Unit-1
Introduction to Python Programing

What is Python?

- Creator: Guido van Rossum
- Started: 1989, at Centrum Wiskunde & Informatica (CWI), Netherlands
- First Release: 1991
- Reason: Started as a hobby project during Christmas holidays
- Predecessor: Inspired by the ABC language, but Guido improved it by fixing its limitations

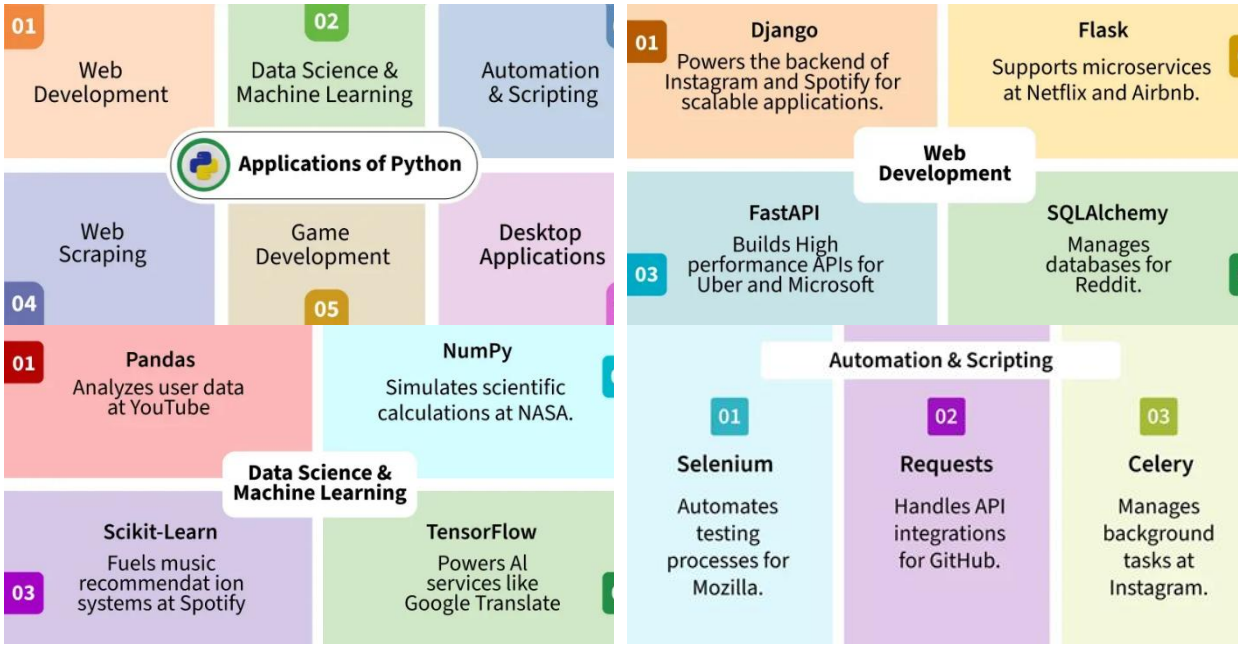
Why is it called Python?

