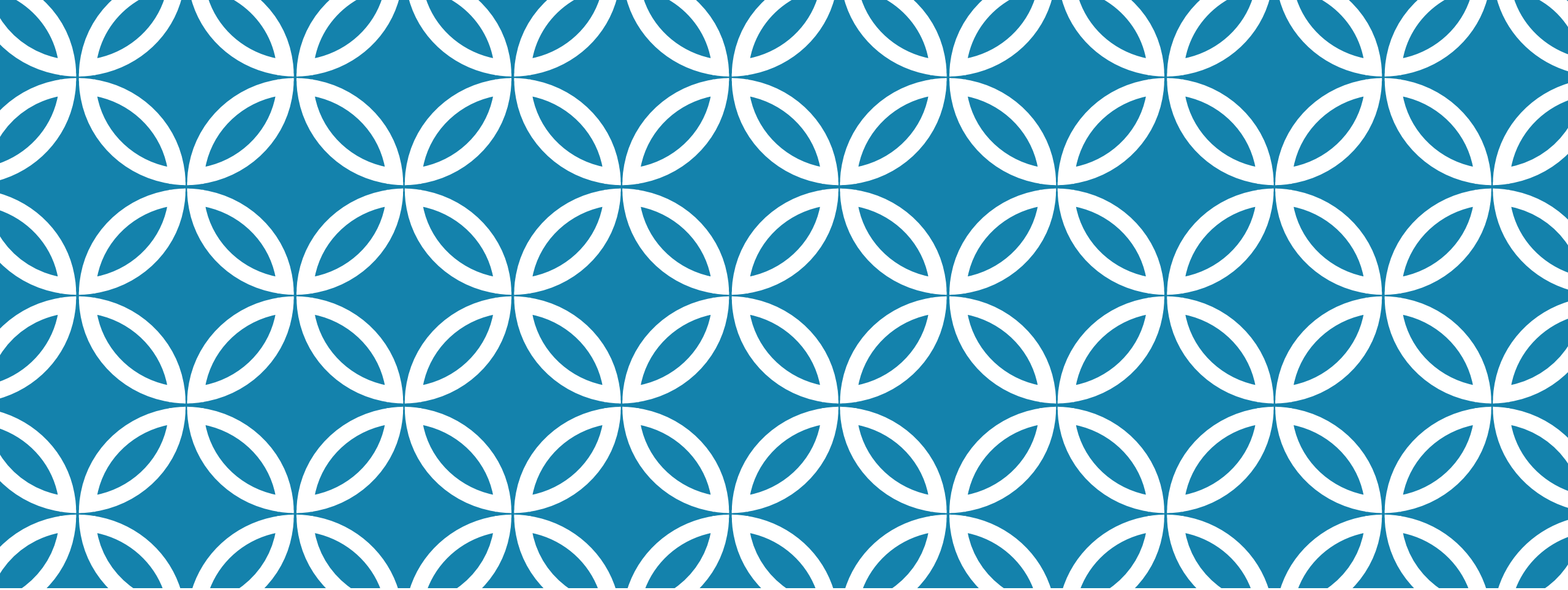
