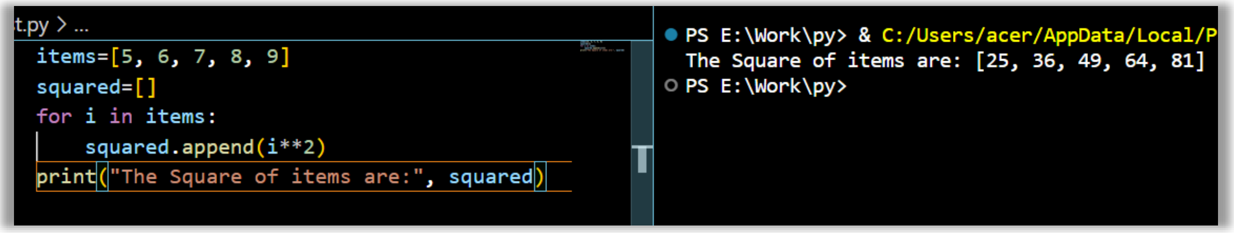
Map Function

Map applies a function to all the items in an input_list.

Here is the Syntax:

map(function_to_apply, list_of_inputs)

Most of the times we want to pass all the list elements to a function one-by-one and then collect the output.



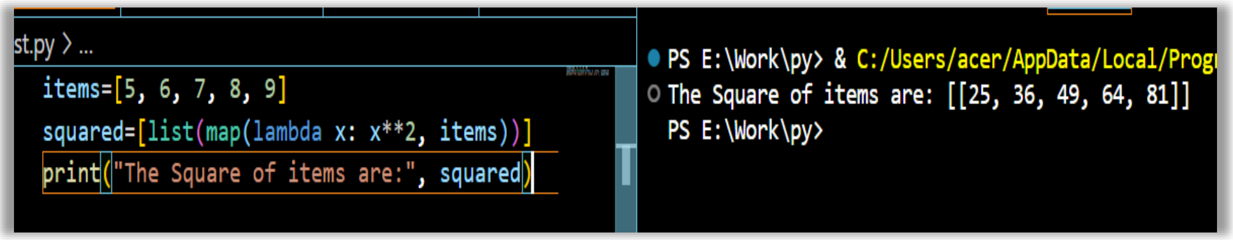
```
items=[5, 6, 7, 8, 9]
squared=[]
for i in items:
    squared.append(i**2)
print("The Square of items are:", squared)
```

```
PS E:\Work\py> & C:/Users/acer/AppData/Local/P
The Square of items are: [25, 36, 49, 64, 81]
PS E:\Work\py>
```

For instance

Map allows us to implement this in a much simpler and nicer way.

Program using Map function:



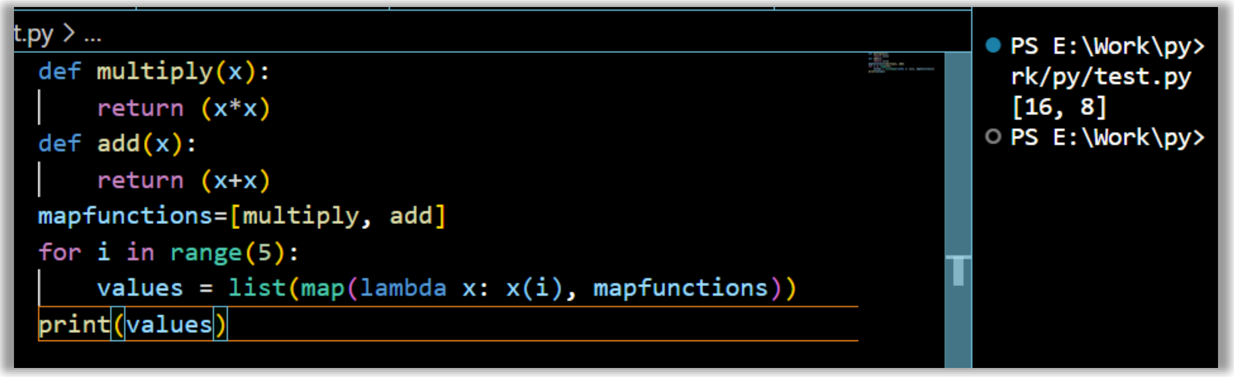
```
st.py > ...
  items=[5, 6, 7, 8, 9]
  squared=[list(map(lambda x: x**2, items))]
  print("The Square of items are:", squared)
```

```
PS E:\Work\py> & C:/Users/acer/AppData/Local/Prog
The Square of items are: [[25, 36, 49, 64, 81]]
PS E:\Work\py>
```

Most of the times we use lambdas with map so we did the same.

