Introduction to Python OOPS

OOPS concepts in python are very closely related to our real world, where we write programs to solve our problems. Solving any problem by creating objects is the most popular approach in programming.

This approach is termed as **Object Oriented Programming**. Object oriented programming maps our code instructions with the real world problems, making it easier to write and simpler to understand. They map real-world entities(such as company and employee) as 'software objects' that have some 'data' associated with them and can perform some 'functions'.

What is OOPS in Python?

OOPS in programming stands for **Object Oriented Programming System**. It is a programming paradigm or methodology, to design a program using classes and objects OOPS treats every entity as an object.

Object-oriented programming in Python is centered around objects. Any code written using OOPS is to solve our problem, but is represented in the form of Objects. We can create as many objects as we want, for a given class.

So what are objects? **Objects** are anything that has properties and some behaviours. The properties of objects are often referred to as variables of the object, and behaviours are referred to as the functions of the objects. Objects can be real-life or logical.

Suppose, a Pen is a real-life object. The property of a pen include: it's color, type(gel pen or ball pen). And, the behaviour of the pen may include that, it can write, draw etc.

Any file in your system is an example of logical object. Files have properties like file_name, file_location, file_size and their behaviours include they can hold data, can be downloaded, shared, etc.

Some Major Benefits of OOPS Include:

1. They reduce the redundancy of the code by writing clear and re-usable codes (using inheritance).
2. They are easier to visualize because they completely relate to real-world scenarios. For example, the concept of objects, inheritance, abstractions, relate very closely to real-world scenarios(we will discuss them further in this article).
3. Every object in oops represent a different part of the code and have their own logic and data to communicate with each other. So, there are no complications in the code.

Difference Between Object Oriented & Procedural Oriented Programming

Do you know Python follows 4 types of programming paradigms?

They include: *imperative*, *functional*, *procedural*, and *object-oriented programming!*

Here we will see the difference between 2 of the most important programming paradigms in Python: **Procedural Oriented Programming (POP)** & **Object Oriented Programming (OOP)**. Let's begin.

1. What are They?

Let us see the approach used by each paradigm -:

POP: Suppose you want to cook maggie! Then you go through a list of steps like --

- Boil some water in a Pan
- Add maggie into it
- Add maggie masala
- Cook and serve.

In a similar way, POP requires a certain procedure of steps to operate. **POP consists of functions**. A POP program is divided into functions, each of which is dedicated to a specific task. The functions are arranged in a specific order and the program control flows sequentially.

OOP: OOP consists of **objects**. They divide the program into objects. These objects are the entities that combine the properties and methods of real-world objects.

2. Where are They Preferred?

POP is suitable for small tasks only. Because, the complexity of the code increases as the length of the program increases, and it ends up being filled with functions. It becomes even more difficult to debug.

OOP is suitable for larger problems. They can make the code reusable using recursion, which makes the code cleaner and less complicated.

3. Which Provides More Security?

POP is less secure, because it provides the functions, with all the data. So, our data is not hidden. POP is not a recommended option if you want to secure your credentials or any private information!

OOP is more secure because it provides you security through data hiding. OOP has a special concept known as *"Encapsulation"*, which blesses it with the property of data hiding(we will read about this further).

4. Approach of Programming

POP follows the Top-down approach of programming. The top-down approach of programming, focuses on breaking down a big problem into smaller and understandable chunks of codes. Then it solves those smaller chunks of problems.

OOPS concepts, follows the Bottom-up approach of programming. The Bottom-Up approach first focuses on solving the smaller problems at the very fundamental level, and then integrating them into a whole and complete solution.

Class and Objects in Python

Suppose you wish to store the number of books you have, you can simply do that by using a variable. Or, say you want to calculate the sum of 5 numbers and store it in a variable, well, that can be done too!

Primitive data structures like numbers, strings, and lists are designed to store simple values in a variable. Suppose, your name, or square of a number, or count of some marbles (say).