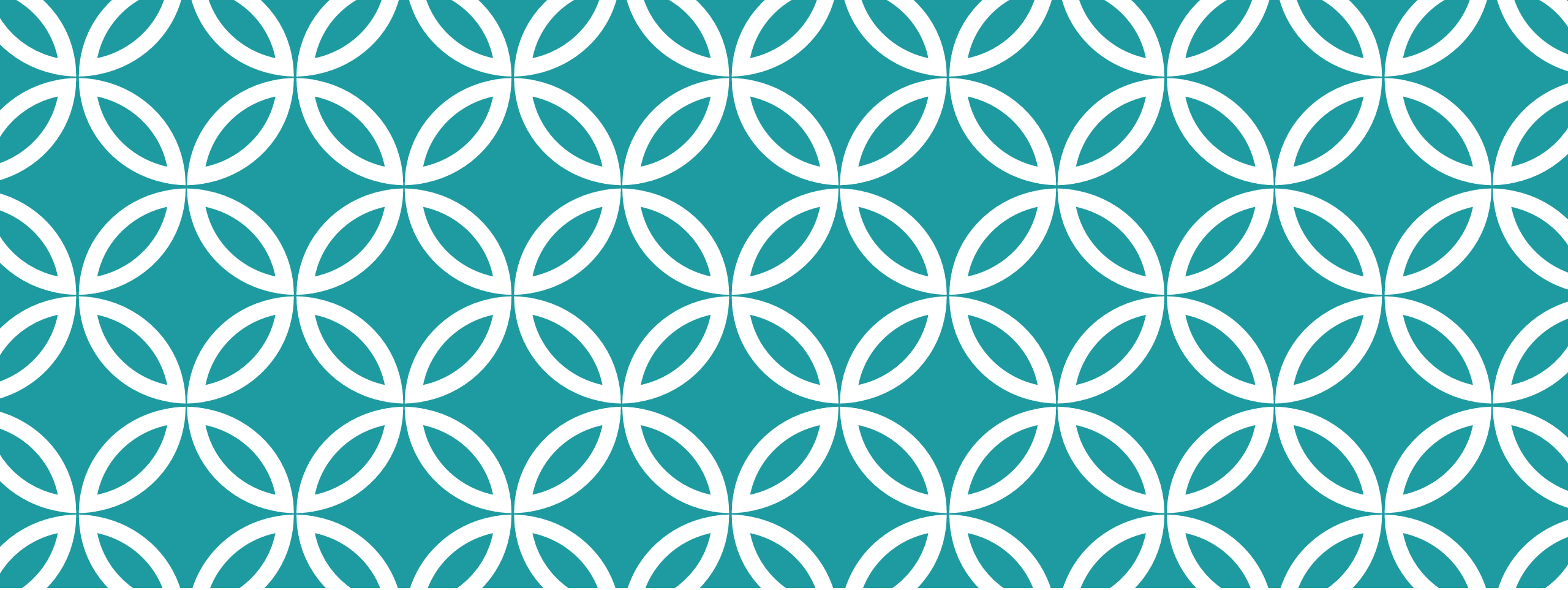


