CSIT121
Object-Oriented Design and Programming

Dr. Fenghui Ren

School of Computing and Information Technology
University of Wollongong

Lecture 6 outline

- Exception handling
- Cause an exception
- Recover from an exception
- Handle different exception types
- Cleaning up when an exception has occurred
- Creating new types of exception
- Case study

Introduction

- Programs are fragile.
- A valid result sometimes can’t be calculated.
- In old days, programmers have to check the inputs manually.
- However, when the situations become complicated, manually checking is not enough.
- Programs simply crashes due to unexpected usage of the programs.
- We will study exceptions, i.e., error objects that need to be handled during unexpected situations.

Raising exceptions

- In Python, an exception is just an object.
- We can use existing classes or define our own.
- All Python exception classes inherit from a built-in class called ‘BaseException’.
- When unexpected situations happen, exception objects will be created to handle them.
- The easiest way to cause an exception to occur is to do something silly.

Raising exceptions

```
>>> print "hello World"
SyntaxError: Missing parentheses in call to 'print
'. Did you mean print(...)?
>>> print("hello world")
hello world
>>>
```

- The print statement is valid in Python 2, but not in Python 3.
- We have to enclose the arguments in parentheses.
- The print statement raise a SyntaxError, which is an exception.

Common exceptions in Python

- In addition to SyntaxError, Python has the following common built-in exceptions.

```
>>> x=5/0
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    x=5/0
ZeroDivisionError: division by zero
>>>
>>> 1st=[1,2,3]
SyntaxError: invalid decimal literal
>>>
>>> lst=[1,2,3]
>>> print(lst[3])
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    print(lst[3])
IndexError: list index out of range
>>>
>>> lst + 2
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    lst + 2
TypeError: can only concatenate list (not "int") t
o list
```

```
>>> lst.add
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    lst.add
AttributeError: 'list' object has no attribute 'ad
d'
>>>
>>> d = {'a':'hello'}
>>> d['b']
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    d['b']
KeyError: 'b'
>>>
>>> print(this_is_not_a_var)
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    print(this_is_not_a_var)
NameError: name 'this_is_not_a_var' is not defined
```

Raising exceptions in your program

- In the previous examples, errors/exceptions were raised automatically.
- How can we raise exceptions in our program to inform the user or a calling function that the inputs are invalid?

Let's have a look at another example.

```
class EvenOnly(list):
    def append(self, integer):
        if not isinstance(integer, int):
            raise TypeError("Only integers can be added")
        if integer % 2:
            raise ValueError("Only even numbers can be added
        super().append(integer)
```

```
>>> e = EvenOnly()
>>> e.append("a string")
Traceback (most recent call last):
  File "<pyshell#33>", line 1, in <module>
    e.append("a string")
  File "<pyshell#31>", line 4, in append
    raise TypeError("Only integers can be added")
TypeError: Only integers can be added
>>> e.append(3)
Traceback (most recent call last):
  File "<pyshell#34>", line 1, in <module>
    e.append(3)
  File "<pyshell#31>", line 6, in append
    raise ValueError("Only even numbers can be added")
ValueError: Only even numbers can be added
>>> e.append(2)
>>> print(e)
[2]
```

- This class extends the ‘list’ built-in class and overrides the append method.
- Check the input is an instance of the int type and the input is an even number.
- If not, use the keyword ‘raise’ to cause an exception.
- The keyword ‘raise’ followed by the built-in exception object to be created, i.e., TypeError and ValueError object.
- We can also raise an object of new Exceptions classes created by ourselves.

The effects of an exception

- When an exception is raised, it stops program execution immediately.
- Any lines that were supposed to run after the exception is raised are not executed.
- Furthermore, if we have a function that calls another function that raises an exception, nothing is executed in the first function after the point where the second function was called.
- Raising an exception stops all execution right up through the function call stack until it is either handled or forces the interpreter to exit.