But what if you need to store the details of all the Employees in your company? For example, you may try to store every employee in a list, you may later be confused about which index of the list represents what details of the employee(e.g. which is the name field, or the **empID** etc.)

Example:

```
employee1 = ['John Smith', 104120, "Developer", "Dept. 2A"]
employee2 = ['Mark Reeves', 211240, "Database Designer",
"Dept. 11B"]
employee3 = ['Steward Jobs', 131124, "Manager", "Dept. 2A"]
```

Even if you try to store them in a dictionary, after an extent, the whole codebase will be too complex to handle. So, in these scenarios, we use Classes in python.

A **class** is used to create user-defined data structures in Python. Classes define functions, which are termed as methods, that describe the behaviors and actions that an object created from a class can perform. OOPS concepts in python majorly deals with classes and objects.

Classes make the code more manageable by avoiding complex codebases. It does so, by creating a blueprint or a design of how anything should be defined. It defines what properties or functions, any object which is derived from the class should have.

IMPORTANT:

A class just defines the structure of how anything should look. It does not point to anything or anyone in particular. **For example,** say, HUMAN is a class, which has suppose -- **name, age, gender, city**. It does not point to any specific HUMAN out there, but yes, it explains the properties and functions any HUMAN should have or any object of class HUMAN should have.

An instance of a class is called the **object**. It is the implementation of the class and exists in real.

An **object** is a collection of data (variables) and methods (functions) that access the data. It is the real implementation of a class.

Consider this example, here Human is a class - It is just a blueprint which defines how Human should be, and not a real implementation. You may say that "Human" class just exists logically.

However, "Ron" is an object of the Human class (please refer the image given above for understanding). That means, Ron is created by using the blueprint of the Human class, and it contains the real data. "Ron" exists physically, unlike "Human" (which just exists logically). He exists in **real**, and implements all the **properties** of the class Human, such as, *Ron have a name, he is 15 years old, he is a male and lives in Delhi*. Also, Ron implements all the **methods** of Human class, suppose, *Ron can walk, speak, eat and sleep*.

And many humans can be created using the blueprint of class Human. Such as, we may create 1000s of more humans by referring to the blueprint of the class Human, using objects.

Quick Tip:

class = blueprint(suppose an architectural drawing). The Object is an actual thing that is built based on the 'blueprint' (suppose a house). An instance is a virtual copy (but not a real copy) of the object.

When a class is defined, only the blueprint of the object is created, and no memory is allocated to the class. **Memory allocation occurs only when the object or instance is created.** The object or instance contains the real data or information.

How to Define a Class in Python?

Classes in Python can be defined by the keyword class, which is followed by the name of the class and a colon.

Syntax:

```
class Human:
    pass
```

Indented code below the class definition is considered part of the class body.

'**pass**' is commonly used as placeholders, in the place of code whose implementation we may skip for the time being. "pass" allows us to run the code without throwing an error in Python.

What is an __init__ method?

The properties that all Human objects must have, are defined in a method called **init**(). Every time a new Human object is created, __init__() sets the initial state of the object by assigning the values of we provide inside the object's properties. That is, __init__() initializes each new instance of the class. __init__() can take any number of parameters, but the first parameter is always a variable called self.

The self parameter is a **reference to the current instance of the class**. It means,
the self parameter points to the address of the current object of a class, allowing us to access the data of its(the object's) variables.

So, even if we have **1000 instances** (objects) of a class, we can always get each of their individual data due to this self, because it will point to the address of that particular object and return the respective value.

Note:

We can use any name in place of self, but it has to be the first parameter of any function in the class.

Let us see, how to define __init__() in the Human class:

Code:

```
class Human:
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender
```

In the body of .__init__(), we are using the self variable 3 times, for the following:

self.name = 'name' creates an attribute called name and assigns to it the value of the name parameter.

self.age = age attribute is created and assigned to the value of age parameter passed.

self.gender = gender attribute is created and assigned to the value of gender parameter passed.