PYTHON



EXCEPTION HANDLING

An exception can be defined as an unusual condition in a program resulting in the interruption in the flow of the program.

Whenever an exception occurs, the program stops the execution, and thus the further code is not executed. Therefore, an exception is the run-time errors that are unable to handle to Python script. An exception is a Python object that represents an error

Python provides a way to handle the exception so that the code can be executed without any interruption. If we do not handle the exception, the interpreter doesn't execute all the code that exists after the exception.

Python has many **built-in exceptions** that enable our program to run without interruption and give the output. These exceptions are given below:

COMMON EXCEPTIONS

Python provides the number of built-in exceptions, but here we are describing the common standard exceptions. A list of common exceptions that can be thrown from a standard Python program is given below.

1. **ZeroDivisionError:** Occurs when a number is divided by zero.
2. **NameError:** It occurs when a name is not found. It may be local or global.
3. **IndentationError:** If incorrect indentation is given.
4. **IOError:** It occurs when Input Output operation fails.
5. **EOFError:** It occurs when the end of the file is reached, and yet operations are being performed.

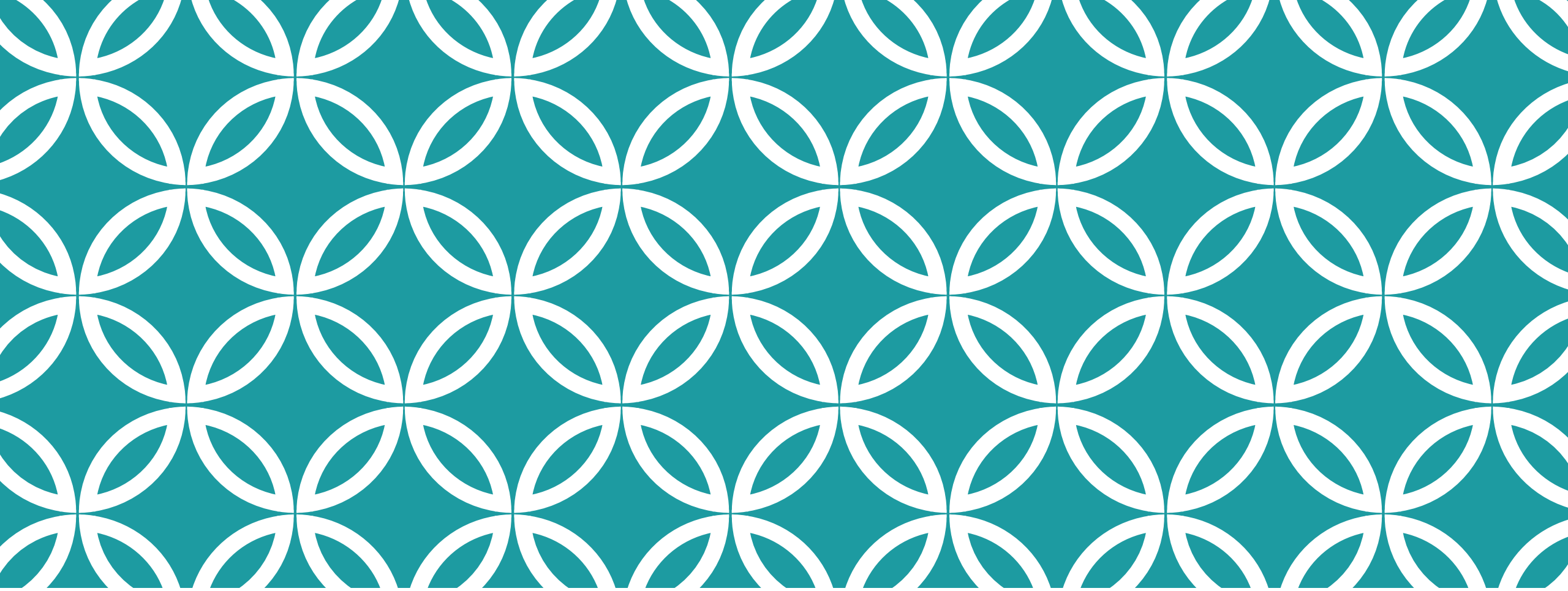
THE PROBLEM WITHOUT HANDLING EXCEPTIONS

As we have already discussed, the exception is an abnormal condition that halts the execution of the program.

Suppose we have two variables **a** and **b**, which take the input from the user and perform the division of these values. What if the user entered the zero as the denominator? It will interrupt the program execution and through a **ZeroDivision** exception. Let's see the following example.

```
a = int(input("Enter a:"))
b = int(input("Enter b:"))
c = a/b
print("a/b = %d" %c)

#other code:
print("Hi I am other part of the program")
```

