Data Types in Python

In programming languages, every value or data has an associated type to it known as data type. Some commonly used data types.

String: A String is a stream of characters enclosed within quotes.

"Hello World!"

1234

Integer: All the numbers (positive, negative and zero) without any fractional part come under Integers.

...-3, -2, -1, 0, 1, 2, 3,...

Float: Any number with a decimal point.

24.3, 345.210, -321.86

Boolean: In a general sense, anything that can take one of two possible values is considered a Boolean. As per the Python Syntax, True and False are considered as Boolean values.

True, False

Conditional Statements

Conditional Statement: Conditional Statement allows you to execute a block of code only when a specific condition is True.

```
if True:
   print("If Block")
   print("Inside If")
# Output is:
If Block
Inside If
```

If - Else Statement: When the If - Else conditional statement is used, the Else block of code executes if the condition is False.

```
a = int(input()) # -1
if a > 0:
   print("Positive")
else:
   print("Not Positive")
# Output is:
Not Positive
```

Nested Conditions: The conditional block inside another if/else conditional block is called as a nested conditional block.

```
if Condition A:
    if Condition B:
        block of code
else:
    block of code
```

```
if Condition A:
    block of code
else:
    if Condition B:
        block of code
```

Elif Statement: Use the elif statement to have multiple conditional statements between if and else. The elif statement is optional.

```
if Condition A:
    block of code
elif Condition B:
    block of code
else:
    block of code
```

Identation:

1. Space(s) in front of the conditional block is called indentation. 2. Indentation(spacing) is used to identify the Conditional Blocks. 3. Standard practice is to use four spaces for indentation.

Strings - working with strings

String Concatenation: Joining strings together is called string concatenation.

```
a = "Hello" + " " + "World"
print(a) # Hello World
```

String Repetition: * operator is used for repeating strings any number of times as required.

```
a = "$" * 10
print(a) # $$$$$$$$$$
```

Length of String: len() returns the number of characters in a given string.

```
username = input() # Ravi
length = len(username)
print(length) # 4
```

String Indexing: We can access an individual character in a string using their positions (which start from 0) . These positions are also called *index*.

```
username = "Ravi"
first_letter = username[0]
print(first_letter) # R
```

String Slicing: Obtaining a part of a string is called string slicing. Start from the *start_index* and stops at the *end_index*. (end_index is not included in the slice).

```
message = "Hi Ravi"
part = message[3:7]
print(part) # Ravi
```

Slicing to End: If *end_index* is not specified, slicing stops at the end of the string.

```
message = "Hi Ravi"
part = message[3:]
print(part) # Ravi
```

Slicing from Start: If the *start_index* is not specified, the slicing starts from the index 0.

```
message = "Hi Ravi"
part = message[:2]
print(part) # Hi
```

Negative Indexing: Use negative indexes to start the slice from the end of the string.

```
b = "Hello, World!"
print(b[-5:-2]) # orl
```

Reversing String: Reverse the given string using the extended slice operator.

```
txt = "Hello World"
txt = txt[::-1]
print(txt) # dlroW olleH
```

Membership check-in strings:

in: By using the in operator, one can determine if a value is present in a sequence or not.

```
language = "Python"
result = "P" in language
```

```
print(result) # True
```

not in: By using the, not in operator, one can determine if a value is not present in a sequence or not.

```
language = "Python"
result = "P" not in language
print(result) # False
```

Calculations in Python

Addition: Addition is denoted by + sign.

```
print(2 + 5)  # 7
print(1 + 1.5) # 2.5
```

Subtraction: Subtraction is denoted by - sign.

```
print(5 - 2) # 3
```

Multiplication: Multiplication is denoted by * sign.

```
print(2 * 5) # 10
print(5 * 0.5) # 2.5
```

Division: Division is denoted by / sign.

```
print(80 / 5) # 16.0
```

Modulus: To find the remainder, we use the Modulus operator %.

```
print(7 % 2) # 1
```

Exponent: To find a power b, we use Exponent Operator **.

```
print(7 ** 2) # 49
```

Floor division: To find an integral part of the quotient we use Floor Division Operator //.

```
print(13 // 5) # 2
```

Input and Output Basics

Take Input From User: input() allows flexibility to take input from the user. Reads a line of input as a string.

```
username = input() # Ajay
```

Printing the Output: print() function prints the message to the screen or any other standard output device.

```
print(username) # Ajay
```

Comments: Comment starts with a hash # . It can be written in its own line next to a statement of code.

```
# This is a comment
```

String Methods

Name	Syntax	Usage
isdigit()	str.isdigit()	Gives True if all the characters are digits. Otherwise, False.
strip()	str.strip()	Removes all the leading and trailing spaces from a string.
strip() with separator	str.strip(separator)	We can also specify separator(string) that need to be removed.
replace()	str.replace(old, new)	Gives a new string after replacing all the occurrences of the old substring with the new substring.
startswith()	str_var.startswith(value)	Gives True if the string starts with the specified value. Otherwise, False.
endswith()	str.endswith(value)	Gives True if the string ends with the specified value. Otherwise, False.
upper()	str.upper()	Gives a new string by converting each character of the given string to uppercase.
lower()	str.lower()	Gives a new string by converting each character of the given string to lowercase.
split()	str.split()	The split() method splits a string into a list.
split() with separator	str.split(separator, maxsplit)	Specifies the separator to use when splitting the string. By default any whitespace is a separator.
join()	str.join(iterable)	The join() method takes all items in an iterable and joins them into one string.