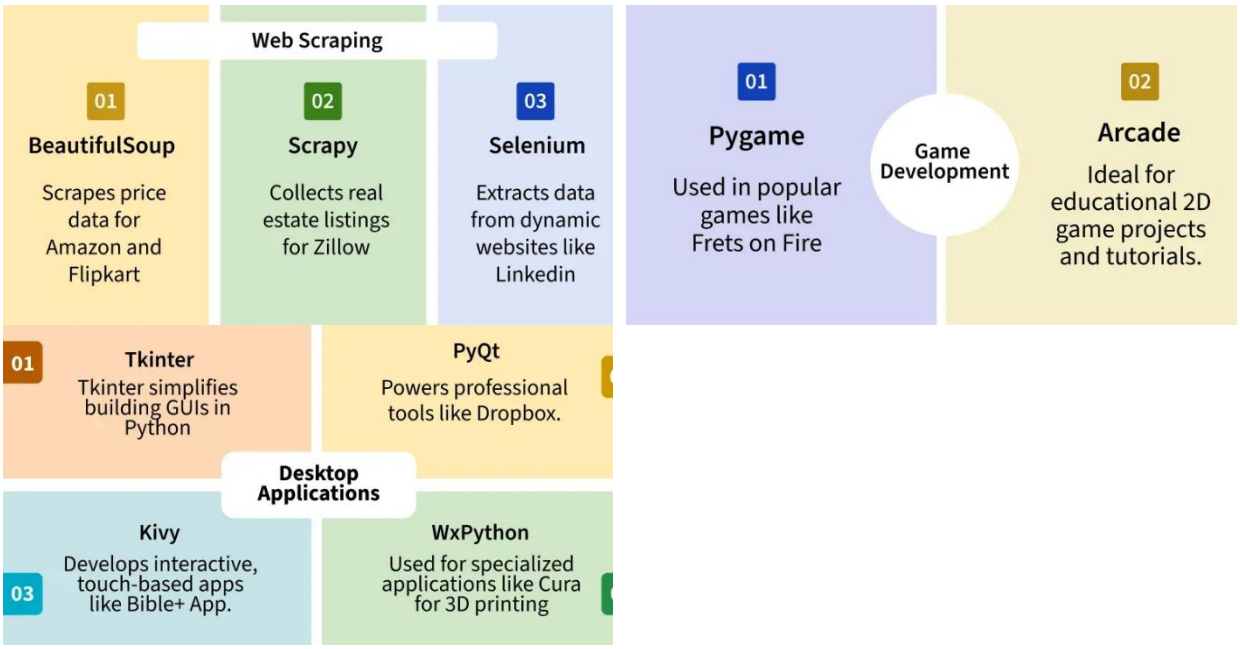
- Named after the BBC comedy series "Monty Python's Flying Circus"
- Guido wanted a name that was short, unique, and a little mysterious
- He served as Python's BDFL (Benevolent Dictator for Life) until 2018
- Worked at Google and later at Dropbox

Java VS Python

Parameters	**Python**	**Java**
Code	Python has generally fewer lines of code.	Java has long lines of code.
Framework	Compare to JAVA, Python has a lower number of Frameworks. Popular ones are Django and Flask.	Java has a large number of Frameworks. Popular ones are Spring, Hibernate, etc.
Syntax	The syntax is easy to remember almost similar to human language.	The syntax is complex as it throws errors if you miss semicolons or curly braces.
Key Features	Less line no of code, Rapid deployment, and dynamic typing.	Self-memory management, Robust, Platform independent
Speed	Python is slower since it uses an interpreter and also determines the data type at run time.	Java is faster in speed as compared to python.

Parameters	Python	Java
Databases	Python's database access layers are weaker than Java's JDBC. This is why it is rarely used in enterprises.	(JDBC)Java Database Connectivity is the most popular and widely used to connect with databases.
Machine Learning Libraries	Tensorflow, Pytorch	Weka, Mallet, Deeplearning4j, MOA
Practical Agility	Python has always had a presence in the agile space and has grown in popularity for many reasons, including the rise of the DevOps movement.	Java enjoys more consistent refactoring support than Python thanks on one hand to its static type system which makes automated refactoring more predictable and reliable, and on the other to the prevalence of IDEs in Java development.
Multiple Inheritance	Python supports multiple Inheritance.	Java partially supports Multiple Inheritance through interfaces.





Variables

- Variables are used to store data that can be referenced and manipulated during program execution. A variable is essentially a name that is assigned to a value.
- Unlike Java and many other languages, Python variables do not require explicit declaration of type.
- The type of the variable is inferred based on the value assigned.
- Rules for Python variables:
  - A variable name must start with a letter or the underscore character
  - A variable name cannot start with a number
  - A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
  - Variable names are case-sensitive (age, Age and AGE are three different variables)
  - A variable name cannot be any of the Python keywords.
- Variables do not need to be declared with any particular *type*, and can even change type after they have been set.