There are **2 types of attributes** in Python:

1. Class Attribute:

These are the variables that are same for all instances of the class. They do not have new values for each new instance created. They are defined just below the class definition.

Code:

```
class Human:
    #class attribute
    species = "Homo Sapiens"
```

Here, the species will have a fixed value for any object we create.

2. Instance Attribute:

Instance attributes are the variables which are defined inside of any function in class. Instance attribute have different values for every instance of the class. These values depends upon the value we pass while creating the instance.

Code:

```
class Human:
    #class attribute
    species = "Homo Sapiens"
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender
```

Here, name,age and gender are the instance attributes. They will have different values as for new instances of class.

For properties that should have a similar value per instance of a class, use **class attributes**. For properties that differ per instance, use **instance attributes**.

Creating an Object in Class

When we create a new object from a class, it is called **instantiating an object**. An object can be instantiated by the class name followed by the parantheses. We can assign the object of a class to any variable.

Syntax:

```
x = ClassName()
```

As soon as an object is instantiated, memory is allocated to them. So, if we compare 2 instances of a same class using '==', it will return false(because both will have different memory assigned).

Suppose, we try to create objects of our Human class, then we also need to pass the values for name, age and gender.

Code:

```
class Human:
    #class attribute
    species = "Homo Sapiens"
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender

x = Human("Ron", 15, "Male")
y = Human("Miley", 22, "Female")
```

Here, we have created 2 objects of the class Human passing all the required arguments.

Warning: If we do not pass the required arguments, it will throw a **TypeError:** *TypeError: init() missing 3 required positional arguments: 'name', 'age' and 'gender'.*

Let us now see, how to access those values using objects of the class. We can access the values of the instances by using the dot notation.

Code:

```
class Human:
    #class attribute
    species = "Homo Sapiens"
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender

x = Human("Ron", 15, "Male")
y = Human("Miley", 22, "Female")
print(x.name)
print(y.name)
```

Output:

```
Ron
Miley
```

So, we find that we can access the instance and class attributes just by using the dot operator.

Code:

```
class Human:
    species = "Homo Sapiens"

    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender


# x and y are instances of class Human
x = Human("Ron", 15, "male")
y = Human("Miley", 22, "female")

print(x.species)  # species are class attributes, hence will
have same value for all instances
print(y.species)
```

```
# name, gender and age will have different values per
instance, because they are instance attributes
print(f"Hi! My name is {x.name}. I am a {x.gender}, and I am
{x.age} years old")
print(f"Hi! My name is {y.name}. I am a {y.gender}, and I am
{y.age} years old")
```

Output:

```
Homo Sapiens
Homo Sapiens
Hi! My name is Ron. I am a male, and I am 15 years old
Hi! My name is Miley. I am a female, and I am 22 years old
```

In the above example, we have our class attributes values same "Homo Sapiens", but the instance attributes values are different as per the value we passed while creating our object.

However, we can change the value of class attributes, by assigning classname.classAttribute with any new value.

Code:

```
class Human:
    #class attribute
    species = "Homo Sapiens"
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender

Human.species = "Sapiens"
obj = Human("Brek",11,"male")
print(obj.species)
```

Output:

```
Sapiens
```

Instance Methods

An instance method is a function defined within a class that can be called only from instances of that class. Like init(), an instance method's first parameter is always self.

Let's take an example and implement some functions can class Human can perform --

Code:

```
class Human:
    #class attribute
    species = "Homo Sapiens"
```

```
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender

    #Instance Method
    def speak(self):
        return f"Hello everyone! I am {self.name}"

    #Instance Method
    def eat(self, favouriteDish):
        return f"I love to eat {favouriteDish}!!!"

x = Human("Ciri",18,"female")
print(x.speak())
print(x.eat("momos"))
```

Output:

```
Hello everyone! I am Ciri
I love to eat momos!!!
```

This Human class has two instance methods:

1. **speak():** It returns a string displaying the name of the Human.
2. **eat():** It has one parameter "favouriteDish" and returns a string displaying the favourite dish of the Human.