The effects of an exception

- Raise an exception

```
>>> def no_return():
...     print("I am about to raise an exception")
...     raise Exception("This is always raised")
...     print("This line will never execute")
...     return "I won't be returned"
...
>>> no_return()
I am about to raise an exception
Traceback (most recent call last):
  File "<pyshell#43>", line 1, in <module>
    no_return()
  File "<pyshell#42>", line 3, in no_return
    raise Exception("This is always raised")
Exception: This is always raised
```

- Raise an exception through a function call

```
def call_exceptor():
    print("Call_exceptor starts here...")
    no_return()
    print("an exception was raised...")
    print("...so there lines don't run")


call_exceptor()
Call_exceptor starts here...
I am about to raise an exception
Traceback (most recent call last):
  File "<pyshell#52>", line 1, in <module>
    call_exceptor()
  File "<pyshell#51>", line 3, in call_exceptor
    no_return()
  File "<pyshell#42>", line 3, in no_return
    raise Exception("This is always raised")
Exception: This is always raised
```

Handling exceptions

- The other side of the exception coin is how our code should react to or recover from it.
- We handle exceptions by wrapping any code that might throw one inside a ‘try…except’ clause.
- The basic syntax like this:

```
>>> try:
...     no_return()
... except:
...     print("I caught an exception")
...     print("Executed after the exception")
...
...
I am about to raise an exception
I caught an exception
Executed after the exception
```

Handling exceptions

- What if we just want to catch a particular type of exception?
- We should specify the exception needs to be caught.

```
>>> def funny_division(divider):
...     try:
...         return 100/divider
...     except ZeroDivisionError:
...         return "Zero is not a good idea"
...
...
>>> print(funny_division(0))
Zero is not a good idea
>>> print(funny_division(50.0))
2.0
>>> print(funny_division("hello"))
Traceback (most recent call last):
  File "<pyshell#82>", line 1, in <module>
    print(funny_division("hello"))
  File "<pyshell#79>", line 3, in funny_division
    return 100/divider
TypeError: unsupported operand type(s) for /: 'int' and 'str'
```

- The last exception is not caught because it is not a 'ZeroDivisionError' exception.
- Use the 'except Exception:' if you want to catch all exception types. Because all exception types are the subclasses of 'Exception' class
- 'BaseException' will also cover the system-level exceptions which are very rare.

Handling exceptions

- We can catch two or more different exceptions and handle them with the same code.
- In this example, the number 0 and the string are both caught by the except clause, but the number 13 is not caught because it is a ValueError, which is not included in the types of exception being handled.
- What if we want to catch different exceptions and do different thing?
- What if we want to do something with an exception and then allow it to continue to bubble up to the parent function after?

```
>>> def funny_division2(divider):
...     try:
...         if divider == 13:
...             raise ValueError("13 is an unlucky number")
...         return 100/divider
...     except (ZeroDivisionError, TypeError):
...         return "Enter a number other than zero"
...
...
>>> for val in (0, "hello", 50.0, 13):
...     print("Testing {}: ".format(val))
...     print(funny_division2(val))
...
...
Testing 0:
Enter a number other than zero
Testing hello:
Enter a number other than zero
Testing 50.0:
2.0
Testing 13:
Traceback (most recent call last):
  File "<pyshell#99>", line 3, in <module>
    print(funny_division2(val))
  File "<pyshell#97>", line 4, in funny_division2
    raise ValueError("13 is an unlucky number")
ValueError: 13 is an unlucky number
```