String Formatting: String Formatting simplifies the concatenation. It increases the readability of code and type conversion is not required.

Add Placeholders: Add placeholders {} where the string needs to be formatted.

```
name = "Raju"

age = 10

msg = "Hi {}. You are {} years old."

print(msg.format(name, age)) # Hi Raju. You are 10 years old.
```

Numbering Placeholders: Numbering placeholders, will fill values according to the position of arguments.

```
name = input() # Raju

age = int(input()) # 10

msg = "Hi {1}. You are {0} years old."

print(msg.format(name, age)) # Hi 10. You are Raju years old.
```

Naming Placeholder: Naming placeholders will fill values according to the keyword arguments.

```
name = input() # Raju

age = int(input()) # 10

msg = "Hi {name}. You are {age} years old."

print(msg.format(age=age, name=name)) # Hi Raju. You are 10 years old.
```

Relational & Logical Operators

Relational Operators are used to comparing values. Gives True or False as the result of a comparison.

Operator	**Name**	**Example**	**Output**
>	Is greater than	print(2 > 1)	True
<	Is less than	print(5 < 10)	True
==	Is equal to	print(3 == 4)	False
<=	Is less than or equal to	print(2 <= 1)	False
>=	Is greater than or equal to	print(2 >= 1)	True
!=	Is not equal to	print(2 != 1)	True

Logical operators are used to performing logical operations on Boolean values. Gives True or False as a result.

Name	**Code**	**Output**
and	print((5 < 10) and (1 < 2))	True
or	print((5 < 10) or (2 < 2))	True
not	print(not (2 < 3))	False

Logical Operators Truth Table:

A	**B**	**A and B**

A	B	A and B
True	True	True
True	False	False
False	False	False
False	True	False

A	B	A or B
True	True	True
True	False	True
False	False	False
False	True	True

A	Not A
True	False
False	True

Loops

Loops: Loops allow us to execute a block of code several times.

While Loop: Allows us to execute a block of code several times as long as the condition is True.

```
a = 1
while a < 3:
  a = a + 1
  print(a)
# Output is:
2
3
```

For Loop: for statement iterates over each item of a sequence.

Syntax:

```
for each_item in sequence:
  block of code
```

Range: Generates a sequence of integers starting from 0. Stops before n (n is not included).

Syntax: range(n)

```
for number in range(3):
  print(number)
# Output is:
0
1
2
```

Range with Start and End: Generates a sequence of numbers starting from the start. Stops before the end (the end is not included). **Syntax**: range(start, end)

```
for number in range(5, 8):
  print(number)
# Output is:
5
6
7
```

Lists - Working with Lists

List: List is the most versatile python data structure. Holds an ordered sequence of items.

Accessing List Items: To access elements of a list, we use Indexing.

```
list_a = [5, "Six", 2, 8.2]

print(list_a[1]) # Six
```

Iterating Over a List:

```
list_a = [5, "Six", 8.2]

for item in list_a:

  print(item)

# Output is:

5

Six

8.2
```

List Concatenation: Similar to strings, + operator concatenates lists.

```
list_a = [1, 2]

list_b = ["a", "b"]

list_c = list_a + list_b

print(list_c) # [1, 2, 'a', 'b']
```

List Slicing: Obtaining a part of a list is called List Slicing.

```
list_a = [5, "Six", 2]

list_b = list_a[:2]

print(list_b) # [5, 'Six']
```

Extended Slicing: Similar to string extended slicing, we can extract alternate items using the step.

```
list_a = ["R", "B", "G", "O", "W"]

list_b = list_a[0:5:3]

print(list_b) # ['R', 'O']
```

Reversing a List: -1 for step will reverse the order of items in the list.

```
list_a = [5, 4, 3, 2, 1]

list_b = list_a[::-1]

print(list_b) # [1, 2, 3, 4, 5]
```

Slicing With Negative Index: You can also specify negative indices while slicing a List.

```
list_a = [5, 4, 3, 2, 1]
```

list_b = list_a[-3:-1]

print(list_b) *# [3, 2]*

Negative Step Size: Negative Step determines the decrement between each index for slicing. The start index should be greater than the end index in this case

list_a = [5, 4, 3, 2, 1]

list_b = list_a[4:2:-1]

print(list_b) *# [1, 2]*

Membership check-in lists:

Name	**Usage**
in	By using the in operator, one can determine if a value is present in a sequence or not.
not in	By using the, not in operator, one can determine if a value is not present in a sequence or not.

Nested Lists: A list as an item of another list.