  Eg.

```
x = 4       # x is of type int
x = "Sally" # x is now of type str
print(x)
```

- If you want to specify the data type of a variable, this can be done with casting.

  Eg.

```
x = str(3)    # x will be '3'
y = int(3)    # y will be 3
z = float(3)  # z will be 3.0
```

- You can get the data type of a variable with the type() function.

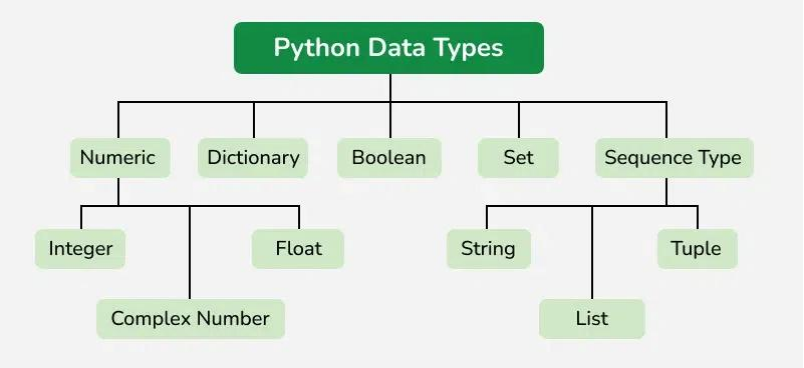
  Eg.

```
x = 5
y = "John"
```

```
print(type(x))
print(type(y))
```

- Python allows you to assign values to multiple variables in one line:
  Eg.
  ```
  x, y, z = "Orange", "Banana", "Cherry"
  print(x)
  print(y)
  print(z)
  ```
- And you can assign the *same* value to multiple variables in one line:
  Eg.
  ```
  x = y = z = "Orange"
  print(x)
  print(y)
  print(z)
  ```
- The print() function is often used to output variables.
  Eg.
  ```
  x = "Python is awesome"
  print(x)
  ```
- We can remove a variable from the namespace using the del keyword. This effectively deletes the variable and frees up the memory it was using.
  Eg.
  ```
  # Assigning value to variable
  x = 10
  print(x)
  # Removing the variable using del
  del x
  ```
- Using multiple assignments, we can swap the values of two variables without needing a temporary variable.
  Eg.
  ```
  a, b = 5, 10
  a, b = b, a
  print(a, b)
  ```

Data types:



The following are standard or built-in data types in Python:

- **Numeric:** int, float, complex
- **Sequence Type:** string, list, tuple
- **Mapping Type:** dict
- **Boolean:** bool
- **Set Type:** set, frozenset
- **Binary Types:** bytes, bytearray, memoryview

Example	Data Type
x = "Hello World"	str
x = 20	int
x = 20.5	float
x = 1j	complex
x = ["apple", "banana", "cherry"]	list
x = ("apple", "banana", "cherry")	tuple
x = range(6)	range
x = {"name" : "John", "age" : 36}	dict
x = {"apple", "banana", "cherry"}	set
x = frozenset({"apple", "banana", "cherry"})	frozenset
x = True	bool
x = b"Hello"	bytes
x = bytearray(5)	bytearray

x = memoryview(bytes(5))	memoryview
x = None	NoneType

Indentation
Python relies on indentation (whitespace at the beginning of a line) to define scope in the code. Other programming languages often use curly-brackets for this purpose.
Eg.
```
a = 33
b = 200
if b > a:
print("b is greater than a") # you will get an error
```

Conditionals Statements

- Conditional statements in Python are used to execute certain blocks of code based on specific conditions. These statements help control the flow of a program, making it behave differently in different situations.
- Python supports the usual logical conditions from mathematics:
  - Equals: a == b
  - Not Equals: a != b
  - Less than: a < b
  - Less than or equal to: a <= b
  - Greater than: a > b
  - Greater than or equal to: a >= b
- These conditions can be used in several ways, most commonly in "if statements" and loops.
- **If Conditional Statement**
  If statement is the simplest form of a conditional statement. It executes a block of code if the given condition is true.
  Eg.
  ```
  number = 15
  if number > 0:
    print("The number is positive")
  ```
- **Short Hand if**
  Short-hand if statement allows us to write a single-line if statement.
  Eg.
  ```
  age = 19
  if age > 18: print("Eligible to Vote.")
  ```
- **If else Conditional Statement**
  If Else allows us to specify a block of code that will execute if the condition(s) associated with an if or elif statement evaluates to False. Else block provides a way to handle all other cases that don't meet the specified conditions.
  Eg.
  ```
  number = 7
  if number % 2 == 0:
  ```

```
  print("The number is even")
else:
  print("The number is odd")
```