Handling exceptions

```
>>> def funny_division3(divider):
...     try:
...         if divider == 13:
...             raise ValueError("13 is an unlucky number")
...         return 100 / divider
...     except ZeroDivisionError:
...         return "Enter a number other than zero"
...     except TypeError:
...         return "Enter a numerical value"
...     except ValueError:
...         print("No, No, not 13!")
...         raise
...
...
>>> for val in (0, "hello", 50.0, 13):
...     print("Testing {}: ".format(val))
...     print(funny_division3(val))
...
...
Testing 0:
Enter a number other than zero
Testing hello:
Enter a numerical value
Testing 50.0:
2.0
Testing 13:
No, No, not 13!
Traceback (most recent call last):
  File "<pyshell#114>", line 3, in <module>
    print(funny_division3(val))
  File "<pyshell#112>", line 4, in funny_division3
    raise ValueError("13 is an unlucky number")
ValueError: 13 is an unlucky number
```

- If we stack exception like this, only the first matching clause will be run, even if more than one of them fits.
- For example, if we catch Exception before we catch ZeroDivisionError, then only the Exception handler will be executed because ZeroDivisionError is an Exception by inheritance.
- So we should place specific exception handlers before general exception handlers.
- Usually, we catch Exception after catching all the specific exceptions.

Handling exceptions

Another two optional keywords in the exception handling:

- 'else' clause (optional): specifying the codes will be executed if no exception happens.
- 'finally' clause (optional): specifying the codes will be executed no matter exceptions happen or not.

```
try:
  # the try block

except ExceptionA as e:
  # handle ExceptionA in this block

except ExceptionB as e:
  # handle ExceptionB in this block

except ExceptionC as e:
  # handle ExceptionC in this block
else:
  # if there is no exceptions
```

```
try:
  # the try block

except ExceptionA as e:
  # handle ExceptionA in this block

except ExceptionB as e:
  # handle ExceptionB in this block

except ExceptionC as e:
  # handle ExceptionC in this block
finally:
  # exceptions or no exception
  # this block always get executed
```

Handling exceptions

- The following example randomly picks an exception to throw and raises it.

```
1 import random
2
3 some_exceptions = [ValueError, TypeError, IndexError, None]
4
5 try:
6     choice = random.choice(some_exceptions)
7     print("raising {}".format(choice))
8     if choice:
9         raise choice("An error")
10 except ValueError:
11     print("Caught a ValueError")
12 except TypeError:
13     print("Caught a TypeError")
14 except Exception as e:
15     print("Caught some other error: %s" % (e.__class__.__name__))
16 else:
17     print("This code called if there is no exception")
18 finally:
19     print("This cleanup code is always called")
```

```
raising <class 'IndexError'>
Caught some other error: IndexError
This cleanup code is always called
```

```
raising <class 'ValueError'>
Caught a ValueError
This cleanup code is always called
```

```
raising None
This code called if there is no exception
This cleanup code is always called
```

Handling exceptions

The ‘finally’ clause is important because it will be executed no matter what happens. It can be used to