Having gained a thorough knowledge of what Python classes, objects, and methods are, it is time for us to turn our focus towards the OOP core principles, upon which it is built.

Encapsulation

You must have seen medicine capsules, where all the medicines remain enclosed inside the cover of the capsule. Basically, capsule encapsulates several combinations of medicine.

Similarly in programming, the variables and the methods remains enclosed inside a capsule called the 'class' ! Yes, we have learnt a lot about classes in Python and we already know that, all the variables and functions we create in **OOP** remains inside the class.

The process of binding data and corresponding methods (behavior) together into a single unit is called encapsulation in Python.

In other words, encapsulation is a programming technique that binds the class members (variables and methods) together and prevents them from being accessed by other classes. It is one of the concept of OOPS in Python.

Encalpsulation is a way to ensure **security**. It hides the data from the access of outsiders. An organization can protect its object/information against unwanted access by clients or any unauthorized person by encapsulating it.

Getters and setters

We mainly use encapsulation for **Data Hiding**. We does so by
defining **getter** and **setter** methods for our classes. If anyone wants some data, they can only get it by calling the getter method. And, if they want to set some vlaue to the data, they must use the setter method for that, otherwise they won't be able to do the same. But internally, how this getter and setter methods are performing remains hidden from outside world.

Code of getter & setter in Encapsulation

```
class Library:
    def __init__(self, id, name):
        self.bookId = id
        self.bookName = name

    def setBookName(self, newBookName): #setters method to set
the book name
        self.bookName = newBookName

    def getBookName(self): #getters method to get the book
name
        print(f"The name of book is {self.bookName}")


book = Library(101,"The Witchers")
book.getBookName()
book.setBookName("The Witchers Returns")
book.getBookName()
```

Output:

```
The name of book is The Witchers
The name of book is The Witchers Returns
```

In the above example, we defined the **getter** getBookName() and **setter** setBookName() to get and set the names of books respectively. So, now we can only get and set the book names upon calling the methods, otherwise, we cannot directly get or modify any value. This promotes high security to our data, because others are not aware at a deep level how the following methods are implemented(if their access are restricted).

Abstraction

It is likely that you are reading this article on your laptop, phone, or tablet. You are also probably making notes, highlighting important points, and you may be saving some points in your internal files while reading it. As you read this, all you see before you is a 'screen' and all this data that is shown to you. As you type, all you see is the keys on the keyboard and

you don't have to worry about the internal details, like how pressing a key may lead to displaying that word onscreen. Or, how clicking on a button on your screen could open a new tab!

So, everything we can see here is at an abstract level. We are not able to see the internal details, but just the result it is producing(which actually matters to us).

Abstraction in a similar way, just shows us the functionalities anything holds, hiding all the implementations or inner details.

The main goal of **Abstraction** is to hide background details or any unnecessary implementation about the data so that users only see the required information. It helps in handling the complexity of the codes.

Key Points of Abstract classes

- **Abstraction** is used for hiding the background details or any unnecessary implementation about the data, so that users only see the required information.
- In Python, abstraction can be achieved by using **abstract classes**
- A class that consists of one or more **abstract methods** is called the "abstract class".
- **Abstract methods** do not contain any implementation of their own.
- Abstract class can be **inherited by any subclass**. The subclasses that inherits the abstract classes provides the implementations for their abstract methods .
- Abstract classes can act like **blueprint to other classes**, which are useful when we are designing large functions. And the subclass which inherits them can refer the abstract methods for implementing the features.
- Python provides the **abc** module to use the abstraction

Advantages of OOPS in Python

There are numerous advantages of OOPS concepts in Python, making it favourable for writing serious softwares. Let us look into a few of them --

1. Effective problem solving because, for each mini-problem, we write a class that does what is required. And then we can reuse those classes, which makes it even quicker to solve the next problem.
2. Flexibility of having multiple forms of a single class, through polymorphism
3. Reduced high complexity of code, though abstraction.
4. High security and data privacy through encapsulation.
5. Reuse of code, by the child class inheriting properties of parent class through inheritance.
6. Modularity of code allows us to do easy debugging, instead of looking into hundreds lines of codes to find a single issue.