- **Short Hand if-else**
  The short-hand if-else statement allows us to write a single-line if-else statement. This method is also known as **ternary operator**. Ternary Operator essentially a shorthand for the if-else statement that allows us to write more compact and readable code, especially for simple conditions.
  Eg.
```
marks = 45
res = "Pass" if marks >= 40 else "Fail"
print(f"Result: {res}")
```
- **elif Statement**
  elif statement in Python stands for "else if." It allows us to check multiple conditions, providing a way to execute different blocks of code based on which condition is true. Using elif statements makes our code more readable and efficient by eliminating the need for multiple nested if statements.
  Eg.
```
age = 25

if age <= 12:
   print("Child.")
elif age <= 19:
   print("Teenager.")
elif age <= 35:
   print("Young adult.")
else:
   print("Adult.")
```
- **pass Statement**
  if statements cannot be empty, but if you for some reason have an if statement with no content, put in the pass statement to avoid getting an error. The pass statement is a null operation - nothing happens when it executes. It serves as a placeholder.
  Eg.
```
age = 16
if age < 18:
  pass
else:
  print("Access granted")
```

Loops

- Loops in Python are used to repeat actions efficiently. The main types are For loops (counting through items) and While loops (based on conditions).
- **While Loop**

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

Eg.

```
i = 1
while i < 6:
  print(i)
  i += 1
```

- **Infinite While Loop**

  If we want a block of code to execute infinite number of times then we can use the while loop in Python to do so.

  Eg.

  ```
  while (True):
     print("Hello Geek")
  ```

- **For Loop**

  For loops allow to execute a block of code repeatedly, once for each item in the sequence.

  Eg.

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

  - **Iterating over a sequence**

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

    Eg.

    ```
    fruits = ["apple", "banana", "cherry"]
    for x in fruits:
      print(x)
    ```

  - **Looping Through a String**

    Even strings are iterable objects, they contain a sequence of characters.

    Eg.

    ```
    for x in "apple":
      print(x)
    ```

- **Nested Loops**

  Python programming language allows to use one loop inside another loop which is called nested loop.

  Eg.

  ```
  for i in range(1, 5):
     for j in range(i):
        print(i, end=' ')
     print()
  ```

- **Loop Control Statements**

  Loop control statements change execution from their normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed. Python supports the following control statements.

  - **continue Statement**

    The continue statement in Python returns the control to the beginning of the loop.

Eg.

```
i = 0
while i < 6:
  i += 1
  if i == 3:
    continue
  print(i)
```

- **break Statement**
  With the break statement we can stop the loop even if the while condition is true.
  Eg.

```
i = 1
while i < 6:
  print(i)
  if i == 3:
    break
  i += 1
```

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

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

- **pass Statement**
  We use pass statement in Python to write empty loops. Pass is also used for empty control statements, functions and classes.
  Eg.

```
for letter in 'engineering':
    pass
print('Last Letter :', letter)
```

Working with Strings

- A string is a sequence of characters enclosed in quotes. It can include letters, numbers, symbols or spaces. Since Python has no separate character type, even a single character is treated as a string with length one. Strings are widely used for text handling and manipulation.
- Strings in python are surrounded by either single quotation marks, or double quotation marks.
- 'hello' is the same as "hello".
- We can display a string literal with the print() function.
  Eg.

```
print("Hello")
print('Hello')
```