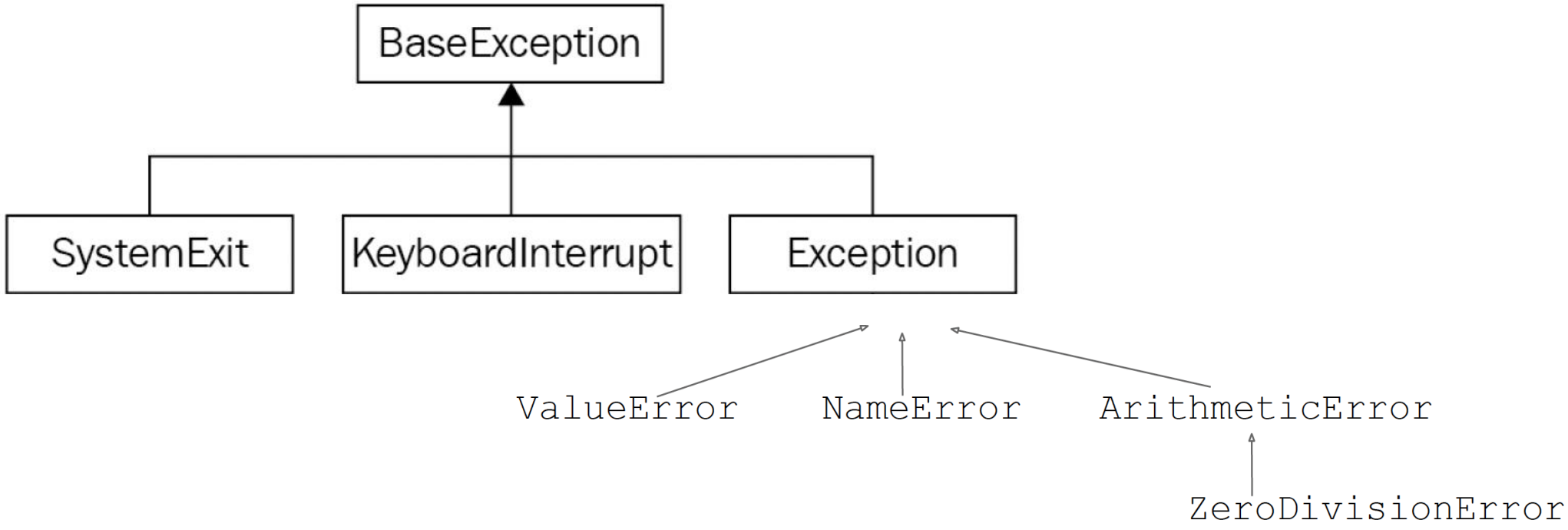
- Cleaning up an open database connection
- Closing an open file
- Sending a closing handshake over the network
- Executing some codes before the ‘return’ statement

```
>>> def funny_division3(divider):
...     try:
...         if divider == 13:
...             raise ValueError("13 is an unlucky number")
...         return 100 / divider
...     except ZeroDivisionError:
...         return "Enter a number other than zero"
...     except TypeError:
...         return "Enter a numerical value"
...     except ValueError:
...         print("No, No, not 13!")
...         raise
...     finally:
...         print("This line is displayed before return")
...
...
>>> print(funny_division3(0))
This line is displayed before return
Enter a number other than zero
```

The exception hierarchy

- Most exceptions are subclasses of the Exception class.
- Exception class inherits from BaseException class.
- The SystemExit class is a key built-in exception class derived directly from BaseException class. It is used to handle the exception whenever the program exists naturally. Then the exception is raised to allow us to clean up the code before the program ultimately exits.
- The KeyboardInterrupt class is another key built-in exception class derived from BaseException class. It is thrown when the user explicitly interrupts program execution with an OS-dependent key combination (normally Ctrl+c). As SystemExit, it handle any cleanup tasks before the program exits.

The exception hierarchy



When we use the ‘except:’ clause without specifying any type of exception, it will catch all subclasses of BaseException. If you don’t want to catch the two specific exceptions, use ‘except: Exception’ clause.

Defining our own exceptions

- If we find that none of the built-in exceptions are suitable for our program, we can define new exceptions for our own.
- Normally, the new exceptions inherit from the ’Exception’ class.
- Then the ‘__init__’ method of the ‘Exception’ class will handle the default arguments past to the new exception objects.

```
>>> class InvalidWithdrawal(Exception):
...     pass
...
>>> raise InvalidWithdrawal("You don't have enough balance in your a
ccount")
Traceback (most recent call last):
  File "<pyshell#142>", line 1, in <module>
    raise InvalidWithdrawal("You don't have enough balance in yo
ur account")
InvalidWithdrawal: You don't have enough balance in your account
```

Defining our own exceptions

- Of course, we can also override the __init__ method to accepts extra arguments.
- We can define new methods for our new exception class.

```
class InvalidWithdrawal(Exception):
    def __init__(self, balance, amount):
        super().__init__(f"account doesn't have ${amount}")
        self.amount=amount
        self.balance=balance

    def overage(self):
        return self.amount - self.balance
```

```
>>> try:
...     raise InvalidWithdrawal(25, 50)
... except InvalidWithdrawal as e:
...     print(f"I am sorry, but your widthdrawal is more than your b
alance by ${e.overage()}")
...
...
I am sorry, but your widthdrawal is more than your balance by $2
5
```