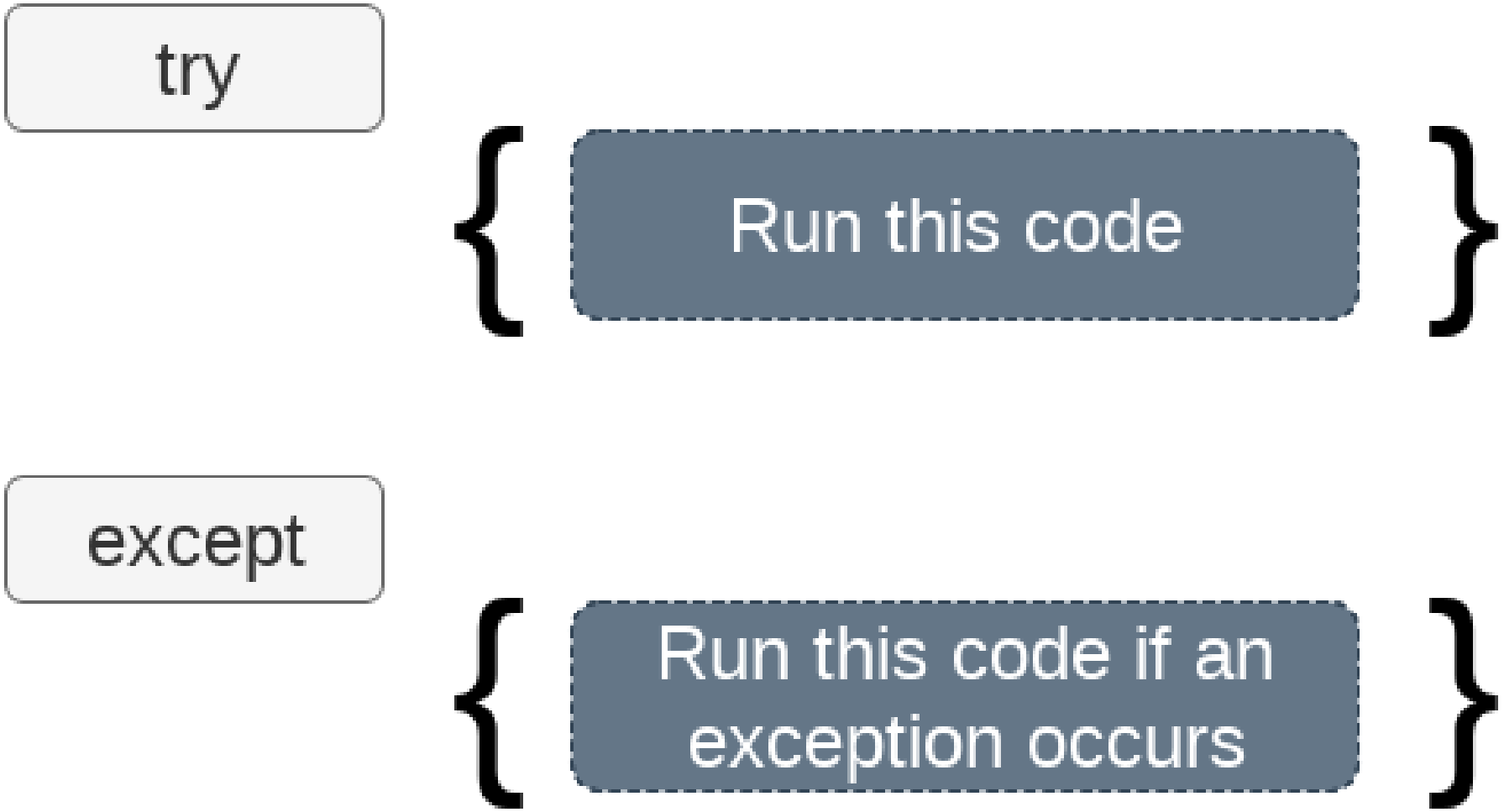
The above program is syntactically correct, but it through the error because of unusual input. That kind of programming may not be suitable or recommended for the projects because these projects are required uninterrupted execution. That's why an exception-handling plays an essential role in handling these unexpected exceptions. We can handle these exceptions in the following way.