Accessing Nested List:

list_a = [5, "Six", [8, 6], 8.2]

print(list_a[2]) *# [8, 6]*

Accessing Items of Nested List:

list_a = [5, "Six", [8, 6], 8.2]

print(list_a[2][0]) *# 8*

List Methods

Name	**Syntax**	**Usage**
append()	list.append(value)	Adds an element to the end of the list.
extend()	list_a.extend(list_b)	Adds all the elements of a sequence to the end of the list.
insert()	list.insert(index,value)	Element is inserted to the list at specified index.
pop()	list.pop()	Removes last element.
remove()	list.remove(value)	Removes the first matching element from the list.
clear()	list.clear()	Removes all the items from the list.
index()	list.index(value)	Returns the index at the first occurrence of the specified value.

Name	Syntax	Usage
count()	list.count(value)	Returns the number of elements with the specified value.
sort()	list.sort()	Sorts the list.
copy()	list.copy()	Returns a new list. It doesn't modify the original list.

Functions

Functions: Block of reusable code to perform a specific action.

Defining a Function: Function is uniquely identified by the function_name.

```
def function_name():
   reusable code
```

Calling a Function: The functional block of code is executed only when the function is called.

```
def function_name():
   reusable code
function_name()
def sum_of_two_number(a, b):
   print(a + b) # 5


sum_of_two_number(2, 3)
```

Function With Arguments: We can pass values to a function using an Argument.

```
def function_name(args):
   reusable code
function_name(args)
```

Returning a Value: To return a value from the function use return keyword. Exits from the function when return statement is executed.

```
def function_name(args):
   block of code
   return msg


function_name(args)
def sum_of_two_number(a, b):
   total = a + b
```

```
    return total


result = sum_of_two_number(2, 3)
print(result) # 5
```

Function Arguments: A function can have more than one argument.

```
def function_name(arg_1, arg_2):
    reusable code


function_name(arg_1, arg_2)
```

Keyword Arguments: Passing values by their names.

```
def greet(arg_1, arg_2):
    print(arg_1 + " " + arg_2) # Good Morning Ram


greet(arg_1="Good Morning", arg_2="Ram")
```

Positional Arguments: Values can be passed without using argument names. These values get assigned according to their position. Order of the arguments matters here.

```
def greet(arg_1, arg_2):
    print(arg_1 + " " + arg_2) # Good Morning Ram


greeting = input() # Good Morning
name = input() # Ram
greet(greeting, name)
```

Default Values:

```
def greet(arg_1="Hi", arg_2="Ram"):
    print(arg_1 + " " + arg_2) # Hi Ram


greeting = input() # Hello
name = input() # Teja
greet()
```

Arbitrary Function Arguments: We can define a function to receive any number of arguments.

Variable Length Arguments: Variable length arguments are packed as tuple.

```
def more_args(*args):
  print(args) # (1, 2, 3, 4)


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

Unpacking as Arguments: If we already have the data required to pass to a function as a sequence, we can unpack it with * while passing.

```
def greet(arg1="Hi", arg2="Ram"):
    print(arg1 + " " + arg2) # Hello Teja


data = ["Hello", "Teja"]
greet(*data)
```

Multiple Keyword Arguments: We can define a function to receive any number of keyword arguments. Variable length kwargs are packed as dictionary.

```
def more_args(**kwargs):
    print(kwargs)  # {'a': 1, 'b': 2}


more_args(a=1, b=2)
```

Function Call Stack: Stack is a data structure that stores items in an Last-In/First-Out manner. Function Call Stack keeps track of function calls in progress.

```
def function_1():
    pass


def function_2():
    function_1()
```

Recursion: A function calling itself is called a Recursion.

```
def function_1():
    block of code
    function_1()
```

Passing Immutable Objects:

```
def increment(a):
    a += 1
```

```
a = int(input()) # 5
increment(a)
print(a) # 5
```

- Even though variable names are same, they are referring to two different objects.
- Changing the value of the variable inside the function will not affect the variable outside.

Passing Mutable Objects:

```
def add_item(list_x):
   list_x += [3]


list_a = [1,2]
add_item(list_a)
print(list_a) # [1, 2, 3]
```

- The same object in the memory is referred by both list_a and list_x

```
def add_item(list_x=[]):
   list_x += [3]
   print(list_x)


add_item()
add_item([1,2])
add_item()
# Output is:
[3]
[1, 2, 3]
[3, 3]
```

- Default args are evaluated only once when the function is defined, not each time the function is called.

Nested Loops

Nested Loops: An inner loop within the repeating block of an outer loop is called a Nested Loop. The Inner Loop will be executed one time for each iteration of the Outer Loop.

Syntax:

```
for item in sequence A:
   Block 1
```

```
    for item in sequence B:
        Block 2
```

Syntax of while in for loop:

```
for item in sequence:
    Block 1
    while Condition:
        Block 2
```

Syntax of for in while loop:

```
while Condition:
    Block 1
    for item in sequence:
        Block 2
```

Loop Control Statements:

Name	**Usage**
Break	break statement makes the program exit a loop early.
Continue	continue is used to skip the remaining statements in the current iteration when a condition is satisfied.
Pass	pass statement is used as a syntactic placeholder. When it is executed, nothing happens.
Break (In Nested Loop)	break in the inner loop stops the execution of the inner loop.

Built - In - Functions

Name	**Usage**
print()	Function prints the message to the screen or any other standard output device.
int()	Converts valid data of any type to integer.
str()	Converts data of any type to a string.
id()	To find the id of a object.
round(number, digits(optional))	Rounds the float value to the given number of decimal digits.
bool()	Converts to boolean data type.
ord(character)	Gives unicode value of the character.
chr(unicode)	Gives character with the unicode value.
list(sequence)	Takes a sequence and converts it into list.
tuple(sequence)	Takes a sequence and converts it into tuple.
set(sequence)	Takes any sequence as argument and converts to set, avoiding duplicates.
dict(sequence)	Takes any number of key-value pairs and converts to dictionary.
float()	Converts to float data type.
type()	Check the datatype of the variable or value using.
min()	Returns the smallest item in a sequence or the smallest of two or more arguments.
max()	Returns the largest item in a sequence or the largest of two or more arguments.
sum(sequence)	Returns the sum of items in a sequence.
sorted(sequence)	Returns a new sequence with all the items in the given sequence ordered in increasing order.

Name	**Usage**
sorted(sequence, reverse=True)	Returns a new sequence with all the items in the given sequence ordered in decreasing order.
len(sequence)	Returns the length of the sequence.
map()	Applies a given function to each item of a sequence (list, tuple etc.) and returns a sequence of the results.
filter()	Method filters the given sequence with the help of a function that tests each element in the sequence to be true or not.
reduce()	Receives two arguments, a function and an iterable. However, it doesn't return another iterable, instead, it returns a single value.

Floating Point Approximation: Float values are stored approximately.

```
print(0.1 + 0.2) # 0.30000000000000004
```

Floating Point Errors: Sometimes, floating point approximation gives unexpected results.

```
print((0.1 + 0.2) == 0.3) # False
```

Different compound assignment operators are: +=, -=, *=, /=, %=

```
a = 10

a += 1

print(a) # 11

a = 10

a -= 2

print(a) # 8

a = 10

a /= 2

print(a) # 5.0

a = 10

a %= 2

print(a) # 0
```

Single And Double Quotes: A string is a sequence of characters enclosed within quotes.

```
sport = 'Cricket'

sport = "Cricket"
```

Escape Characters: Escape Characters are a sequence of characters in a string that is interpreted differently by the computer. We use escape characters to insert characters that are illegal in a string.

```
print("Hello\nWorld")

# Output is:

Hello

World
```

We got a new line by adding \n escape character.

Name	Usage
\n	New Line
\t	Tab Space
\\	Backslash
\'	Single Quote
\"	Double Quote

Set Methods, Operations and Comparisons

Set Methods:

Name	Syntax	Usage
add()	set.add(value)	Adds the item to the set, if the item is not present already.
update()	set.update(sequence)	Adds multiple items to the set, and duplicates are avoided.
discard()	set.discard(value)	Takes a single value and removes if present.
remove()	set_a.remove(value)	Takes a value and removes it if it is present or raises an error.
clear()	set.clear()	Removes all the items in the set.

Set Operations:

Union: Union of two sets is a set containing all elements of both sets. **Syntax**: set_a | set_b (or) set_a.union(sequence)

```
set_a = {4, 2, 8}

set_b = {1, 2}

union = set_a | set_b

print(union) # {1, 2, 4, 8}
```

Intersection: The intersection of two sets is a set containing common elements of both sets. **Syntax**: set_a & set_b (or) set_a.intersection(sequence)

```
set_a = {4, 2, 8}

set_b = {1, 2}

intersection = set_a & set_b

print(intersection) # {2}
```

Difference: The difference of two sets is a set containing all the elements in the first set but not the second.

Syntax: set_a - set_b (or) set_a.difference(sequence)

```
set_a = {4, 2, 8}

set_b = {1, 2}

diff = set_a - set_b

print(diff) # {8, 4}
```

Symmetric Difference: Symmetric difference of two sets is a set containing all elements which are not common to both sets. **Syntax**: set_a ^ set_b (or) set_a.symmetric_difference(sequence)

```
set_a = {4, 2, 8}

set_b = {1, 2}

symmetric_diff = set_a ^ set_b

print(symmetric_diff) # {8, 1, 4}
```

Set Comparisons: Set comparisons are used to validate whether one set fully exists within another.

issubset(): set2.issubset(set1) Returns True if all elements of the second set are in the first set. Else, False.

issuperset(): set1.issuperset(set2) Returns True if all elements of second set are in first set. Else, False.

isdisjoint(): set1.isdisjoint(set2) Returns True when they have no common elements. Else, False.

Tuples

Tuple: Holds an ordered sequence of items. Tuple is an immutable object, whereas a list is a mutable object.

```
tuple_a = (5, "Six", 2, 8.2)
```

Accessing Tuple Elements: Accessing Tuple elements is also similar to string and list accessing and slicing.

