Python

Definition:
Python is a high-level scripting language which can be used for a variety of text processing, system administration and internet related tasks. Python is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable. It uses English keywords frequently where as other languages use punctuation, and it has fewer syntactical constructions than other languages.

 Python is Interpreted: Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.
 Python is Interactive: You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
 Python is Object-Oriented: Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
 Python is a Beginner's Language: Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

History of Python

 Python was developed by **Guido van Rossum** in the late eighties and early nineties at the National Research Institute for Mathematics and Computer Science in the Netherlands.
 Python is derived from many other languages, including ABC, Modula-3, C, C++, Algol-68, SmallTalk, Unixshell, and other scripting languages.

Characteristics of Python

Following are important characteristics of **Python Programming** −

- It supports functional and structured programming methods as well as OOP.
- It can be used as a scripting language or can be compiled to byte-code for building large applications.
- It provides very high-level dynamic data types and supports dynamic type checking.
- It supports automatic garbage collection.
- It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.

Python Data Types

There are different types of data types in Python. Some built-in Python data types are:

- **Numeric data types**: *int, float, complex*
- **String data types**: *str*
- **Sequence types**: *list, tuple, range*
- **Binary types**: *bytes, bytearray, memoryview*
- **Mapping data type**: *dict*
- **Boolean type**: *bool*
- **Set data types**: *set, frozenset*

Python Numeric Data Type

Python numeric data type is used to hold numeric values like;

1. int - holds signed integers of non-limited length.
2. long- holds long integers
3. float- holds floating precision numbers and it's accurate up to 15 decimal places.
4. complex- holds complex numbers.

In Python, we need not declare a datatype while declaring a variable like C or C++. We can simply just assign values in a variable.

But if we want to see what type of numerical value is it holding right now, we can use **type( ).**

```
#create a variable with integer value.
a=100
print("The type of variable having value", a, " is ", type(a))

#create a variable with float value.
b=10.2345
print("The type of variable having value", b, " is ", type(b))

#create a variable with complex value.
c=100+3j
print("The type of variable having value", c, " is ", type(c))
```

If you run the above code you will see output like the below image.

```
The type of variable having value 100  is  <class 'int'>
The type of variable having value 10.2345  is  <class 'float'>
The type of variable having value (100+3j)  is  <class 'complex'>
```

Python String Data Type

The string is a sequence of characters. Python supports Unicode characters. Generally, strings are represented by either single or double-quotes.

```
a = "string in a double quote"
b= 'string in a single quote'
print(a)
print(b)

# using ',' to concatenate the two or several strings
print(a,"concatenated with",b)

#using '+' to concate the two or several strings
print(a+" concated with "+b)
```

```
string in a double quote
string in a single quote
string in a double quote concated with string in a single quote
string in a double quote concated with string in a single quote
```

Python List Data Type

The list is a versatile data type exclusive in Python. In a sense, it is the same as the array in C/C++. But the interesting thing about the list in Python is it can simultaneously hold different types of data. Formally list is an ordered sequence of some data written using square brackets([]) and commas(,).

```
#list of having only integers
a= [1,2,3,4,5,6]
print(a)

#list of having only strings
b=["hello","john","reese"]
print(b)

#list of having both integers and strings
c= ["hey","you",1,2,3,"go"]
print(c)

#index are 0 based. this will print a single character
print(c[1]) #this will print "you" in list c
```

```
[1, 2, 3, 4, 5, 6]
['hello', 'john', 'reese']
['hey', 'you', 1, 2, 3, 'go']
you
```

Python Tuple

The tuple is another data type which is a sequence of data similar to a list. But it is immutable. That means data in a tuple is write-protected. Data in a tuple is written using parenthesis and commas.

```
#tuple having only integer type of data.
a=(1,2,3,4)
print(a) #prints the whole tuple

#tuple having multiple type of data.
b=("hello", 1,2,3,"go")
print(b) #prints the whole tuple

#index of tuples are also 0 based.

print(b[4]) #this prints a single element in a tuple, in this case "go"
```

```
(1, 2, 3, 4)
('hello', 1, 2, 3, 'go')
go
```

Python Dictionary

Python Dictionary is an unordered sequence of data of key-value pair form. It is similar to the hash table type. Dictionaries are written within curly braces in the form **key:value.** It is very useful to retrieve data in an optimized way among a large amount of data.

```
#a sample dictionary variable

a = {1:"first name",2:"last name", "age":33}

#print value having key=1
print(a[1])
#print value having key=2
print(a[2])
#print value having key="age"
print(a["age"])
```

```
first name
last name
33
```

Python Sets

Sets are used to store multiple items in a single variable.