EXCEPTION HANDLING IN PYTHON

THE TRY-EXPECT STATEMENT

If the Python program contains suspicious code that may throw the exception, we must place that code in the **try** block. The **try** block must be followed with the **except** statement, which contains a block of code that will be executed if there is some exception in the try block.


try
{
Run this code
}
except
{
Run this code if an exception occurs
}

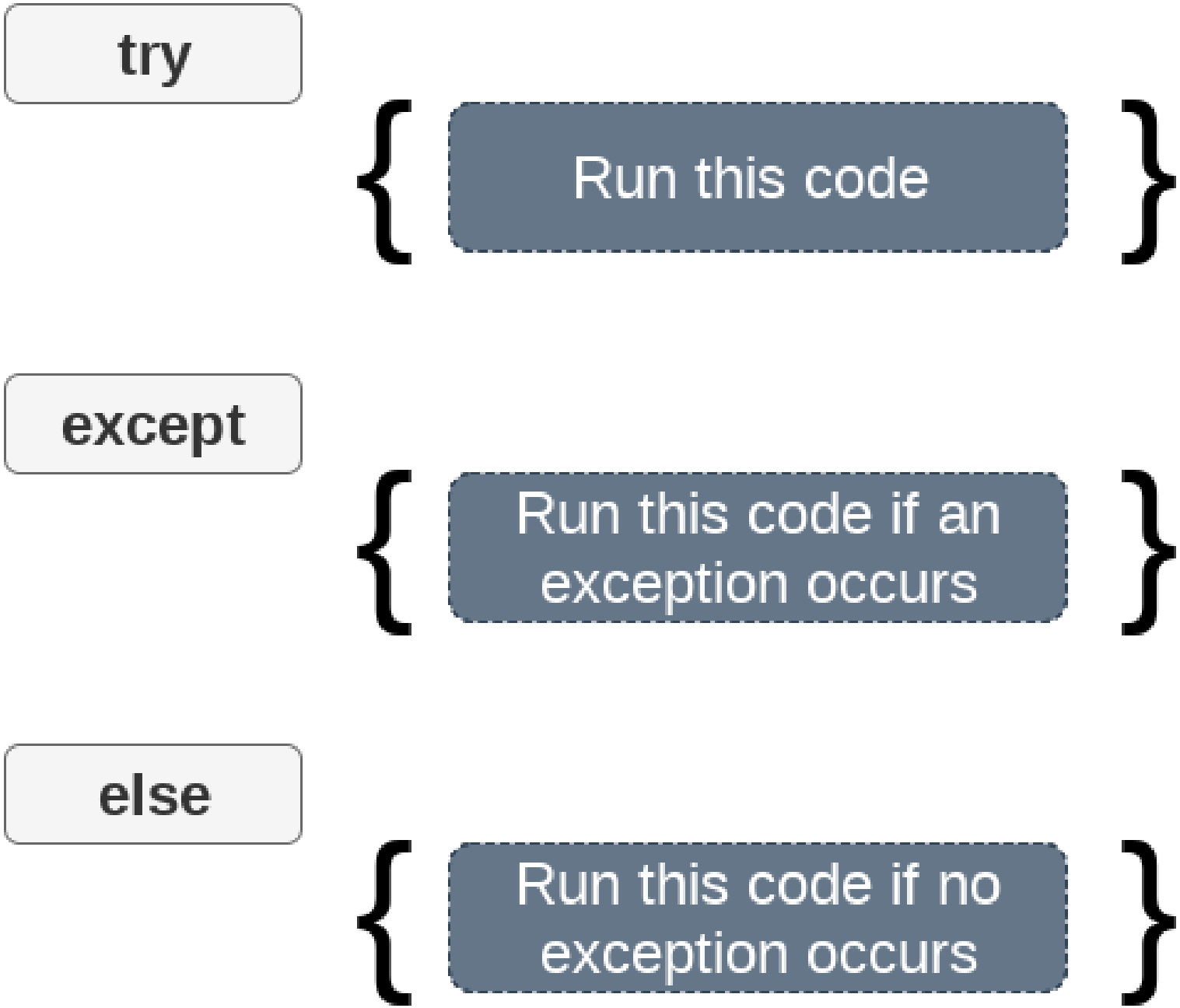
```
try:
    #block of code
except Exception1:
    #block of code
except Exception2:
    #block of code

#other code
```

```
try:
    a = int(input("Enter a:"))
    b = int(input("Enter b:"))
    c = a/b
except:
    print("Can't divide with zero")
```