More about exceptions

```
def divide_with_exception(number, divisor):
    try:
        print(f"{number} / {divisor} = {number / divisor}")
    except ZeroDivisionError:
        print("You can't divide by zero")

def divide_with_if(number, divisor):
    if divisor == 0:
        print("You can't divide by zero")
    else:
        print(f"{number} / {divisor} = {number / divisor}")
```

- The above two functions behave identically. Which is better? The if statement or the try statement?
- We can use the if statement to check particular/known conditions, but how about other unknown situations?
- The try statement will catch all unexpected situations.

Chained exceptions

- Sometimes a method responds to an exception by throwing another exception type that is specific to the current application.
- Chained exceptions enable an exception object to maintain the complete stack-trace information from the original exception.

```
#chained exceptions
class ChainedExceptions:

    def method1(self):
        try:
            self.method2()
        except Exception as e:
            raise Exception("Exception thrown in method1")

    def method2(self):
        try:
            self.method3()
        except Exception as e:
            raise Exception("Exception thrown in method2")

    def method3(self):
        raise Exception("Exception thrown in method3")
```

```
chained_exceptions= ChainedExceptions()
chained_exceptions.method1()
Traceback (most recent call last):
  File "<pyshell#187>", line 12, in method2
    self.method3()
  File "<pyshell#187>", line 17, in method3
    raise Exception("Exception thrown in method3")
Exception: Exception thrown in method3

During handling of the above exception, another exception
occurred:

Traceback (most recent call last):
  File "<pyshell#187>", line 6, in method1
    self.method2()
  File "<pyshell#187>", line 14, in method2
    raise Exception("Exception thrown in method2")
Exception: Exception thrown in method2

During handling of the above exception, another exception
occurred:

Traceback (most recent call last):
  File "<pyshell#189>", line 1, in <module>
    chained_exceptions.method1()
  File "<pyshell#187>", line 8, in method1
    raise Exception("Exception thrown in method1")
Exception: Exception thrown in method1
```

Examples: keep asking the user

- Write a program to ask the user to repeat a particular behaviour until the action is satisfied.

```
while True:
    try:
        user_input = input ("Enter a positive integer: ")

        try:
            number = int(user_input)
        except:
            raise ValueError("Invalid integer format")

        if(number<=0):
            raise ValueError("Input must be a positive number")

        print("You have enterred {}".format(number))
        break

    except ValueError as e:
        print("Error: "+str(e))
```

```
Enter a positive integer: abc
Error: Invalid integer format
Enter a positive integer: -10
Error: Input must be a positive number
Enter a positive integer: xyz
Error: Invalid integer format
Enter a positive integer: 0
Error: Input must be a positive number
Enter a positive integer: 5
You have enterred 5
```

Examples: medical thermometer

```
class TemperatureException(Exception):
    def __init__(self, degrees):
        self.degrees=degrees

    def getMessage(self):
        return f"The temperature {self.degrees} isn't in the normal range."

class MedicalThermometer:
    def measure(self):
        while True:
            try:
                temperature_input = input ("Enter a temperature (0 for exit):")

                try:
                    number=int(temperature_input)
                except:
                    raise ValueError("Invalid temperature input")

                if(number>43 or 0<number<14):
                    raise TemperatureException(number)
                elif number>=38:
                    print("Fever!")
                elif 0<number<35:
                    print("Hypothermia!")
                elif number==0:
                    break
                else:
                    print("Normal.")

            except ValueError as e:
                print("Error: "+str(e))
            except TemperatureException as e:
                print(e.getMessage())
```

```
>>> medical_thermometer=MedicalThermometer()
>>> medical_thermometer.measure()
Enter a temperature (0 for exit):hello
Error: Invalid temperature input
Enter a temperature (0 for exit):35
Normal.
Enter a temperature (0 for exit):38
Fever!
Enter a temperature (0 for exit):30
Hypothermia!
Enter a temperature (0 for exit):50
The temperature 50 isn't in the normal range.
Enter a temperature (0 for exit):0
```