Instead of a list of inputs we can even have a list of functions!

e.g.



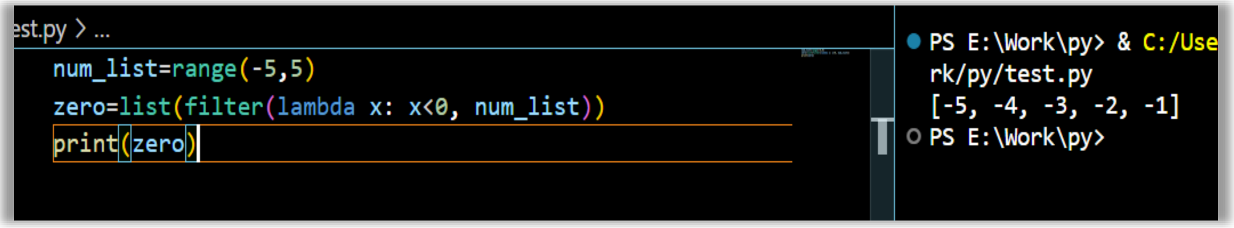
```
t.py > ...
def multiply(x):
    return (x*x)
def add(x):
    return (x+x)
mapfunctions=[multiply, add]
for i in range(5):
    values = list(map(lambda x: x(i), mapfunctions))
print(values)
```

```
PS E:\Work\py>
rk/py/test.py
[16, 8]
PS E:\Work\py>
```

Filter Function

As the name suggests, filter creates a list of elements for which a function returns true.

e.g.



```
est.py > ...
    num_list=range(-5,5)
    zero=list(filter(lambda x: x<0, num_list))
    print(zero)
```

```
PS E:\Work\py> & C:/Use
rk/py/test.py
[-5, -4, -3, -2, -1]
PS E:\Work\py>
```

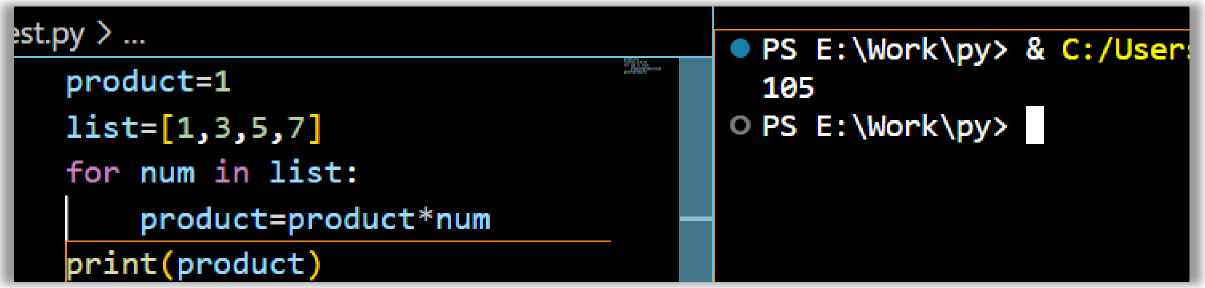
The filter resembles a for loop but it is a built-in function and faster.

Reduce Function

Reduce is a really useful function for performing some computation on a list and returning the result.

It applies a rolling computation to sequential pairs of values in a list.

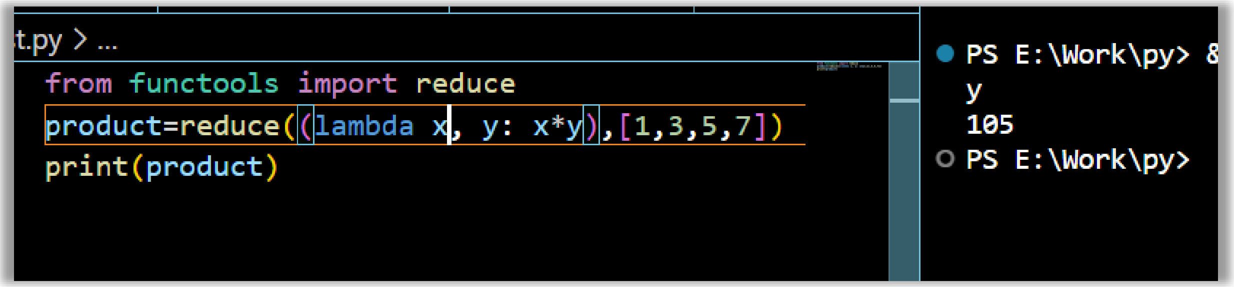
For instance:



```
est.py > ...
    product=1
    list=[1,3,5,7]
    for num in list:
        product=product*num
    print(product)
```

```
PS E:\Work\py> & C:/User
105
PS E:\Work\py>
```