ELSE WITH TRY

We can also use the else statement with the try-except statement in which, we can place the code which will be executed in the scenario if no exception occurs in the try block.

```
try:
    #block of code
except Exception1:
    #block of code
else:
    #this code executes if no except block is executed
```


try
{
Run this code
}
except
{
Run this code if an exception occurs
}
else
{
Run this code if no exception occurs
}

```
try:
    a = int(input("Enter a:"))
    b = int(input("Enter b:"))
    c = a/b
    print("a/b = %d"%c)
# Using Exception with except statement. If we print(Exception) it will return exception
 class
except Exception:
    print("can't divide by zero")
    print(Exception)
else:
    print("Hi I am else block")
```

THE EXCEPT STATEMENT WITH NO EXCEPTION

We can use the exception variable with the **except** statement. It is used by using the **as** keyword. this object will return the cause of the exception. Consider the following example:

```
try:
    a = int(input("Enter a:"))
    b = int(input("Enter b:"))
    c = a/b
    print("a/b = %d"%c)
    # Using exception object with the except statement
except Exception as e:
    print("can't divide by zero")
    print(e)
else:
    print("Hi I am else block")
```

POINTS TO REMEMBER

1. Python facilitates us to not specify the exception with the except statement.
2. We can declare multiple exceptions in the except statement since the try block may contain the statements which throw the different type of exceptions.
3. We can also specify an else block along with the try-except statement, which will be executed if no exception is raised in the try block.
4. The statements that don't throw the exception should be placed inside the else block.

DECLARING MULTIPLE EXCEPTIONS

The Python allows us to declare the multiple exceptions with the except clause. Declaring multiple exceptions is useful in the cases where a try block throws multiple exceptions. The syntax is given below.

```
try:
    #block of code

except (<Exception 1>,<Exception 2>,<Exception 3>,...<Exception n>)
    #block of code

else:
    #block of code
```

```
try:
    a=10/0;
except(ArithmeticError, IOError):
    print("Arithmetic Exception")
else:
    print("Successfully Done")
```