Exception notes

Exceptions are a good mechanism to flag input data that does not pass validation: Normally, regular input data should pass validation, so if it doesn't pass, then this should be an exception, not the rule.

That said, it is good software engineering and security practice to allow for input data that may fail validation. This is especially so if the source of the input data is a user or system that you have no control over.

Input data that fails validation could be supplied by:

- A malicious user trying to subvert your system
- Another system whose data output has changed, e.g., as a result of a configuration change or software upgrade, and whose format is no longer compatible with yours.

Validation allows you to handle such situations in a way you control – as opposed to leaving it to your code to do… whatever!

Exception notes

- We don’t need to rely on built-in exceptions raised by code that comes with Python – we can throw any exception ourselves in our own code.

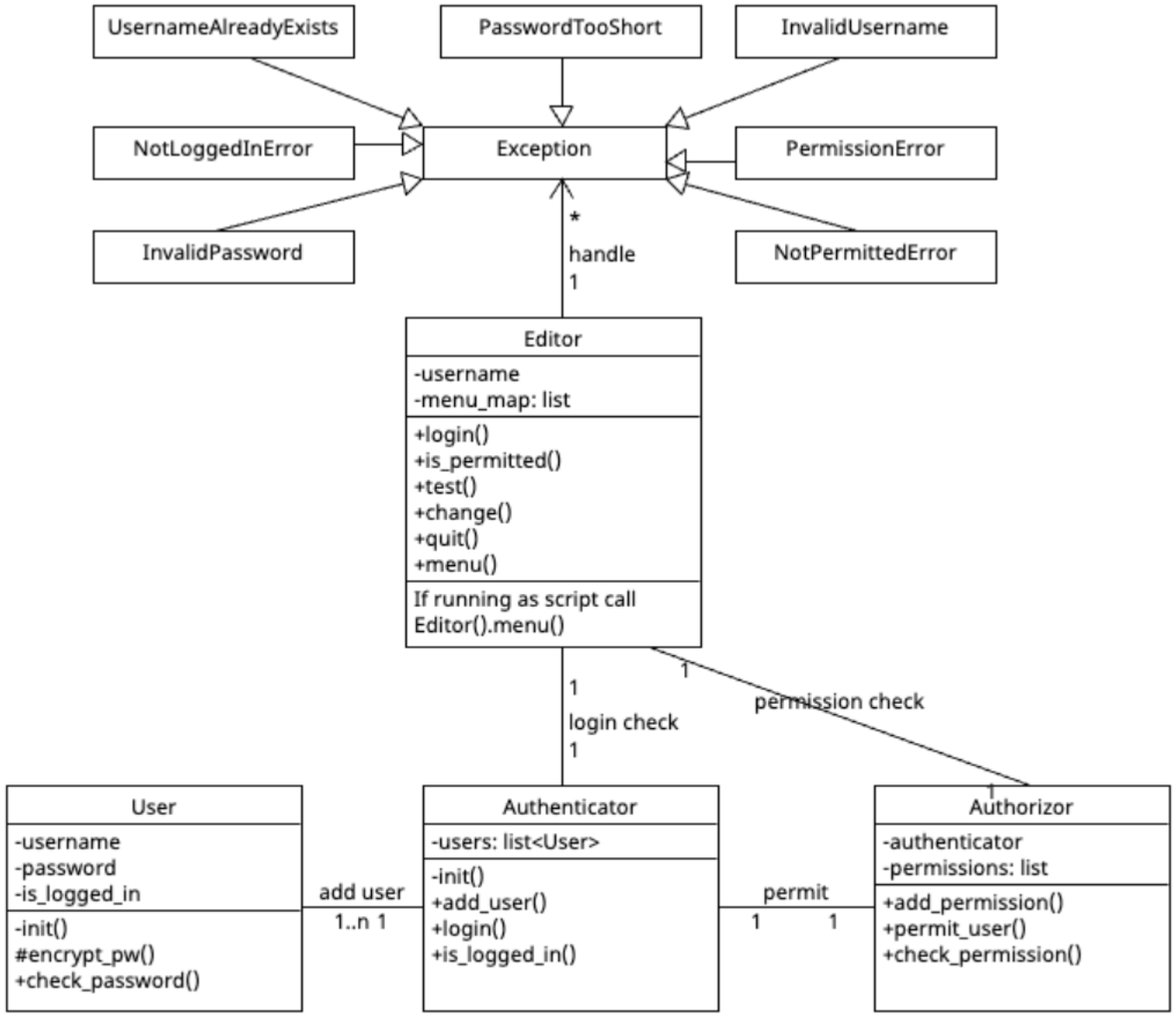
- We can also create our own exception classes, instantiate and raise them.