Reduce Function:



```
t.py > ...
  from functools import reduce
  product=reduce((lambda x, y: x*y),[1,3,5,7])
  print(product)
```

```
PS E:\Work\py> &
y
105
PS E:\Work\py>
```

Enumerate

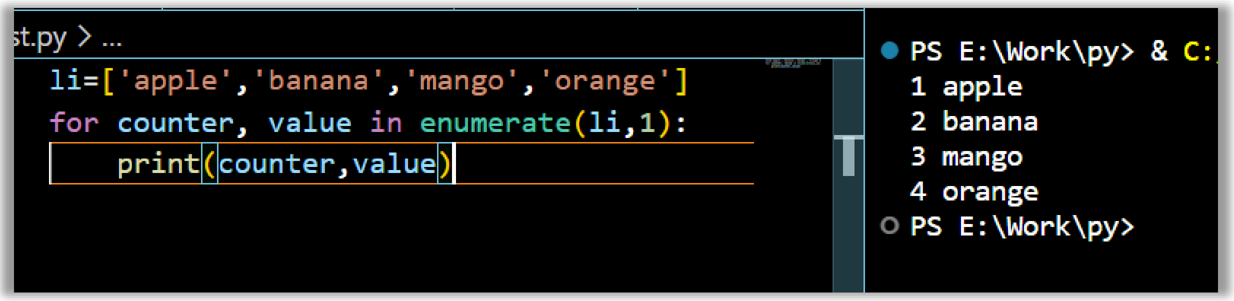
Enumerate is a built-in function of Python.

Its usefulness cannot be summarized in a single line.

Yet most of the newcomers and even some advanced programmers are unaware of it.

It allows us to loop over something and have an automatic counter.

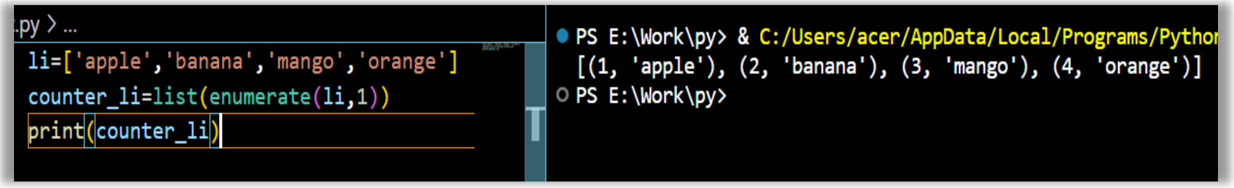
e.g.



```
t.py > ...
  li=['apple','banana','mango','orange']
  for counter, value in enumerate(li,1):
      print(counter,value)
```

```
PS E:\Work\py> & C:
1 apple
2 banana
3 mango
4 orange
PS E:\Work\py>
```

Enumerate using List:



```
.py > ...
li=['apple','banana','mango','orange']
counter_li=list(enumerate(li,1))
print(counter_li)
```

```
PS E:\Work\py> & C:/Users/acer/AppData/Local/Programs/Pythor
[(1, 'apple'), (2, 'banana'), (3, 'mango'), (4, 'orange')]
PS E:\Work\py>
```

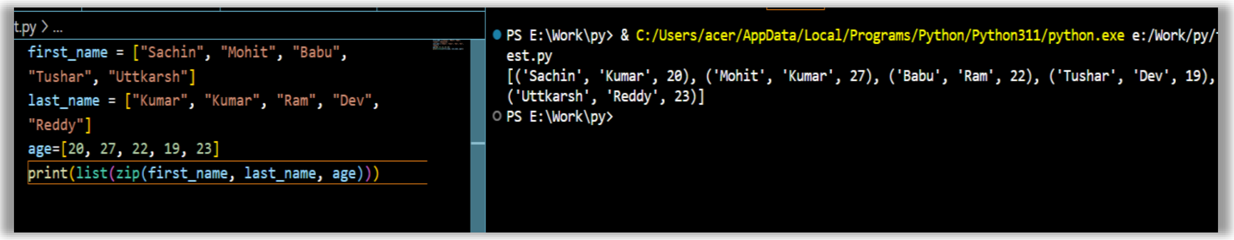
Zip & Unzip

Zip Function:

Zip is a useful function that allows you to combine two lists easily.