- **Quotes Inside Quotes**
  We can use quotes inside a string, as long as they don't match the quotes surrounding the string.
  Eg.
```
print("It's alright")
print("He is called 'Johnny'")
print('He is called "Johnny"')
```
- **Assign String to a Variable**
  Assigning a string to a variable is done with the variable name followed by an equal sign and the string.
  Eg.
```
a = "Hello"
print(a)
```
- **Multiline Strings**
  We can assign a multiline string to a variable by using three quotes or three single quotes.
  Eg.
```
a = """Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua."""
print(a)
b = '''Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua.'''
print(b)
```
- **Accessing characters in String**
  Strings are indexed sequences. Positive indices start at **0** from the **left**; negative indices start at **-1** from the **right.**
  Eg.
```
s = "Engineering"
print(s[0])   # first character
print(s[4])   # 5th character
print(s[-10])   # 3rd character
print(s[-5])    # 5th character from end
```
- **String Slicing**
  Slicing is a way to extract a portion of a string by specifying the **start** and **end** indexes. The syntax for slicing is **string[start:end]**, where **start** starting index and **end** is stopping index (excluded).
  Eg.
```
s = "Engineering"
print(s[1:4])    # characters from index 1 to 3
print(s[:3])     # from start to index 2
print(s[3:])     # from index 3 to end
```

```
print(s[::-1])   # reverse string
print(s[-5:-2])
```

- **String Immutability**
  Strings are **immutable,** which means that they cannot be changed after they are created. If we need to manipulate strings then we can use methods like **concatenation, slicing** or **formatting** to create new strings based on original.
  Eg.
```
s = "engineering"
s = "E" + s[1:]   # create new string
print(s)
```
- **Modify Strings**
  - The upper() method returns the string in upper case.
  - The lower() method returns the string in lower case.
  - The strip() method removes any whitespace from the beginning or the end.
  - The replace() method replaces a string with another string.
  - The split() method splits the string into substrings if it finds instances of the separator.

  Eg.
```
a = "Hello, World!"
print(a.upper())
print(a.lower())
print(a.strip())
print(a.replace("H", "J"))
print(a.split(",")) # returns ['Hello', ' World!']
```
- **Deleting a String**
  In Python, it is not possible to delete individual characters from a string since strings are immutable. However, we can delete an entire string variable using the del keyword.
  Eg.
```
s = "engineering"
del s
```
- **String Concatenation**
  To concatenate, or combine, two strings you can use the + operator.
  Eg.
```
a = "Hello"
b = "World"
c = a + b
print(c)
```
- **Format Strings**
  - **F-Strings**
    F-String was introduced in Python 3.6, and is now the preferred way of formatting strings.
    To specify a string as an f-string, simply put an f in front of the string literal, and add curly brackets {} as placeholders for variables and other operations.
    Eg.

```
age = 36
txt = f"My name is John, I am {age}"
print(txt)
```

- **format() function**
  Another way to format strings is by using format() method.
  Eg.
  ```
  s = "My name is {} and I am {} years old.".format("Alice", 22)
  print(s)
  ```

- **String Membership Testing**
  in keyword checks if a particular substring is present in a string.
  Eg.
  ```
  s = "engineering"
  print("engineer" in s)
  print(“engineers" in s)
  ```

Working with Lists

In Python, a list is a built-in data structure that can hold an ordered collection of items. Unlike arrays in some languages, Python lists are very flexible:

- Can contain duplicate items
- **Mutable:** items can be modified, replaced, or removed
- **Ordered:** maintains the order in which items are added
- **Index-based:** items are accessed using their position (starting from 0)
- Can store mixed data types (integers, strings, booleans, even other lists)
  Eg.
  ```
  thislist = ["apple", "banana", "cherry"]
  print(thislist)
  ```
- **List Length**
  To determine how many items a list has, use the len() function.
  Eg.
  ```
  thislist = ["apple", "banana", "cherry"]
  print(len(thislist))
  ```
- A list can contain different data types.
  Eg.
  ```
  list1 = ["abc", 34, True, 40, "male"]
  ```
- From Python's perspective, lists are defined as objects with the data type 'list'.
  Eg.
  ```
  mylist = ["apple", "banana", "cherry"]
  print(type(mylist))
  ```
- **The list() Constructor**
  It is also possible to use the list() constructor when creating a new list.
  Eg.
  ```
  thislist = list(("apple", "banana", "cherry")) # note the double round-brackets
  print(thislist)
  ```
- **Access Items**
  List items are indexed and you can access them by referring to the index number