```
tuple_a = (5, "Six", 2, 8.2)

print(tuple_a[1]) # Six
```

Tuple Slicing: The slice operator allows you to specify where to begin slicing, where to stop slicing, and what step to take. Tuple slicing creates a new tuple from an old one.

```
tuple= ('a','b','c','d','e','f','g','h','i','j')

print(tuple[0:2]) # ('a', 'b')

print(tuple[-1:-3:-2]) # ('j',)

print(tuple[1:7:2]) # ('b', 'd', 'f')
```

Membership Check: Check if the given data element is part of a sequence or not.Membership Operators in and not in.

```
tuple_a = (1, 2, 3, 4)

is_part = 5 in tuple_a

print(is_part) # False

tuple_a = (1, 2, 3, 4)

is_part = 5 not in tuple_a

print(is_part) # True
```

Tuple Packing: () brackets are optional while creating tuples. In Tuple Packing, Values separated by commas will be packed into a tuple.

```
a = 1, 2, 3

print(type(a))

print(a)

# Output is:

<class 'tuple'>

(1, 2, 3)
```

Unpacking: Values of any sequence can be directly assigned to variables. Number of variables in the left should match the length of the sequence.

```
tuple_a = ('R', 'e', 'd')

(s_1, s_2, s_3) = tuple_a

print(s_1, s_2, s_3) # R e d
```

Dictionaries

Dictionaries: Unordered collection of items. Every dictionary item is a Key-value pair.

Creating a Dictionary: Created by enclosing items within {curly} brackets. Each item in the dictionary has a key-value pair separated by a comma.

```
dict_a = {

  "name": "Teja",
  "age": 15
}
```

Immutable Keys: Keys must be of an immutable type and must be unique. Values can be of any data type and can repeat.

Accessing Items:To access the items in dictionary, we use square bracket [ ] along with the key to obtain its value.

```
dict_a = {
  'name': 'Teja',
  'age': 15
}
print(dict_a['name']) # Teja
```

Accessing Items - Get: The get() method returns None if the key is not found.

```
dict_a = {
  'name': 'Teja',
  'age': 15
}
print(dict_a.get('name')) # Teja
print(dict_a.get('city')) # None
```

Membership Check: Checks if the given key exists.

```
dict_a = {
  'name': 'Teja',
  'age': 15
}
result = 'name' in dict_a
print(result) # True
```

Adding a Key-Value Pair:

```
dict_a = {'name': 'Teja','age': 15 }
dict_a['city'] = 'Goa'
print(dict_a) # {'name': 'Teja', 'age': 15, 'city': 'Goa'}
```

Modifying an Existing Item: As dictionaries are mutable, we can modify the values of the keys.

```
dict_a = {
   'name': 'Teja',
   'age': 15
}
dict_a['age'] = 24
print(dict_a) # {'name': 'Teja', 'age': 24}
```

Deleting an Existing Item: We can also use the del keyword to remove individual items or the entire dictionary itself.

```
dict_a = {
  'name': 'Teja',
  'age': 15
}
del dict_a['age']
print(dict_a) # {'name': 'Teja'}
```

Sets

Sets: Unordered collection of items. Every set element is Unique (no duplicates) and Must be immutable.

No Duplicate Items: Sets contain unique elements

```
set_a = {"a", "b", "c", "a"}
print(set_a) # {'b', 'a', 'c'}
```

Immutable Items: Set items must be immutable. As List is mutable, Set cannot have list as an item.

```
set_a = {"a", ["c", "a"]}
print(set_a)  # TypeError: unhashable type: 'list'
```

Dictionary Views & Methods

Dictionary Views:

View	**Syntax**	**Usage**
keys	dict.keys()	Returns dictionary Keys.
Values	dict.values()	Returns dictionary Values.
items	dict.items()	Returns dictionary items(key-value) pairs.

Dictionary Methods:

Name	**Syntax**	**Usage**
copy	dict.copy()	Returns copy of a dictionary.

Name	**Syntax**	**Usage**
update	dict.update(iterable)	Inserts the specified items to the dictionary.
clear	dict.clear()	Removes all the elements from a dictionary.

Scopes & NameSpaces

Object: In general, anything that can be assigned to a variable in Python is referred to as an object. Strings, Integers, Floats, Lists, Functions, Module etc. are all objects.

Namespaces: A namespace is a collection of currently defined names along with information about the object that the name references. It ensures that names are unique and won’t lead to any conflict.

```
def greet_1():
    a = "Hello"
    print(a)
    print(id(a))


def greet_2():
    a = "Hey"
    print(a)
    print(id(a))


print("Namespace - 1")
greet_1()
print("Namespace - 2")
greet_2()
# Output is:
```

```
Namespace - 1
Hello
140639382368176
Namespace - 2
Hey
140639382570608
```

Types of namespaces:

Built-in Namespace: Created when we start executing a Python program and exists as long as the program is running. This is the reason that built-in functions like id(), print() etc. are always available to us from any part of the program.

Global Namespace: This namespace includes all names defined directly in a module (outside of all functions). It is created when the module is loaded, and it lasts until the program ends.

Local Namespace: Modules can have various functions and classes. A new local namespace is created when a function is called, which lasts until the function returns.

Scope of a Name:

In Python, the scope of a name refers to where it can be used. The name is searched for in the local, global, and built-in namespaces in that order.

Global variables: In Python, a variable defined outside of all functions is known as a global variable. This variable name will be part of Global Namespace.