After calling zip, an iterator is returned. In order to see the content wrapped inside, we need to first convert it to a list.

e.g.



```
t.py > ...
first_name = ["Sachin", "Mohit", "Babu",
"Tushar", "Uttkarsh"]
last_name = ["Kumar", "Kumar", "Ram", "Dev",
"Reddy"]
age=[20, 27, 22, 19, 23]
print(list(zip(first_name, last_name, age)))
```

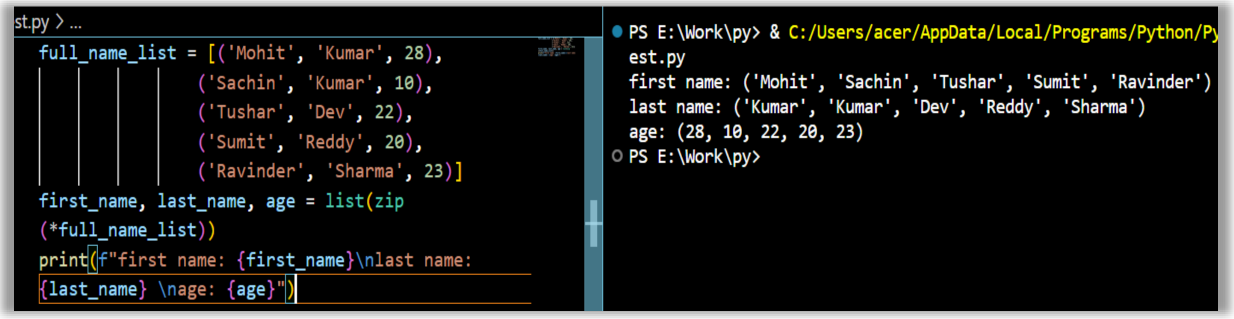
```
PS E:\Work\py> & C:/Users/acer/AppData/Local/Programs/Python/Python311/python.exe e:/Work/py/t
est.py
[('Sachin', 'Kumar', 20), ('Mohit', 'Kumar', 27), ('Babu', 'Ram', 22), ('Tushar', 'Dev', 19),
('Uttkarsh', 'Reddy', 23)]
PS E:\Work\py>
```

Unzip Function:

We can use the zip function to unzip a list as well. This time, we need an input of a list with an asterisk before it.

The outputs are the separated lists.

e.g.



```
st.py > ...
full_name_list = [('Mohit', 'Kumar', 28),
                  ('Sachin', 'Kumar', 10),
                  ('Tushar', 'Dev', 22),
                  ('Sumit', 'Reddy', 20),
                  ('Ravinder', 'Sharma', 23)]
first_name, last_name, age = list(zip
(*full_name_list))
print(f"first name: {first_name}\nlast name:
{last_name} \nage: {age}")
```

```
PS E:\Work\py> & C:/Users/acer/AppData/Local/Programs/Python/Py
est.py
first name: ('Mohit', 'Sachin', 'Tushar', 'Sumit', 'Ravinder')
last name: ('Kumar', 'Kumar', 'Dev', 'Reddy', 'Sharma')
age: (28, 10, 22, 20, 23)
PS E:\Work\py>
```

Call by Reference v/s Call by Value

Call by Reference	Call by Value
While calling a function, in a programming language instead of copying the values of variables, the address of the variables is used, it is known as "Call By Reference".	While calling a function, when we pass values by copying variables, it is known as "Call By Values".
In this method, a variable itself is passed.	A copy of the variable is passed in a call by value.
Change in the variable also affects the value of the variable outside the function.	Changes made in a copy of a variable never modify the value of the variable outside the function.

Allows you to make changes in the values of variables by using function calls.	Does not allow you to make any changes in the actual variables.
The original value is modified.	Original value not modified.