Eg.

```
thislist = ["apple", "banana", "cherry"]
print(thislist[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 list with the specified items.
  Eg.

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

- **Change Item Value**
  To change the value of a specific item, refer to the index number.
  Eg.

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

- **Change a Range of Item Values**
  To change the value of items within a specific range, define a list with the new values, and refer to the range of index numbers where you want to insert the new values.
  Eg.

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "mango"]
thislist[1:3] = ["blackcurrant", "watermelon"]
print(thislist)
```

- **Loop Through a List**
  You can loop through the list items by using a for loop.
  Eg.

```
thislist = ["apple", "banana", "cherry"]
for x in thislist:
  print(x)
```

- **List Comprehension**
  List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.
  Eg.

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = []
for x in fruits:
  if "a" in x:
    newlist.append(x)
print(newlist)
```

- **Sort List Alphanumerically**
  List objects have a sort() method that will sort the list alphanumerically, ascending, by default.
  Eg.

```
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
thislist.sort()
print(thislist)
```

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

Working with Sets

Python set is an unordered collection of multiple items having different datatypes. In Python, sets are mutable, unindexed and do not contain duplicates. The order of elements in a set is not preserved and can change.

- Can store None values.
- Implemented using hash tables internally.
- Do not implement interfaces like Serializable or Cloneable.
- Python sets are not inherently thread-safe; synchronization is needed if used across threads.
  Eg.
  ```
  thisset = {"apple", "banana", "cherry"}
  print(thisset)
  ```
- True and 1 is considered the same value.
  Eg.
  ```
  thisset = {"apple", "banana", "cherry", True, 1, 2}
  print(thisset)
  ```
- **Python frozenset**
  frozenset is an immutable version of a set.
  Like sets, it contains unique, unordered, unchangeable elements.
  Unlike sets, elements cannot be added or removed from a frozenset.
  Eg.

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

- **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.
  Eg.
```
thisset = {"apple", "banana", "cherry"}
for x in thisset:
  print(x)
```
- **Add Items**
  To add one item to a set use the add() method.
  Eg.
```
thisset = {"apple", "banana", "cherry"}
thisset.add("orange")
print(thisset)
```
- **Remove Item**
  To remove an item in a set, use the remove(), or the discard() method.
  Eg.
```
thisset = {"apple", "banana", "cherry"}
thisset.remove("banana")
print(thisset)
```

Method	Shortcut	Description
add()		Adds an element to the set
clear()		Removes all the elements from the set
copy()		Returns a copy of the set
difference()	-	Returns a set containing the difference between two or more sets
difference_update()	-=	Removes the items in this set that are also included in another, specified set
discard()		Remove the specified item
intersection()	&	Returns a set, that is the intersection of two other sets
intersection_update()	&=	Removes the items in this set that are not present in other, specified set(s)

isdisjoint()		Returns whether two sets have a intersection or not
issubset()	<=	Returns True if all items of this set is present in another set
	<	Returns True if all items of this set is present in another, *larger* set
issuperset()	>=	Returns True if all items of another set is present in this set
	>	Returns True if all items of another, *smaller* set is present in this set
pop()		Removes an element from the set
remove()		Removes the specified element
symmetric_difference()	^	Returns a set with the symmetric differences of two sets
symmetric_difference_update()	^=	Inserts the symmetric differences from this set and another
union()	|	Return a set containing the union of sets
update()	|=	Update the set with the union of this set and others

Working with Tuples

A tuple in Python is an immutable ordered collection of elements.

- Tuples are similar to lists, but unlike lists, they cannot be changed after their creation (i.e., they are immutable).
- Tuples can hold elements of different data types.
- The main characteristics of tuples are being ordered, heterogeneous and immutable.
  Eg.
  ```
  thistuple = ("apple", "banana", "cherry")
  print(thistuple)
  ```
- Tuples can contain elements of various data types, including other tuples, lists, dictionaries and even functions.
- **Access Tuple Items**
  We can access tuple items by referring to the index number, inside square brackets.
  Eg.
  ```
  thistuple = ("apple", "banana", "cherry")
  print(thistuple[1])
  ```

- **Range of Indexes**
  We can specify a range of indexes by specifying where to start and where to end the range.
  Eg.
  ```
  thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
  print(thistuple[2:5])
  ```
- **Add Items**
  Since tuples are immutable, they do not have a built-in append() method, but there are other ways to add items to a tuple.
  - **Convert into a list**: Just like the workaround for *changing* a tuple, we can convert it into a list, add our item(s), and convert it back into a tuple.
    Eg.
    ```
    thistuple = ("apple", "banana", "cherry")
    y = list(thistuple)
    y.append("orange")
    thistuple = tuple(y)
    ```
  - **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.
    Eg.
    ```
    thistuple = ("apple", "banana", "cherry")
    y = ("orange",)
    thistuple += y
    print(thistuple)
    ```
- **Unpacking a Tuple**
  When we create a tuple, we normally assign values to it. This is called "packing" a tuple.
  Eg.
  ```
  fruits = ("apple", "banana", "cherry")
  ```
  But, in Python, we are also allowed to extract the values back into variables. This is called "unpacking".
  Eg.
  ```
  fruits = ("apple", "banana", "cherry")
  (green, yellow, red) = fruits
  print(green)
  print(yellow)
  print(red)
  ```
- **Loop Through a Tuple**
  We can loop through the tuple items by using a for loop.
  Eg.
  ```
  thistuple = ("apple", "banana", "cherry")
  for x in thistuple:
    print(x)
  ```
- **Join Two Tuples**
  To join two or more tuples you can use the + operator
  Eg.