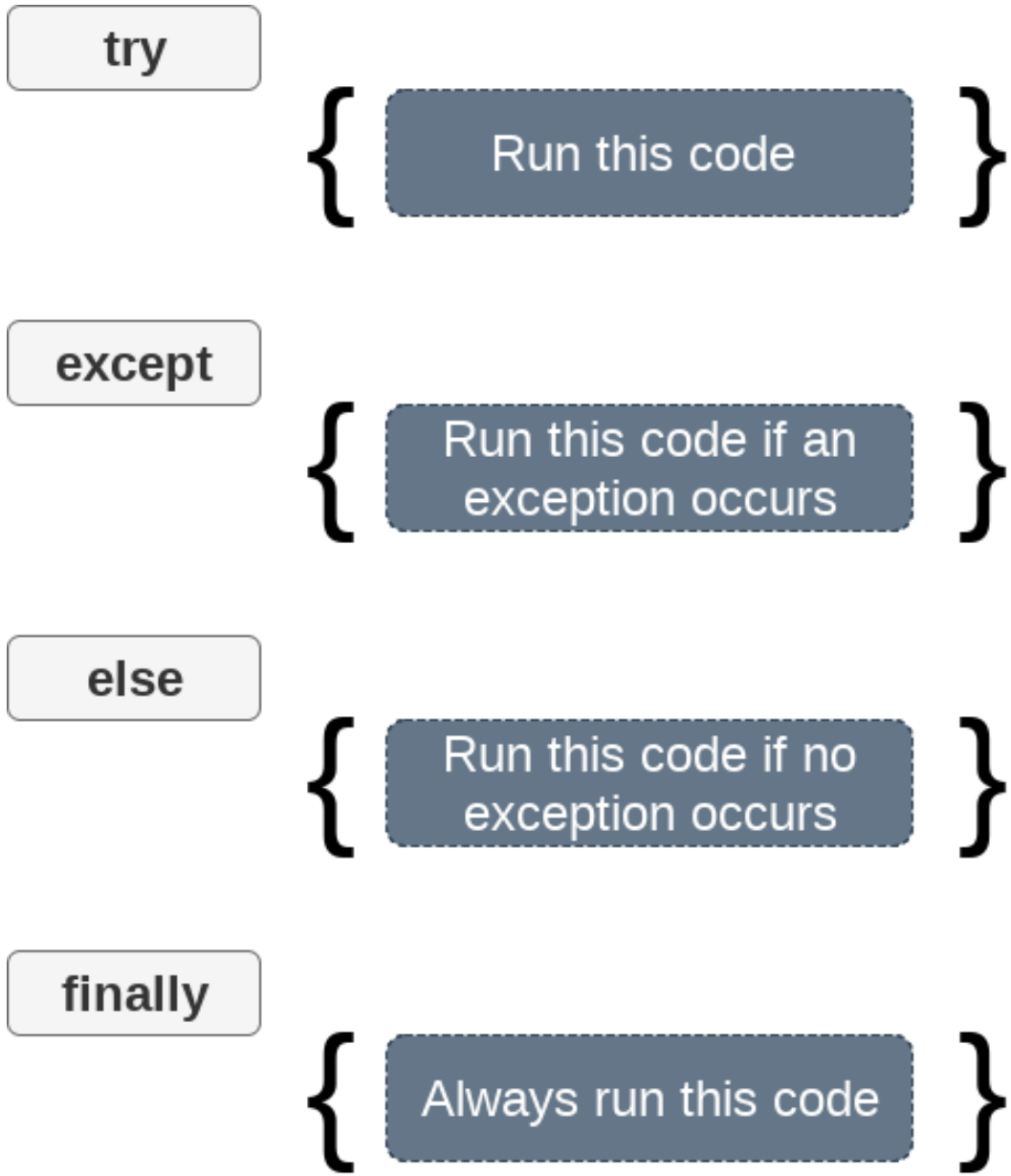
THE TRY...FINALLY BLOCK

Python provides the optional **finally** statement, which is used with the **try** statement. It is executed no matter what exception occurs and used to release the external resource. The finally block provides a guarantee of the execution.

We can use the finally block with the try block in which we can pace the necessary code, which must be executed before the try statement throws an exception.

The syntax to use the finally block is given below.

```
try:
    # block of code
    # this may throw an exception
finally:
    # block of code
    # this will always be executed
```


try
{ Run this code }
except
{ Run this code if an exception occurs }
else
{ Run this code if no exception occurs }
finally
{ Always run this code }

```
try:
    fileptr = open("file2.txt","r")
    try:
        fileptr.write("Hi I am good")
    finally:
        fileptr.close()
        print("file closed")
except:
    print("Error")
```

RAISING EXCEPTIONS

An exception can be raised forcefully by using the **raise** clause in Python. It is useful in in that scenario where we need to raise an exception to stop the execution of the program.

For example, there is a program that requires 2GB memory for execution, and if the program tries to occupy 2GB of memory, then we can raise an exception to stop the execution of the program.

The syntax to use the raise statement is given below.

raise Exception_class,<value>

POINTS TO REMEMBER

1. To raise an exception, the raise statement is used. The exception class name follows it.
2. An exception can be provided with a value that can be given in the parenthesis.
3. To access the value "**as**" keyword is used. "**e**" is used as a reference variable which stores the value of the exception.
4. We can pass the value to an exception to specify the exception type.