- The finally clause is an addition to the try blocks we’ve seen so far and contains statements that we always want to have executed regardless of whether there is an exception or not.

Case study: a simple authentication and authorization system

- Authentication is the process of ensuring a user is really the person they say they are. We'll use a username and private password combination.
- Authorization, on the other hand, is all about determining whether a given (authenticated) user is permitted to perform a specific action, such as read, write, execute files.
- A simple administrative features to add users.

Case study: UML class diagram



Case study: UML class diagram

```
class AuthException(Exception):
    def __init__(self, username, user=None):
        super().__init__(username, user)
        self.username = username
        self.user = user


class UsernameAlreadyExists(AuthException):
    pass


class PasswordTooShort(AuthException):
    pass


class InvalidUsername(AuthException):
    pass


class InvalidPassword(AuthException):
    pass


class PermissionError(Exception):
    pass


class NotLoggedInError(AuthException):
    pass


class NotPermittedError(AuthException):
    pass
```

```
class User:
    def __init__(self, username, password):
        """Create a new user object. The password
        will be encrypted before storing."""
        self.username = username
        self.password = self._encrypt_pw(password)
        self.is_logged_in = False

    def _encrypt_pw(self, password):
        """Encrypt the password with the username and return
        the sha digest."""
        hash_string = self.username + password
        hash_string = hash_string.encode("utf8")
        return hashlib.sha256(hash_string).hexdigest()

    def check_password(self, password):
        """Return True if the password is valid for this
        user, false otherwise."""
        encrypted = self._encrypt_pw(password)
        return encrypted == self.password
```

Case study: UML class diagram

```
class Authenticator:
    def __init__(self):
        """Construct an authenticator to manage
        users logging in and out."""
        self.users = {}

    def add_user(self, username, password):
        if username in self.users:
            raise UsernameAlreadyExists(username)
        if len(password) < 6:
            raise PasswordTooShort(username)
        self.users[username] = User(username, password)

    def login(self, username, password):
        try:
            user = self.users[username]
        except KeyError:
            raise InvalidUsername(username)

        if not user.check_password(password):
            raise InvalidPassword(username, user)

        user.is_logged_in = True
        return True

    def is_logged_in(self, username):
        if username in self.users:
            return self.users[username].is_logged_in
        return False
```

```
class Authorizor:
    def __init__(self, authenticator):
        self.authenticator = authenticator
        self.permissions = {}

    def add_permission(self, perm_name):
        """Create a new permission that users
        can be added to"""
        try:
            perm_set = self.permissions[perm_name]
        except KeyError:
            self.permissions[perm_name] = set()
        else:
            raise PermissionError("Permission Exists")

    def permit_user(self, perm_name, username):
        """Grant the given permission to the user"""
        try:
            perm_set = self.permissions[perm_name]
        except KeyError:
            raise PermissionError("Permission does not exist")
        else:
            if username not in self.authenticator.users:
                raise InvalidUsername(username)
            perm_set.add(username)

    def check_permission(self, perm_name, username):
        if not self.authenticator.is_logged_in(username):
            raise NotLoggedInError(username)
        try:
            perm_set = self.permissions[perm_name]
        except KeyError:
            raise PermissionError("Permission does not exist")
        else:
            if username not in perm_set:
                raise NotPermittedError(username)
            else:
                return True
```