Set is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Tuple, and Dictionary, all with different qualities and usage.

A set is a collection which is *unordered*, *unchangeable**, and *unindexed*.

*** Note:** Set *items* are unchangeable, but you can remove items and add new items.

Sets are written with curly brackets.

Example

Create a Set:

```
thisset = {"apple", "banana", "cherry"}
print(thisset)
```

- Set items are unordered, unchangeable, and do not allow duplicate values.
- Unordered means that the items in a set do not have a defined order.

Input and Output in Python

How to Take Input from User in Python
Sometimes a developer might want to take user input at some point in the program. To do this Python provides an input() function.

Syntax:
input('prompt')

where prompt is an optional string that is displayed on the string at the time of taking input.

Example: Python get user input with a message

```
# Taking input from the user
name = input("Enter your name: ")

# Output
print("Hello, " + name)
print(type(name))
```

Output:
Enter your name: GFG

Hello, GFG

<class 'str'>

Note: Python takes all the input as a string input by default. To convert it to any other data type we have to convert the input explicitly. For example, to convert the input to int or float we have to use the int() and float()

Example: Integer input in Python

```
# Taking input from the user as integer
num = int(input("Enter a number: "))

add = num + 1

# Output
print(add)
```

Output:
Enter a number: 25

26

To take Multiple Inputs in Python:
We can take multiple inputs of the same data type at a time in python, using map( ) method in python.

```
a, b, c = map(int, input("Enter the Numbers : ").split())
print("The Numbers are : ",end = " ")
print(a, b, c)
```

Output :
Enter the Numbers : 2 3 4

The Numbers are : 2 3 4

How take inputs for the Sequence Data Types like List, Set, Tuple, etc.
In the case of List and Set the input can be taken from the user in two ways.

1. Taking List/Set elements one by one by using the append()/add() methods.
2. Using map() and list() / set() methods.

Taking List/Set elements one by one
Take the elements of the List/Set one by one and use the append() method in the case of List, and add() method in the case of a Set, to add the elements to the List / Set.

```
List = list()
Set = set()
l = int(input("Enter the size of the List : "))
s = int(input("Enter the size of the Set  : "))
print("Enter the List elements : ")
for i in range(0, l):
    List.append(int(input()))
print("Enter the Set elements : ")
for i in range(0, s):
    Set.add(int(input()))
print(List)
print(Set)
```

Output :
Enter the size of the List : 4

Enter the size of the Set : 3

Enter the List elements :

9

0

1

3

Enter the Set elements :

2

9

1

[9, 0, 1, 3]

{9, 2, 1}

Using map() and list() / set() Methods

```
List = list(map(int, input("Enter List elements : ").split()))
Set = set(map(int, input("Enter the Set elements :").split()))
print(List)
print(Set)
```

Output :
Enter List elements : 3 4 8 9 0 11

Enter the Set elements :34 88 230 234 123

[3, 4, 8, 9, 0, 11]

{34, 230, 234, 88, 123}

Taking Input for Tuple

We know that tuples are immutable, there are no methods available to add elements to tuples. To add a new element to a tuple, first type cast the tuple to the list, later append the element to the list, and again type cast list back to a tuple.

```
T = (2, 3, 4, 5, 6)
print("Tuple before adding new element")
print(T)
L = list(T)
L.append(int(input("Enter the new element : ")))
T = tuple(L)
print("Tuple After adding the new element")
print(T)
```

Output :
Tuple before adding new element

(2, 3, 4, 5, 6)

Enter the new element : 35

Tuple After adding the new element

(2, 3, 4, 5, 6, 35)

Display Output in Python

Python provides the print() function to display output to the standard output devices.

Syntax: *print(value(s), sep= ' ', end = '\n', file=file, flush=flush)*

Parameters:
value(s) : *Any value, and as many as you like. Will be converted to string before printed*
sep='separator' : *(Optional) Specify how to separate the objects, if there is more than one.Default :' '*
end='end': *(Optional) Specify what to print at the end.Default : '\n'*
file : *(Optional) An object with a write method. Default :sys.stdout*
flush : *(Optional) A Boolean, specifying if the output is flushed (True) or buffered (False). Default: False*
Returns: *It returns output to the screen.*

Example: Python Print Output

```
# Python program to demonstrate
# print() method
print("GFG")

# code for disabling the softspace feature
print('G', 'F', 'G')
```

Output

```
GFG
G F G
```

In the above example, we can see that in the case of the 2nd print statement there is a space between every letter and the print statement always add a new line character at the end of the string. This is because after every character the sep parameter is printed and at the end of the string the end parameter is printed.