```
tuple1 = ("a", "b" , "c")
tuple2 = (1, 2, 3)
tuple3 = tuple1 + tuple2
print(tuple3)
```

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

Working with Dictionaries

Python dictionary is a data structure that stores the value in key: value pairs. Values in a dictionary can be of any data type and can be duplicated, whereas keys can't be repeated and must be immutable.

- Keys are case sensitive which means same name but different cases of Key will be treated distinctly.
- Keys must be immutable which means keys can be strings, numbers or tuples but not lists.
- Duplicate keys are not allowed and any duplicate key will overwrite the previous value.
- Internally uses hashing. Hence, operations like search, insert, delete can be performed in Constant Time.
- From Python 3.7 Version onward, Python dictionary are Ordered.

  Eg.
  ```
  thisdict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
  }
  print(thisdict)
  ```
- **Accessing Items**

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

  Eg.
  ```
  thisdict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
  }
  x = thisdict["model"]
  ```
  There is also a method called get() that will give you the same result.

  Eg.
  ```
  x = thisdict.get("model")
  ```
- **Get Keys**

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

Eg.

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

- **Change Values**

  We can change the value of a specific item by referring to its key name.

  Eg.

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

- **Update Dictionary**

  The update() method will update the dictionary with the items from the given argument.

  Eg.

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

- **Adding Items**

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

  Eg.

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

- **Removing Items**
  - The pop() method removes the item with the specified key name.

    Eg.

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

  - The popitem() method removes the last inserted item (in versions before 3.7, a random item is removed instead).

    Eg.

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

- The del keyword removes the item with the specified key name.

  Eg.

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

- The clear() method empties the dictionary.

  Eg.

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

- **Loop Through a Dictionary**

  We can loop through a dictionary by using a for loop.

  Eg.

```
for x in thisdict:
  print(x)
for x in thisdict:
  print(thisdict[x])
for x in thisdict.values():
  print(x)
for x in thisdict.keys():
  print(x)
for x, y in thisdict.items():
  print(x, y)
```

Method	Description
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and value

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