```
x = "Global Variable"
print(x) # Global Variable


def foo():
  print(x) # Global Variable


foo()
```

Local Variables: In Python, a variable defined inside a function is a local variable. This variable name will be part of the Local Namespace.

```
def foo():
  x = "Local Variable"
  print(x) # Local Variable


foo()
print(x) # NameError: name 'x' is not defined
```

Local Variables & Global Variables:

```
x = "Global Variable"


def foo():
   x = "Local Variable"
   print(x)


print(x)
foo()
print(x)
```

```
# Output is:
Global Variable
Local Variable
Global Variable
```

Modifying Global Variables: global keyword is used to define a name to refer to the value in Global Namespace.

```
x = "Global Variable"

def foo():
   global x
   x = "Global Change"
   print(x)

print(x)
foo()
print(x)
# Output is:
Global Variable
Global Change
Global Change
```

Python Standard Library

The collection of predefined utilities is referred as the Python Standard Library. All these functionalities are organized into different modules.

Module: In Python context, any file containing Python code is called a module.

Package: These modules are further organized into folders known as packages.

Importing module: To use a functionality defined in a module we need to import that module in our program.

```
import module_name
```

Importing from a Module: We can import just a specific definition from a module.

```
from math import factorial
print(factorial(5)) # 120
```

Aliasing Imports: We can also import a specific definition from a module and alias it.

```
from math import factorial as fact

print(fact(5)) # 120
```

Random module: Randomness is useful in whenever uncertainty is required.
Example: Rolling a dice, flipping a coin, etc,.

random module provides us utilities to create randomness.

Randint: randint() is a function in random module which returns a random integer in the given interval.

```
import random

random_integer = random.randint(1, 10)

print(random_integer) # 8
```

Choice: choice() is a function in random module which returns a random element from the sequence.

```
import random

random_ele = random.choice(["A","B","C"])

print(random_ele) # B
```

Classes

Classes: Classes can be used to bundle related attributes and methods. To create a class, use the keyword class

```
class className:

    attributes

    methods
```

Self: self passed to method contains the object, which is an instance of class.

Special Method: In Python, a special method __init__ is used to assign values to attributes.

```
class Mobile:

    def __init__(self, model):

        self.model = model
```

Instance of Class: Syntax for creating an instance of class looks similar to function call. An instance of class is an Object.

```
class Mobile:

    def __init__(self, model):

        self.model = model


mobile_obj = Mobile("iPhone 12 Pro")
```

Class Object: An object is simply a collection of attributes and methods that act on those data.

```
class Mobile:
  def __init__(self, model):
    self.model = model
  def make_call(self,number):
    return "calling..{}".format(number)
```

Attributes of an Object: Attributes can be set or accessed using . (dot) character.

```
class Mobile:
  def __init__(self, model):
    self.model = model


obj = Mobile("iPhone 12 Pro")
print(obj.model) # iPhone 12 Pro
```

Accessing in Other Methods: We can also access and update attributes in other methods.

```
class Mobile:
  def __init__(self, model):
    self.model = model


  def get_model(self):
    print(self.model) # iPhone 12 Pro


obj_1 = Mobile("iPhone 12 Pro")
obj_1.get_model()
```

Updating Attributes: It is recommended to update attributes through methods.

```
class Mobile:
  def __init__(self, model):
    self.model = model


  def update_model(self, model):
    self.model = model
```

```
obj_1 = Mobile("iPhone 12")

obj_1.update_model("iPhone 12 Pro")

print(obj_1.model) # iPhone 12 Pro
```

Instance Attributes: Attributes whose value can differ for each instance of class are modelled as instance attributes.

Accessing Instance Attributes: Instance attributes can only be accessed using instance of class.

```
class Cart:

  def __init__(self):

      self.items = {'book': 3}

  def display_items(self):

      print(self.items)  # {'book': 3}


a = Cart()

a.display_items()
```

Class Attributes: Attributes whose values stay common for all the objects are modelled as Class Attributes.

Accessing Class Attributes:

```
class Cart:

   flat_discount = 0

   min_bill = 100

   def __init__(self):

       self.items = {}


print(Cart.min_bill) # 100
```

Updating Class Attribute:

```
class Cart:

   flat_discount = 0

   min_bill = 100

   def print_min_bill(self):

       print(Cart.min_bill) # 200

a = Cart()

b = Cart()
```

```
Cart.min_bill = 200
b.print_min_bill()
```

Methods: Broadly, methods can be categorized as

- Instance Methods
- Class Methods
- Static Methods

Instance Methods: Instance methods can access all attributes of the instance and have self as a parameter.

