1.

In Python, you can create custom exceptions by defining new classes derived from the built-in Exception class or any of its subclasses. The two most commonly used user-defined exception constraints in Python 3.X are:

ValueError: This exception is raised when a function receives an argument of the correct type but an inappropriate value.
Example:

```
def divide(a, b):
    if b == 0:
        raise ValueError("Division by zero is not allowed.")
    return a / b

try:
    result = divide(10, 0)
except ValueError as ve:
    print(ve)
```

TypeError: This exception is raised when an operation or function is applied to an object of the wrong type.
Example:

```
def multiply(a, b):
    if not isinstance(a, (int, float)) or not isinstance(b, (int, float)):
        raise TypeError("Both arguments must be numeric types.")
    return a * b

try:
    result = multiply(10, "5")
except TypeError as te:
    print(te)
```

2.

When an error (exception) happens in Python, the program looks for a specific code block designed to handle that error. This code block is called an "exception handler." Python checks for exception handlers in the following order:

- First, it looks for a handler in the current part of the code where the error occurred.
- If there's no handler there, it goes to the part of the code that called the current part and checks for a handler there.
- It keeps going up through the different parts of the code (like a ladder) until it finds a handler or reaches the very top of the program.
- If it reaches the top and there's still no handler, the program stops and shows an error message.

The idea is to find a place in the code where the error can be dealt with properly. If there's no such place, the program has to stop to avoid causing unexpected problems.

Let's use a simple example to illustrate how exception handling works with a code

```
def divide(a, b):
    try:
        result = a / b
    except ZeroDivisionError:
        print("Oops! You can't divide by zero.")
    else:
        return result

def main():
    try:
        num1 = int(input("Enter the first number: "))
        num2 = int(input("Enter the second number: "))
        res = divide(num1, num2)
        print("Result:", res)
    except ValueError:
        print("Oops! Please enter valid integer numbers.")
    finally:
        print("Execution completed.")

if __name__ == "__main__":
    main()
```

In this code, we have two functions: **divide()** and **main().** The divide() function attempts to perform division and handles the '**ZeroDivisionError**' if it occurs. The main() function takes two input numbers from the user, calls the divide() function, and handles the '**ValueError'** if the user enters non-numeric values.

3.

Using the __context__ attribute: The **__context__** attribute is a dictionary that can be used to store arbitrary context information about an exception. This information can be accessed

by the '**sys.exc_info()'** function. For example, the following code attaches the current file name and line number to the __context__ attribute of an exception:

```
try:
   raise ValueError("This is a ValueError")
except ValueError as e:
   e.__context__ = {
      "file_name": __file__,
      "line_number": __LINE__,
   }
   print(e)
```

This code will print the following output:

```
ValueError: This is a ValueError
File "attach_context_to_exception.py", line 12, in <module>
   raise ValueError("This is a ValueError")
```

The **file_name** and **line_number** keys in the **__context__** dictionary are used to indicate where the exception occurred.

Using the __cause__ attribute: The **__cause__** attribute can be used to store the direct cause of an exception. This information can be used to track down the source of an exception. For example, the following code raises a '**ValueError'** exception, but it also stores the '**TypeError'** exception that caused it in the __cause__ attribute:

```
try:
   try:
      1 / 0
   except TypeError as e:
      raise ValueError("This is a ValueError") from e
except ValueError as e:
   print(e)
```

This code will print the following output:

```
ValueError: This is a ValueError
Traceback (most recent call last):
  File "attach_context_to_exception.py", line 18, in <module>
    raise ValueError("This is a ValueError") from e
  File "attach_context_to_exception.py", line 16, in <module>
    1 / 0
TypeError: division by zero
```

The **TypeError** exception is the direct cause of the **ValueError** exception.

4.

Here are two methods for specifying the text of an exception object's error message in Python:

Using the __init__ method: The '**__init__'** method is a special method that is called when an exception object is created. This method can be used to specify the error message for the exception object. For example, the following code defines a custom exception called **MyException** and specifies the error message in the __init__ method:

```
class MyException(Exception):
    def __init__(self, message):
        super().__init__(message)
        self.message = message

def raise_my_exception():
    raise MyException("This is my custom exception")

if __name__ == "__main__":
    try:
        raise_my_exception()
    except MyException as e:
        print(e.message)
```

This code will print the following output:

This is my custom exception

Using the set_message method: The '**set_message'** method is a method that is available on all exception objects. This method can be used to change the error message for an exception object after it has been created. For example, the following code raises a **ValueError** exception and then changes the error message using the set_message method:

```
try:
    1 / 0
except ValueError as e:
    e.set_message("This is a custom ValueError message")
    print(e)
```

This code will print the following output:
This is a custom ValueError message

5.

String-based exceptions are no longer used in Python because they are not as informative as exception classes. When you use a string-based exception, the only information that is available to the user is the string itself. This can be difficult to interpret, and it can make it difficult to debug the code.

Exception classes, on the other hand, provide more information about the exception. They can include the following:

- The type of exception
- The message associated with the exception
- The stack trace of the exception

This information can be very helpful for debugging the code and understanding what went wrong.
In addition, exception classes can be used to create custom exceptions. This can be useful for creating exceptions that are specific to your application.
For these reasons, it is generally recommended to use exception classes instead of string-based exceptions.

In the example code you provided, the '**FileNotFoundError'** exception is a good example of an exception class. It provides the following information:

The type of exception: **FileNotFoundError**
The message associated with the exception: **'/path/to/file'**
The stack trace of the exception
This information can be very helpful for debugging the code and understanding why the file could not be found.

If you were to use a string-based exception instead, the only information that would be available would be the string **'[Errno 2]** No such file or directory: **'/path/to/file'**. This is not as informative as the information provided by the **FileNotFoundError** exception class.