Destructors

```
# Python program to illustrate destructor
class Employee:

    # Initializing
    def __init__(self):
        print('Employee created.')

    # Deleting (Calling destructor)
    def __del__(self):
        print('Destructor called, Employee deleted.')

obj = Employee()
del obj
```

Python Generators

What are Python Generators?

Python's generator functions are used to create iterators(which can be traversed like list, tuple) and return a traversal object. It helps to transverse all the items one at a time present in the iterator.

Generator functions are defined as the normal function, but to identify the difference between the normal function and generator function is that in the normal function, we use the return keyword to return the values, and in the generator function, instead of using the return, we use yield to execute our iterator.

Example:

```
def gen_fun():
    yield 10
    yield 20
    yield 30

for i in gen_fun():
    print(i)
10
20
30
```

In the above example, gen_fun() is a generator function. This function uses the yield keyword instead of return, and it will return a value whenever it is called.

Yield vs Return

Yield	Return
It is used in generator functions.	It is used in normal functions.
It is responsible for controlling the flow of the generator function. After returning the value from yield, it pauses the execution by saving the states.	Return statement returns the value and terminates the function.

Difference Between Generator Function & Normal Function

- In generator functions, there are one or more yield functions, whereas, in Normal functions, there is only one function
- When the generator function is called, the normal function pauses its execution, and the call is transferred to the generator function.
- Local variables and their states are remembered between successive calls.

Exception Handling

Writing code in order to solve or automate some huge problems is a pretty powerful thing to do, isn't it? But as Peter Parker says, 'with great power comes great responsibility'. This in fact holds true when writing your code as well! While our code is being executed, there might be some events that disrupt the normal flow of your code. These events are errors, and once these occur python interpreter is in a situation that it cannot deal with and hence raises an **exception**.

In python, an exception is a class that represents error. If these exceptions are not handled, our application or programs go into a crash state. As a developer, we definitely have the power to write code and solve problems but it becomes our responsibility to handle these exceptions that might occur and disrupt the code flow.

Let's see how these exceptions can be handled in this article but first of all, let's take a look at an example that will introduce you to an exception:

```
a = 10
b = 0
c = b/a
print(c)
```

Output:

```
Traceback (most recent call last):
    File "main.py", line 3, in <module>
```

```
    c = a/b
ZeroDivisionError: division by zero
```

Above is an example of what we call an unhandled exception. On line 3, the code went into a halt state as it's an unhandled exception. To avoid such halt states, let's take a look at how to handle exceptions/errors.

What is Exception Handling in Python?

Exceptions can be unexpected or in some cases, a developer might expect some disruption in the code flow due to an exception that might come up in a specific scenario. Either way, it needs to be handled.

Python just like any other programming language provides us with a provision to handle exceptions. And that is by try & except block. Try block allows us to write the code that is prone to exceptions or errors. If any exception occurs inside the try block, the except block is triggered, in which you can handle the exception as now the code will not go into a halt state but the control will flow into the except block where it can be manually handled.

Any critical code block can be written inside a try clause. Once the expected or unexpected exception is raised inside this try, it can be handled inside the except clause (This 'except' clause is triggered when an exception is raised inside 'try'), as this except block detects any exception raised in the try block. By default, except detects all types of exceptions, as all the built-in exceptions in python are inherited from common class Exception.

The basic syntax is as follows:

```
try:
    # Some Code
except:
    # Executed if we get an error in the try block
    # Handle exception here
```

Try and except go hand in hand i.e. the syntax is to write and use both of them. Writing just try or except will give an error.