```
class Cart:
  def __init__(self):
    self.items = {}
  def add_item(self, item_name,quantity):
    self.items[item_name] = quantity
  def display_items(self):
    print(self.items) # {'book': 3}

a = Cart()
a.add_item("book", 3)
a.display_items()
```

Class Methods: Methods which need access to class attributes but not instance attributes are marked as Class Methods. For class methods, we send cls as a parameter indicating we are passing the class.

```
class Cart:
  flat_discount = 0
  @classmethod
  def update_flat_discount(cls, new_flat_discount):
    cls.flat_discount = new_flat_discount

Cart.update_flat_discount(25)
print(Cart.flat_discount) # 25
```

Static Method: Usually, static methods are used to create utility functions which make more sense to be part of the class. @staticmethod decorator marks the method below it as a static method.

```
class Cart:
    @staticmethod
    def greet():
        print("Have a Great Shopping") # Have a Great Shopping


Cart.greet()
```

Instance Methods	**Class Methods**	**Methods**
self as parameter	cls as parameter	No cls or self as parameters
No decorator required	Need decorator @classmethod	Need decorator @staticmethod
Can be accessed through object(instance of class)	Can be accessed through class	Can be accessed through class

OOPS

OOPS: Object-Oriented Programming System (OOPS) is a way of approaching, designing, developing software that is easy to change.

Bundling Data: While modeling real-life objects with object oriented programming, we ensure to bundle related information together to clearly separate information of different objects.

Encapsulation: Bundling of related properties and actions together is called Encapsulation. Classes can be used to bundle related attributes and methods.

Inheritance: Inheritance is a mechanism by which a class inherits attributes and methods from another class. Prefer modeling with inheritance when the classes have an IS-A relationship.

```
class Product:
    def __init__(self, name):
        self.name = name


    def display_product_details(self):
        print("Product: {}".format(self.name)) # Product: TV



class ElectronicItem(Product):
    pass
```

```
e = ElectronicItem("TV")
e.display_product_details()
```

Super Class & Sub Class:

- Superclass cannot access the methods and attributes of the subclass.
- The subclass automatically inherits all the attributes & methods from its superclass.

```
class Product:
  def __init__(self, name):
     self.name = name
  def display_product_details(self):
     print("Product: {}".format(self.name)) # Product: TV


class ElectronicItem(Product):
   def set_warranty(self, warranty_in_months):
      self.warranty_in_months = warranty_in_months


e = ElectronicItem("TV")
e.display_product_details()
```

Calling Super Class Method: We can call methods defined in the superclass from the methods in the subclass.

```
class Product:
   def __init__(self, name):
     self.name = name
   def display_product_details(self):
     print("Product: {}".format(self.name)) # Product: TV


class ElectronicItem(Product):
   def set_warranty(self, warranty_in_months):
      self.warranty_in_months = warranty_in_months


   def display_electronic_product_details(self):
      self.display_product_details()
```

```
e = ElectronicItem("TV")
e.display_electronic_product_details()
```

Composition: Modeling instances of one class as attributes of another class is called Composition. Prefer modeling with inheritance when the classes have an HAS-A relationship.

```
class Product:
  def __init__(self, name):
   self.name = name
   self.deal_price = deal_price

  def display_product_details(self):
    print("Product: {}".format(self.name)) # Product: Milk

  def get_deal_price(self):
    return self.deal_price

class GroceryItem(Product):
  pass

class Order:
  def __init__(self):
     self.items_in_cart = []

  def add_item(self, product, quantity):
    self.items_in_cart.append((product, quantity))

  def display_order_details(self):
    for product, quantity in self.items_in_cart:
      product.display_product_details()

milk = GroceryItem("Milk")
```

```
order.add_item(milk, 2)
order.display_order_details()
```

Overriding Methods: Sometimes, we require a method in the instances of a sub class to behave differently from the method in instance of a superclass.

```
class Product:
  def __init__(self, name):
    self.name = name

  def display_product_details(self):
    print("Superclass Method")

class ElectronicItem(Product):
  def display_product_details(self): # same method name as superclass
    print("Subclass Method")

e = ElectronicItem("Laptop")
e.display_product_details()
# Output is:
Subclass Method
```

Accessing Super Class's Method: super() allows us to call methods of the superclass (Product) from the subclass. Instead of writing and methods to access and modify warranty we can override __init__.

```
class Product:
 def __init__(self, name):
   self.name = name

 def display_product_details(self):
   print("Product: {}".format(self.name)) # Product: Laptop

class ElectronicItem(Product):

 def display_product_details(self):
        super().display_product_details()
        print("Warranty {} months".format(self.warranty_in_months)) # Warranty 10 months

    def set_warranty(self, warranty_in_months):
        self.warranty_in_months = warranty_in_months


e = ElectronicItem("Laptop")
e.set_warranty(10)
e.display_product_details()
```

MultiLevel Inheritance: We can also inherit from a subclass. This is called MultiLevel Inheritance.