Example: Python Print output with custom sep and end parameter

```
# Python program to demonstrate
# print() method
print("GFG", end = "@")

# code for disabling the softspace feature
print('G', 'F', 'G', sep="#")
```

Output

```
GFG@G#F#G
```

Python String formatting using F string

```
# Declaring a variable
name = "Gfg"

# Output
print(f'Hello {name}! How are you?')
```

Output:

```
Hello Gfg! How are you?
```

Using % Operator

We can use **'%'** operator. % values are replaced with zero or more value of elements. The formatting using % is similar to that of 'printf' in the C programming language.

- %d – integer
- %f – float
- %s – string

```
# Taking input from the user
num = int(input("Enter a value: "))

add = num + 5

# Output
print("The sum is %d" %add)
```

Output:

```
Enter a value: 50

The sum is 55
```

Comments in Python

Single-line comment

Hash character(#) is used to comment the line in the Python program. Comments does not have to be text to explain the code, it can also be used to prevent Python from executing code.

The hash character should be placed before the line to be commented.

Example: How to Make Single Line Comments in Python

```
# Write Python3 code here
# Single line comment used

print("Python Comments")

# print("Mathematics")
```

Output:

```
Python Comments
```

Multi-line string as a comment

Python multi-line comment is a piece of text enclosed in a delimiter (“”””) on each end of the comment. Again there should be no white space between delimiters (“”””).

They are useful when the comment text does not fit into one line; therefore needs to span across lines. Multi-line comments or paragraphs serve as documentation for others reading your code.

Example: How to Make Multi-line Comments in Python

```
# Write Python code here

""" Multi-line comment used
print("Python Comments") """

print("Mathematics")
```

Output:

```
Mathematics
```

Try and Except Statement – Catching Exceptions

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

Example: Let us try to access the array element whose index is out of bound and handle the corresponding exception.

```
# Python program to handle simple runtime error
#Python 3

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

    # Throws error since there are only 3 elements in array
    print ("Fourth element = %d" %(a[3]))

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

Output

```
Second element = 2

An error occurred
```

Catching Specific Exception

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

```
try:

    # statement(s)

except IndexError:

    # statement(s)

except ValueError:

    # statement(s)
```

Example

Print one message if the try block raises a NameError and another for other errors:

```
try:
  print(x)
except NameError:
  print("Variable x is not defined")
except:
  print("Something else went wrong")
```

Else

You can use the else keyword to define a block of code to be executed if no errors were raised:

Example

In this example, the try block does not generate any error:

```
try:
  print("Hello")
except:
  print("Something went wrong")
else:
  print("Nothing went wrong")
```

Output

```
Hello
Nothing went wrong
```

Finally

The finally block, if specified, will be executed regardless if the try block raises an error or not.

Example

```
try:
  print(x)
except:
  print("Something went wrong")
finally:
  print("The 'try except' is finished")
```

Syntax:

```
try:
    # Some Code....
except:
    # optional block
    # Handling of exception (if required)

else:
    # execute if no exception

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

Example

```
# Python program to demonstrate finally

# No exception Exception raised in try block
try:
    k = 5//0  # raises divide by zero exception.
    print(k)

# handles zerodivision exception
except ZeroDivisionError:
    print("Can't divide by zero")

finally:
    # this block is always executed
    # regardless of exception generation.
    print('This is always executed')
```

Output:

```
Can't divide by zero

This is always executed
```

Conditional Statements In Python

1) if statements

Python if statement is one of the most commonly used conditional statements in programming languages. It decides whether certain statements need to be executed or not. It checks for a given condition, if the condition is true, then the set of code present inside the ” if ” block will be executed otherwise not.

The if condition evaluates a Boolean expression and executes the block of code only when the Boolean expression becomes TRUE.

Syntax:

```
If ( EXPRESSION == TRUE ):
    Block of code
else:
    Block of code
```

Example:

```
num = 5
if (num < 10):
    print("Num is smaller than 10")

print("This statement will always be executed")
```

Output: Num is smaller than 10.
This statement will always be executed.

2) if-else statements

The statement itself says if a given condition is true then execute the statements present inside the "if block" and if the condition is false then execute the "else" block.