Case study: UML class diagram

```
class Editor:
    def __init__(self):
        self.username = None
        self.menu_map = {
            "login": self.login,
            "test": self.test,
            "change": self.change,
            "quit": self.quit,
        }

    def login(self):
        logged_in = False
        while not logged_in:
            username = input("username: ")
            password = input("password: ")
            try:
                logged_in = auth.authenticator.login(username, password)
            except auth.InvalidUsername:
                print("Sorry, that username does not exist")
            except auth.InvalidPassword:
                print("Sorry, incorrect password")
            else:
                self.username = username

    def is_permitted(self, permission):
        try:
            auth.authorizor.check_permission(permission, self.username)
        except auth.NotLoggedInError as e:
            print("{} is not logged in".format(e.username))
            return False
        except auth.NotPermittedError as e:
            print("{} cannot {}".format(e.username, permission))
            return False
        else:
            return True

    def test(self):
        if self.is_permitted("test program"):
            print("Testing program now...")

    def change(self):
        if self.is_permitted("change program"):
            print("Changing program now...")

    def quit(self):
        raise SystemExit()

    def menu(self):
        try:
            answer = ""
            while True:
                print(
                    """
Please enter a command:
\tlogin\tLogin
\ttest\tTest the program
\tchange\tChange the program
\tquit\tQuit
"""
                )
                answer = input("enter a command: ").lower()
                try:
                    func = self.menu_map[answer]
                except KeyError:
                    print("{} is not a valid option".format(answer))
                else:
                    func()
        finally:
            print("Thank you for testing the auth module")
```

Case study: UML class diagram

```
# Set up a test user and permission
auth.authenticator.add_user("joe", "joepassword")
auth.authorizor.add_permission("test program")
auth.authorizor.add_permission("change program")
auth.authorizor.permit_user("test program", "joe")
```

An execution without exceptions

```
Please enter a command:
        login    Login
        test     Test the program
        change   Change the program
        quit     Quit

enter a command: login
username: joe
password: joepassword

Please enter a command:
        login    Login
        test     Test the program
        change   Change the program
        quit     Quit

enter a command: test
Testing program now...

Please enter a command:
        login    Login
        test     Test the program
        change   Change the program
        quit     Quit

enter a command: change
joe cannot change program

Please enter a command:
        login    Login
        test     Test the program
        change   Change the program
        quit     Quit

enter a command: quit
Thank you for testing the auth module
```

An execution with exceptions

```
Please enter a command:
        login    Login
        test     Test the program
        change   Change the program
        quit     Quit

enter a command: test
None is not logged in

Please enter a command:
        login    Login
        test     Test the program
        change   Change the program
        quit     Quit

enter a command: change
None is not logged in

Please enter a command:
        login    Login
        test     Test the program
        change   Change the program
        quit     Quit

enter a command: execute
execute is not a valid option

Please enter a command:
        login    Login
        test     Test the program
        change   Change the program
        quit     Quit

enter a command: login
username: a
password: a
Sorry, that username does not exist
username: joe
password: joe
Sorry, incorrect password
```

Suggested reading

Python 3 Object-Oriented Programming

- Chapter 4: Expecting the Unexpected

Python

- https://www.python.org/