```
class Product:
    pass


class ElectronicItem(Product):
    pass


class Laptop(ElectronicItem):
    pass
```

Inheritance & Composition:

Inheritance	**Composition**
Car is a vehicle	Car has a Tyre
Truck is a vehicle	Order has a product

Errors & Exceptions

Errors & Exceptions: There are two major kinds of errors:

- Syntax Errors
- Exceptions

Syntax Errors: Syntax errors are parsing errors which occur when the code is not adhering to Python Syntax.

```
if True print("Hello") # SyntaxError: invalid syntax
```

- When there is a syntax error, the program will not execute even if that part of code is not used.

Exceptions: Errors detected during execution are called exceptions.

Division Example: Input given by the user is not within expected values.

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


divide(5, 0)
# Output is:
ZeroDivisionError: division by zero
```

Working With Exceptions:

Raising Exceptions:

```
raise ValueError("Unexpected Value!!") # ValueError:Unexpected Value
def divide(x, y):
    if y == 0:
        raise ValueError("Cannot divide by zero")
    return x / y


print(divide(10, 0))  # ValueError: Cannot divide by zero
```

Handling Exceptions: Exceptions can be handled with try-except block. Whenever an exception occurs at some line in try block, the execution stops at that line and jumps to except block.

```
try:
 # Write code that
 # might cause exceptions.
except:
 # The code to be run when
 # there is an exception.
def divide(x, y):
    try:
        result = x / y
    except TypeError:
        return "Invalid input"
    return result
```

```
print(divide(10, 5)) # 2.0
print(divide(10, "a"))  # Invalid input
```

Handling Specific Exceptions: We can specifically mention the name of exception to catch all exceptions of that specific type.

```
try:
   # Write code that
   # might cause exceptions.
except Exception:
   # The code to be run when
   # there is an exception.
try:
   result = 5/0
   print(result)
except ZeroDivisionError:
   print("Denominator can't be 0")
except:
   print("Unhandled Exception")
# Output is:
Denominator can't be 0
```

Handling Multiple Exceptions: We can write multiple exception blocks to handle different types of exceptions differently.

```
try:
   # Write code that
   # might cause exceptions.
except Exception1:
   # The code to be run when
   # there is an exception.
except Exception2:
   # The code to be run when
   # there is an exception.
try:
   result = 12/"a"
```

```
    print(result)
except ZeroDivisionError:
    print("Denominator can't be 0")
except ValueError:
    print("Input should be an integer")
except:
    print("Something went wrong")
# Output is:
Denominator can't be 0
```

Working With Dates & Times

Datetime: Python has a built-in datetime module which provides convenient objects to work with dates and times.

```
import datetime
```

Datetime classes: Commonly used classes in the datetime module are:

1. date class

2. time class

3. datetime class

4. timedelta class

Representing Date: A date object can be used to represent any valid date (year, month and day).

```
import datetime

date_object = datetime.date(2022, 12, 17)
print(date_object) # 2022-12-17
```

Attributes of Date Object:

```
from datetime import date

date_object = date(2019, 4, 13)
print(date_object.year) # 2019
print(date_object.month) # 4
print(date_object.day) # 13
```

Today’s Date: Class method today() returns a date object with today’s date.

```
import datetime


date_object = datetime.date.today()

print(date_object) # 2022-12-17
```

Representing Time: A time object can be used to represent any valid time (hours, minutes and seconds).

```
from datetime import time


time_object = time(11, 34, 56)

print(time_object) # 11:34:56
```

Attributes of Time Object:

```
from datetime import time


time_object = time(11, 34, 56)

print(time_object.hour) # 11

print(time_object.minute) # 34

print(time_object.second) # 56
```

Datetime: The datetime class represents a valid date and time together.

```
from datetime import datetime


date_time_obj = datetime(2018, 11, 28, 10, 15, 26)

print(date_time_obj.year) # 2018

print(date_time_obj.month) # 11

print(date_time_obj.hour) # 10

print(date_time_obj.minute) # 15
```

Timedelta: Timedelta object represents duration.

```
from datetime import timedelta


delta = timedelta(days=365, hours=4)

print(delta) # 365 days, 4:00:00
```

Calculating Time Difference:

```
import datetime


dt1 = datetime.datetime(2021, 2, 5)
dt2 = datetime.datetime(2022, 1, 1)
duration = dt2 - dt1
print(duration) # 330 days, 0:00:00
print(type(duration)) # <class 'datetime.timedelta'>
```

Formatting Datetime: The datetime classes have strftime(format) method to format the datetime into any required format like

- mm/dd/yyyy
- dd-mm-yyyy

```
from datetime import datetime


now = datetime.now()
formatted_datetime_1 = now.strftime("%d %b %Y %I:%M:%S %p")
print(formatted_datetime_1) # 05 Feb 2021 09:26:50 AM


formatted_datetime_2 = now.strftime("%d/%m/%Y, %H:%M:%S")
print(formatted_datetime_2) # 05/02/2021, 09:26:50
```

Parsing Datetime: The class method strptime() creates a datetime object from a given string representing date and time.

```
from datetime import datetime


date_string = "28 November, 2018"
print(date_string) # 28 November, 2018


date_object = datetime.strptime(date_string, "%d %B, %Y")
print(date_object) # 2018-11-28 00:00:00
```