The "else" block will execute only when the condition becomes false. It is the block where you will perform some actions when the condition is not true.

if-else statement evaluates the Boolean expression. If the condition is TRUE then, the code present in the " if " block will be executed otherwise the code of the "else" block will be executed

Syntax:

```
If (EXPRESSION == TRUE):

    Statement (Body of the block)

else:

    Statement (Body of the block)
```

Example:

```
num = 5
if(num > 10):
    print("number is greater than 10")
else:
    print("number is less than 10")

print ("This statement will always be executed"
```

3) elif statements

In Python, we have one more conditional statement called “elif” statements. “elif” statement is used to check multiple conditions only if the given condition is false. It’s similar to an “if-else” statement and the only difference is that in “else” we will not check the condition but in “elif” we will check the condition.

“elif” statements are similar to “if-else” statements but “elif” statements evaluate multiple conditions.

Syntax:

```
if (condition):
        #Set of statement to execute if condition is true
elif (condition):
        #Set of statements to be executed when if condition is false and elif condition is true
else:
      #Set of statement to be executed when both if and elif conditions are false
```

Example:

```
num = 10
if (num == 0):
    print(“Number is Zero”)

elif (num > 5):
      print(“Number is greater than 5”)

else:
      print(“Number is smaller than 5”)
```

Output:

Number is greater than 5

4) Nested if-else statements

Nested “if-else” statements mean that an “if” statement or “if-else” statement is present inside another if or if-else block. Python provides this feature as well, this in turn will help us to check multiple conditions in a given program.

An “if” statement is present inside another “if” statement which is present inside another “if” statements and so on.

Nested if Syntax:

```
if(condition):
        #Statements to execute if condition is true
        if(condition):
              #Statements to execute if condition is true
        #end of nested if
#end of if
```

Loops in python

While Loop in Python

In python, a while loop is used to execute a block of statements repeatedly until a given condition is satisfied. And when the condition becomes false, the line immediately after the loop in the program is executed.

Syntax :

```
while expression:
    statement(s)
```

All the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. Python uses indentation as its method of grouping statements.

Example:

```
# Python program to illustrate
# while loop
count = 0
while (count < 3):
    count = count + 1
    print("Hello World")
```

Output:

```
Hello World

Hello World

Hello World
```

For Loop in Python

For loops are used for sequential traversal. For example: traversing a list or string or array etc. In Python, there is no C style for loop, i.e., for (i=0; i<n; i++). There is "for in" loop which is similar to for each loop in other languages. Let us learn how to use for in loop for sequential traversals.

Syntax:

```
for iterator_var in sequence:
    statements(s)
```

```
# Python program to illustrate
# Iterating over range 0 to n-1

n = 4
for i in range(0, n):
    print(i)
```

Output :

```
0

1

2

3
```

Tuple Items - Data Types

- Tuple items can be of any data type:

Example

String, int and boolean data types:

```
tuple1 = ("apple", "banana", "cherry")
tuple2 = (1, 5, 7, 9, 3)
tuple3 = (True, False, False)
```

- A tuple can contain different data types:

Example

A tuple with strings, integers and boolean values:

```
tuple1 = ("abc", 34, True, 40, "male")
```

type()

- From Python's perspective, tuples are defined as objects with the data type 'tuple':

```
<class 'tuple'>
```

Example

What is the data type of a tuple?

```
mytuple = ("apple", "banana", "cherry")
print(type(mytuple))
```

Access Tuple Items

- You can access tuple items by referring to the index number, inside square brackets:

Example

Print the second item in the tuple:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[1])
```

Negative Indexing

- Negative indexing means start from the end.

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

Example

Print the last item of the tuple:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[-1])
```

Range of Indexes

- You can specify a range of indexes by specifying where to start and where to end the range.
- When specifying a range, the return value will be a new tuple with the specified items.

Example

Return the third, fourth, and fifth item:

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[2:5])
```

Range of Negative Indexes

Specify negative indexes if you want to start the search from the end of the tuple:

Example

This example returns the items from index -4 (included) to index -1 (excluded)

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[-4:-1])
```

Check if Item Exists

To determine if a specified item is present in a tuple use the `in` keyword:

Example

Check if "apple" is present in the tuple:

```
thistuple = ("apple", "banana", "cherry")
if "apple" in thistuple:
  print("Yes, 'apple' is in the fruits tuple")
```

Python - Update Tuples

Tuples are unchangeable, meaning that you cannot change, add, or remove items once the tuple is created.

But there are some workarounds.

Change Tuple Values

Once a tuple is created, you cannot change its values. Tuples are **unchangeable**, or **immutable** as it also is called.