Call by Reference:

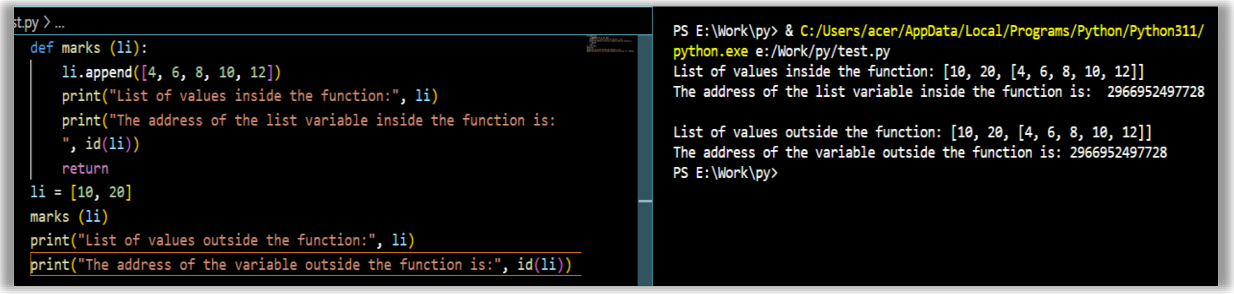
Definition:

When we pass address as an argument to a function then it is called as Call by Reference.

When we call a function using call by reference, then the original value is passed to the function and hence if any change is there, it changes the original value as well.

Python acts as Call by reference if passed data is mutable data types like lists, sets and dictionaries.

e.g.



```
def marks (li):
    li.append([4, 6, 8, 10, 12])
    print("List of values inside the function:", li)
    print("The address of the list variable inside the function is:
    ", id(li))
    return
li = [10, 20]
marks (li)
print("List of values outside the function:", li)
print("The address of the variable outside the function is:", id(li))
```

```
PS E:\Work\py> & C:/Users/acer/AppData/Local/Programs/Python/Python311/
python.exe e:/Work/py/test.py
List of values inside the function: [10, 20, [4, 6, 8, 10, 12]]
The address of the list variable inside the function is:  2966952497728

List of values outside the function: [10, 20, [4, 6, 8, 10, 12]]
The address of the variable outside the function is: 2966952497728
PS E:\Work\py>
```

Call by Value:

Definition:

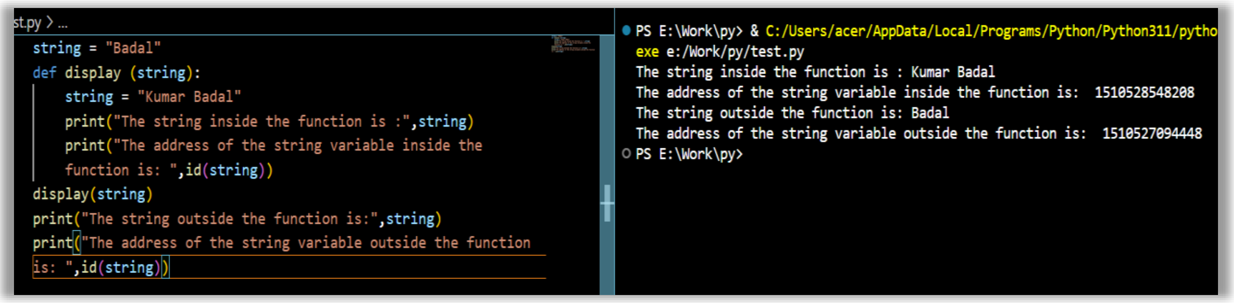
When we pass value as an argument to a function then it is called as Call by Value.

When we call a function using call by value, then the copy of variable is passed to the function.

So if there is any change to the variable in that function, then the original value is not going to change.

Python acts as Call by Value if passed data is immutable data types like tuples, strings and numbers.

e.g.



```
st.py > ...
string = "Badal"
def display (string):
    string = "Kumar Badal"
    print("The string inside the function is :",string)
    print("The address of the string variable inside the
    function is: ",id(string))
display(string)
print("The string outside the function is:",string)
print("The address of the string variable outside the function
is: ",id(string))
```

```
PS E:\Work\py> & C:/Users/acer/AppData/Local/Programs/Python/Python311/pytho
exe e:/Work/py/test.py
The string inside the function is : Kumar Badal
The address of the string variable inside the function is:  1510528548208
The string outside the function is: Badal
The address of the string variable outside the function is:  1510527094448
PS E:\Work\py>
```