EXAMPLE 1

```
try:
    age = int(input("Enter the age:"))
    if(age<18):
        raise ValueError
    else:
        print("the age is valid")
except ValueError:
    print("The age is not valid")
```

EXAMPLE 2

```
try:
    num = int(input("Enter a positive integer: "))
    if(num <= 0):
# we can pass the message in the raise statement
        raise ValueError("That is  a negative number!")
except ValueError as e:
    print(e)
```

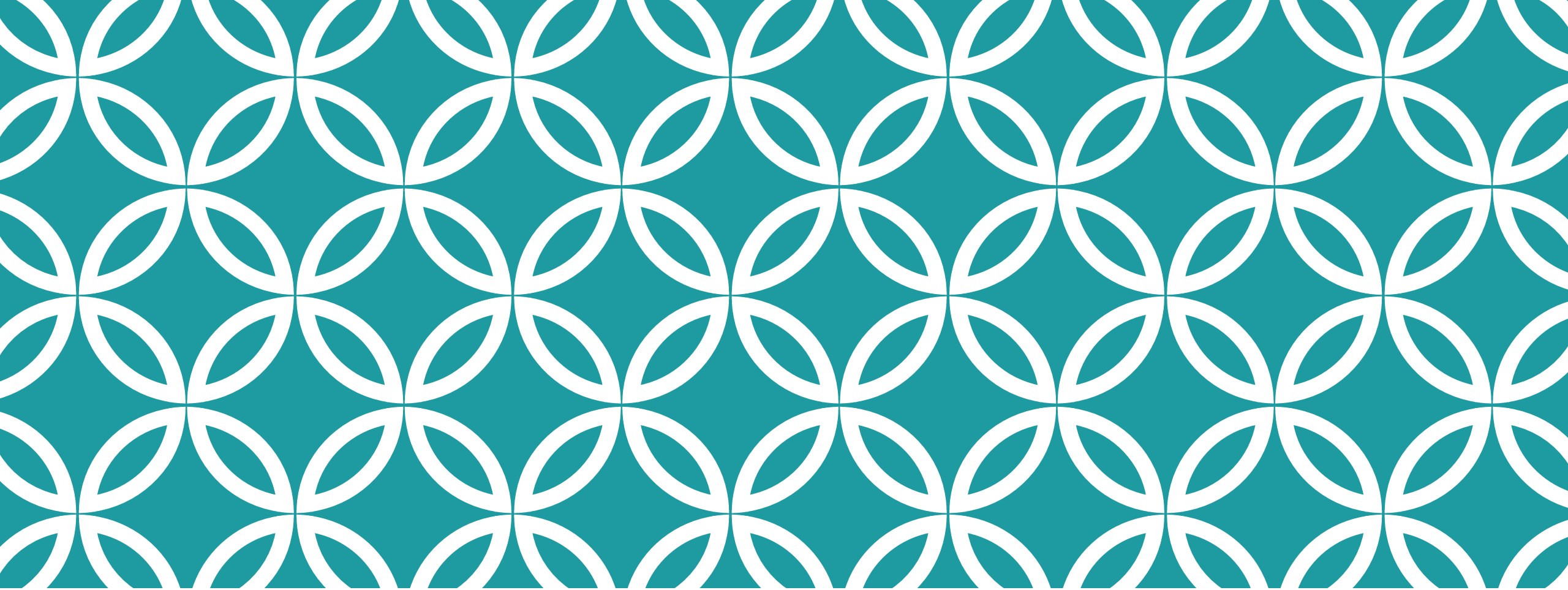
EXAMPLE 3

```
try:
    a = int(input("Enter a:"))
    b = int(input("Enter b:"))
    if b is 0:
        raise ArithmeticError
    else:
        print("a/b = ",a/b)
except ArithmeticError:
    print("The value of b can't be 0")
```

CUSTOM EXCEPTION

The Python allows us to create our exceptions that can be raised from the program and caught using the except clause. However, we suggest you read this section after visiting the Python object and classes.

```
class ErrorInCode(Exception):
    def __init__(self, data):
        self.data = data
    def __str__(self):
        return repr(self.data)

try:
    raise ErrorInCode(2000)
except ErrorInCode as ae:
    print("Received error:", ae.data)
```



CONTINUE IN NEXT UNIT .....