But there is a workaround. You can convert the tuple into a list, change the list, and convert the list back into a tuple.

Example

Convert the tuple into a list to be able to change it:

```
x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
x = tuple(y)

print(x)
```

O/P

```
("apple", "kiwi", "cherry")
```

Add Items

Since tuples are immutable, they do not have a build-in `append()` method, but there are other ways to add items to a tuple.

1. **Convert into a list**: Just like the workaround for *changing* a tuple, you can convert it into a list, add your item(s), and convert it back into a tuple.

Example

Convert the tuple into a list, add "orange", and convert it back into a tuple:

```
thistuple = ("apple", "banana", "cherry")
y = list(thistuple)
y.append("orange")
thistuple = tuple(y)
```

2. **Add tuple to a tuple**. You are allowed to add tuples to tuples, so if you want to add one item, (or many), create a new tuple with the item(s), and add it to the existing tuple:

Example

Create a new tuple with the value "orange", and add that tuple:

```
thistuple = ("apple", "banana", "cherry")
y = ("orange",)
thistuple += y

print(thistuple)
```

Remove Items

Note: You cannot remove items in a tuple.

Tuples are **unchangeable**, so you cannot remove items from it, but you can use the same workaround as we used for changing and adding tuple items:

Example

Convert the tuple into a list, remove "apple", and convert it back into a tuple:

```
thistuple = ("apple", "banana", "cherry")
y = list(thistuple)
y.remove("apple")
thistuple = tuple(y)
```

you can delete the tuple completely:

Example

The `del` keyword can delete the tuple completely:

```
thistuple = ("apple", "banana", "cherry")
del thistuple
print(thistuple) #this will raise an error because the tuple no longer exists
```

Loop Through a Tuple

You can loop through the tuple items by using a `for` loop.

Example

Iterate through the items and print the values:

```
thistuple = ("apple", "banana", "cherry")
for x in thistuple:
  print(x)
```

O/P

```
apple
banana
cherry
```

Set

Sets are used to store multiple items in a single variable.

Set is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Tuple, and Dictionary, all with different qualities and usage.

A set is a collection which is *unordered*, *unchangeable**, and *unindexed*.

*** Note:** Set *items* are unchangeable, but you can remove items and add new items.

Sets are written with curly brackets.

Example

Create a Set:

```
thisset = {"apple", "banana", "cherry"}
print(thisset)
```

Duplicates Not Allowed

Sets cannot have two items with the same value.

Example

Duplicate values will be ignored:

```
thisset = {"apple", "banana", "cherry", "apple"}

print(thisset)
```

```
{'banana', 'cherry', 'apple'}
```

Get the Length of a Set

To determine how many items a set has, use the `len()` function.

Example

Get the number of items in a set:

```
thisset = {"apple", "banana", "cherry"}

print(len(thisset))
```

Set Items - Data Types

Set items can be of any data type:

Example

String, int and boolean data types:

```
set1 = {"apple", "banana", "cherry"}
set2 = {1, 5, 7, 9, 3}
set3 = {True, False, False}
```

A set can contain different data types:

Example

A set with strings, integers and boolean values:

```
set1 = {"abc", 34, True, 40, "male"}
```

type()

From Python's perspective, sets are defined as objects with the data type 'set':

```
<class 'set'>
```

Example

What is the data type of a set?

```
myset = {"apple", "banana", "cherry"}
print(type(myset))
```

Access Items

You cannot access items in a set by referring to an index or a key.

But you can loop through the set items using a `for` loop, or ask if a specified value is present in a set, by using the `in` keyword.

Example

Loop through the set, and print the values:

```
thisset = {"apple", "banana", "cherry"}

for x in thisset:
  print(x)
```

Example

Check if "banana" is present in the set:

```
thisset = {"apple", "banana", "cherry"}

print("banana" in thisset)
```

```
True
```

Add Items

Once a set is created, you cannot change its items, but you can add new items.

To add one item to a set use the `add()` method.

Example

Add an item to a set, using the `add()` method:

```
thisset = {"apple", "banana", "cherry"}

thisset.add("orange")

print(thisset)
```

Add Sets

To add items from another set into the current set, use the `update()` method.

Example

Add elements from `tropical` into `thisset`:

```
thisset = {"apple", "banana", "cherry"}
tropical = {"pineapple", "mango", "papaya"}

thisset.update(tropical)

print(thisset)
```

Remove Item

To remove an item in a set, use the `remove()`, or the `discard()` method.