Let's consider an example where we're dividing two numbers. Python interpreter will raise an exception when we try to divide a number with 0. When it does, we can take custom action on it in the except clause.

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

try:
   divideNos(10, 0)
except:
   print('some exception occured')
```

Output:

```
some exception occured
```

But how do we know what exactly error was read by the python interpreter?

Well, when a python interpreter raises an exception, it is in the form of an object that holds information about the exception type and message. Also, every exception type in python inherits from the base class Exception.

```
def divideNos(a, b):
  return a/b
try:
  divideNos(10, 0)
# Any exception raised by the python interpreter, is inherited by the base class 'Exception', hence any exception raised in the try block will be detected and collected further in the except block for handling.
except Exception as e:
 print(e) # as e is an object of type Exception, Printing here to see what message it holds.
 print(e.__class__)
division by zero
<class 'ZeroDivisionError'>
```

In the above code, we used the Exception class with the except statement. It is used by using the as keyword. This object will contain the cause of the exception and we're printing it in order to look what the reason is inside this Exception object.

Common Exceptions in Python

Before proceeding further, we need to understand what some of the common exceptions that python throws are. All the inbuilt exceptions in python are inherited from the common 'Exception' class. Some of the common inbuilt exceptions are:

Exception Name	Description
Exception	All exceptions inherit this class as the base class for all exceptions.
StopIteration	Raised when the next() method of an iterator while iteration does not point to any object
StandardError	All exceptions inherit this class except stop StopIteration and SystemExit
ArithmeticError	Errors that occur during numeric calculation are inherited by it.
OverflowError	When a calculation exceeds the max limit for a specific numeric data type
ZeroDivisionError	Raised when division or modulo by zero takes place.
AssertionError	Raised when an assert statement fails
AttributeError	Raised in case of failure of attribute assignment
EOFError	Raised when the end of file is reached, usually occurs when there is no input from input() function
ImportError	Raised when an import statement fails
NameError	Raised when an identifier is not found in the local or non-local or global scope.
SyntaxError	Raised when there is an error in python syntax.

Exception Name	Description
IndentationError	Raised when indentation is not proper
TypeError	Raised when a specific operation of a function is triggered for an invalid data type
ValueError	Raised when invalid values are provided to the arguments of some builtIn function for a data type that has a valid type of arguments.
RuntimeError	Raised when an error does not fall into any other category
NotImplementedError	Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.

These are just some of the common exceptions in python. To understand more on types of exceptions check [python's official documentation on exceptions]

Catching Specific Exceptions in Python

In the above example, we caught the exception that was being raised in the try block, but the except blocks are going to catch all the exceptions that try might raise.

Well, it's considered a good practice to catch specific types of exceptions and handle them accordingly. And yes, try can have multiple except blocks. We can also use a tuple of values in an except to receive multiple specific exception types.

Let's take an example to understand this more deeply:

```
def divideNos(a, b):
  return a/b # An exception might raise here if b is 0
(ZeroDivisionError)
try:
   a = input('enter a:')
   b = input('enter b:')
   print('after division', divideNos(a, b))
   a = [1, 2, 3]
   print(a[3]) # An exception will raise here as size of array
'a' is 3 hence is accessible only up until 2nd index

# if IndexError exception is raised
except IndexError:
   print('index error')
# if ZeroDivisionError exception is raised
except ZeroDivisionError:
   print('zero division error')
```

Output:

```
enter a:4
enter b:2
after division 2.0
index error
```

- In the above code the exceptions raised totally depend upon the input that the user might enter. Hence if a user enters a value as 0 for ‘b’, the python interpreter will raise a ZeroDivisionError.
- And as the array ‘a’ has a length of 3, and the user is trying to access an element at index 3, an IndexError will be raised by the python interpreter.
- Each except block has been defined for both the exceptions as one of them receives exceptions of type IndexError and the other receives of type ZeroDivisionError.

If we want the above snippet to catch exception for both IndexError OR ZeroDivisionError, it can be re-written as:

```
def divideNos(a, b):
 return a/b
try:
 a = int(input('enter a:'))
 b = int(input('enter b:'))
 print('after division', divideNos(a,b))
 a = [1, 2, 3]
 print(a[3])
except (IndexError, ZeroDivisionError):
  print('index error OR zero division error')
```

```
enter a:10
enter b:2
after division 5.0
index error OR zero division error
```

Note: Only one of the except blocks is triggered when an exception is raised. Consider an example where we have one except as except IndexError and another as except(IndexError, ZeroDivisionError) then the one written first will trigger.

Raising Custom Exceptions

Even though exceptions in python are automatically raised in runtime when some error occurs. Custom and predefined exceptions can also be thrown manually by raising it for specific conditions or a scenario using the raise keyword. (A custom exception is an exception that a programmer might raise if they have a requirement to break the flow of code in some specific scenario or condition) String messages supporting the reasons for raising an exception can also be provided to these custom exceptions.

Syntax to Raise an Exception

```
try:
    # on some specific condition or otherwise
    raise SomeError(OptionalMsg)
Except SomeError as e:
    # Executed if we get an error in the try block
    # Handle exception ‘e’ accordingly
```

For example:

```
def isStringEmpty(a):
   if(type(a)!=str):
       raise TypeError('a has to be string')
   if(not a):
       raise ValueError('a cannot be null')
   a.strip()
   if(a == ''):
       return False
   return True

try:
 a = 123
 print('isStringEmpty:', isStringEmpty(a))
except ValueError as e:
   print('ValueError raised:', e)

except TypeError as e:
   print('TypeError raised:', e)
```

Output:

```
TypeError raised: a has to be string
```

- In the above code, the variable ‘a’ can hold whatever value that is assigned to it. Here we assign it a number and we’re passing to a custom method isStringEmpty that checks if a string is an empty string.
- But we orchestrated it to throw a TypeError, by assigning ‘a’ variable a number.
- In the method, we’re checking if the variable is a string or not and if it holds a value or not. In this case, it is supposed to be a string, but assigned it as a number as we’re raising an exception by using

try except and ELSE!

Sometimes you might have a use case where you want to run some specific code only when there are no exceptions. For such scenarios, the else keyword can be used with the try block. Note that this else keyword and its block are optional.

Syntax With Else Clause

```
try:
    # on some specific condition or otherwise
    raise SomeError(OptionalMsg)
except SomeError as e:
    # Executed if we get an error in the try block
    # Handle exception ‘e’ accordingly
else
```

```
    # Executed if no exceptions are raised
```

Example:

When an exception is not raised, it flows into the optional else block.

```
try:
 b  = 10
 c = 2
 a = b/c
 print(a)
except:
 print('Exception raised')
else:
  print('no exceptions raised')
```

```
5.0
no exceptions raised
```

In the above code, As both the inputs are greater than 0 which is not a risk to DivideByZeroException, hence try block won't raise any exception and hence 'except' block won't be triggered. And only when the control doesn't flow to the except block, it flows to the else block. Further handling can be done inside this else block if there is something you want to do.

Example:

When an exception is raised, control flows into the except block and not the else block.

```
try:
   b  = 10
   c = 0
   a = b/c
   print(a)
except Exception as e:
   print('Exception raised:', e)
else:
   print('no exceptions raised')
```

Output:

```
Exception raised: division by zero
```

In the above code, As both the 'b' input is 0 which is a risk to DivideByZeroException, hence the 'try' block will raise an exception, and hence the 'except' block will be triggered. And now as there is an exception raised, the control flows to the except block and not to the else block.

Try Clause with Finally

Finally is a keyword that is used along with try and except, when we have a piece of code that is to be run irrespective of if try raises an exception or not. Code inside finally will always run after try and catch.

Example:

Where an exception is raised in the try block and except is triggered.