Example

Remove "banana" by using the `remove()` method:

```
thisset = {"apple", "banana", "cherry"}

thisset.remove("banana")

print(thisset)
```

Loop Items

You can loop through the set items by using a `for` loop:

Example

Loop through the set, and print the values:

```
thisset = {"apple", "banana", "cherry"}

for x in thisset:
  print(x)
```

Python Dictionaries

Dictionaries are used to store data values in key:value pairs.

A dictionary is a collection which is ordered*, changeable and do not allow duplicates.

Dictionaries are written with curly brackets, and have keys and values:

Example

Create and print a dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict)
```

Dictionary Items

Dictionary items are ordered, changeable, and does not allow duplicates.

Dictionary items are presented in key:value pairs, and can be referred to by using the key name.

Example

Print the "brand" value of the dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict["brand"])
```

Duplicates Not Allowed

Dictionaries cannot have two items with the same key:

Example

Duplicate values will overwrite existing values:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964,
  "year": 2020
}
print(thisdict)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 2020}
```

Dictionary Length

To determine how many items a dictionary has, use the `len()` function:

Example

Print the number of items in the dictionary:

```
print(len(thisdict))
```

```
3
```

Dictionary Items - Data Types

The values in dictionary items can be of any data type:

Example

String, int, boolean, and list data types:

```
thisdict = {
  "brand": "Ford",
  "electric": False,
  "year": 1964,
  "colors": ["red", "white", "blue"]
}
```

```
{'brand': 'Ford', 'electric': False, 'year': 1964, 'colors': ['red', 'white', 'blue']}
```

Accessing Items

You can access the items of a dictionary by referring to its key name, inside square brackets:

Example

Get the value of the "model" key:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
x = thisdict["model"]

print(x)
```

```
Mustang
```

Get Keys

The `keys()` method will return a list of all the keys in the dictionary.

Example

Get a list of the keys:

```
x = thisdict.keys()
```

```
dict_keys(['brand', 'model', 'year'])
```

Change Values

You can change the value of a specific item by referring to its key name:

Example

Change the "year" to 2018:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict["year"] = 2018

print(thisdict)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 2018}
```

Adding Items

Adding an item to the dictionary is done by using a new index key and assigning a value to it:

Example

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
```

```
}
thisdict["color"] = "red"
print(thisdict)
```

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'color': 'red'}
```

Update Dictionary

The `update()` method will update the dictionary with the items from a given argument. If the item does not exist, the item will be added.

The argument must be a dictionary, or an iterable object with key:value pairs.

Example

Add a color item to the dictionary by using the `update()` method:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.update({"color": "red"})
```

Removing Items

There are several methods to remove items from a dictionary:

Example

The `pop()` method removes the item with the specified key name:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.pop("model")
print(thisdict)
```

```
{'brand': 'Ford', 'year': 1964}
```

Example

The `del` keyword can also delete the dictionary completely:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
```

Example

The `clear()` method empties the dictionary:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict.clear()
print(thisdict)
```

Loop Through a Dictionary

You can loop through a dictionary by using a `for` loop.

When looping through a dictionary, the return value are the *keys* of the dictionary, but there are methods to return the *values* as well.

Example

Print all key names in the dictionary, one by one:

```
for x in thisdict:
  print(x)
```

```
brand
model
year
```

Example

Print all *values* in the dictionary, one by one:

```
for x in thisdict:
  print(thisdict[x])
```

```
Ford
Mustang
1964
```

What is NumPy?

NumPy is a Python library used for working with arrays.

It also has functions for working in domain of linear algebra, fourier transform, and matrices.

NumPy was created in 2005 by Travis Oliphant. It is an open source project and you can use it freely.

NumPy stands for Numerical Python.

Why Use NumPy?

In Python we have lists that serve the purpose of arrays, but they are slow to process.

NumPy aims to provide an array object that is up to 50x faster than traditional Python lists.

The array object in NumPy is called `ndarray`, it provides a lot of supporting functions that make working with `ndarray` very easy.

Arrays are very frequently used in data science, where speed and resources are very important.

Create a NumPy ndarray Object

NumPy is used to work with arrays. The array object in NumPy is called `ndarray`.

We can create a NumPy `ndarray` object by using the `array()` function.

Example

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])

print(arr)

print(type(arr))
```

```
[1 2 3 4 5]
<class 'numpy.ndarray'>
```

Example

Use a tuple to create a NumPy array:

```
import numpy as np

arr = np.array((1, 2, 3, 4, 5))

print(arr)
```