```
try:
 temp = [1, 2, 3]
 temp[4]
except Exception as e:
 print('in exception block: ', e)
else:
 print('in else block')
finally:
 print('in finally block')
```

```
in exception block: list index out of range
in finally block
```

In the above code, we're creating an array with 3 elements, i.e. max index up till 2. But when we try to access the 4th index, it will raise an exception of index out of range and will be caught in the except block. But here we need to observe that the finally block has also been triggered.

Example:

Rewriting the above code such that exception is not raised. Where an exception is not raised and else and finally both are triggered.

Note: else block will always be triggered before finally and finally will always trigger irrespective of any exceptions raised or not.

```
try:
 temp = [1, 2, 3]
 temp[1]
except Exception as e:
 print('in exception block: ', e)
else:
 print('in else block')
finally:
 print('in finally block')
```

```
in else block
in finally block
```

In the above code, we're creating an array with 3 elements, i.e. max index up till 2. But when we try to access the 2nd index, now it will not raise an exception and control will now flow to another block and then to finally.

But here we need to observe that the finally block has also been triggered even though the exception was not raised.

Why Use Finally or Else in try..except?

```
def Checkout(itemsInCart):
  try:
    id = GetOrderIdForBooking() # Always creates an order
before booking items
    BookItems(itemsInCart) # assuming this method raises an
exception while booking one of the items
  except Exception as e:
    LogErrorOccuredWhileBooking(e) # Log failed booking
  else:
    LogSuccessfulBookingStatus(id) # Log successful booking
  finally:
    EmailOrderStatusToUser(id) # Either ways send order email
to user

itemsInCart = ['mobile', 'earphones', 'charger'] # assume user
has these items in their cart
Checkout(itemsInCart) # checking out items the user has in
their cart.
```

- Consider above as a pseudo-code to a program where it checks out items added by the user in their cart.
- The Checkout method always creates an order id before booking all the items in the cart.
- Assuming itemsInCart contains all the items the user has added to a cart.
- In the checkout experience of an e-commerce website, an orderId is created irrespective of whether a booking of all items has been a success, failed, or partially failed.

- Hence in the above pseudo-code, we're creating an orderId using GetOrderIdForBooking() and assume it returns a random string for each order. BookItems() books all the items for us in the itemsInCart array.

- If some error occurs while booking items, we have to log the reason for the exception LogErrorOccuredWhileBooking() does that job for us and if it's successful we have to log that the booking was successful.
- Now whatever the status of the order is, either way we have to email the status of the order to the user and that's what the EmailOrderStatus() does for us.

The above example perfectly fits for an application of try, except or else and finally block as each block is playing a role that might've been difficult to achieve if we didn't use these blocks.

What Happens if Errors are Raised in Except or Finally Block?

```
try:
  SomeFunction() # assuming it throws some error
except exception as e:
  Log(e) # assuming some exception occurred again while
logging this exception
finally:
  # This will still execute even after an error was raised
inside catch block & gives you a final chance to handle the
situation your way
```

- If an error occurs while handling something in an except block, the finally block still gives us a chance to handle the situation our way.
  Note: Once an exception/error is raised in the except block, finally block is triggered but the program still goes into a halt state post to that and the flow is broken.
- But what if an error occurs in the finally block? Let's orchestrate these series of events in the following code:

```
try:
  raise Exception("message from try block")
except Exception as e:
  print('in except block:', e)
  raise Exception("exception raised in except block")
finally:
  print("in finally")
  raise Exception("exception raised in finally block")
```

```
in except block: message from try block
in finally
Traceback (most recent call last):
  File "<string>", line 2, in <module>
Exception: message from try block
```

During handling of the above exception, another exception occurred:

```
Traceback (most recent call last):
  File "<string>", line 5, in <module>
Exception: exception raised in except block
```

During handling of the above exception, another exception occurred:

```
Traceback (most recent call last):
  File "<string>", line 8, in <module>
Exception: exception raised in finally block
```

- Well, the finally block won't be completed beyond the point where the exception is thrown. Note that except block won't be triggered again.