Object Oriented Programming in Python
Lesson 3
Exception handling

Outline:

- How to cause an exception to occur
- How to recover when an exception has occurred
- How to handle different exception types in different ways
- Cleaning up when an exception has occurred
- Creating new types of exception
- Using the exception syntax for flow control

In OOP, an exception is an object which is created when an abnormal event happens during the execution of the program. There are many different exception classes available, and we can easily define more of our own. The one thing they all have in common is that they inherit from a built-in class called BaseException.

Types of errors:
- SyntaxError : the Python interpreter detects syntactical errors. We do not have to handle them inside the program's flow of control.
- Application errors: we can handle them inside the program's flow of control. The following error classes are all extensions of the Exception class (which is an extension of the BaseException class).

```
>>> x = 5 / 0
ZeroDivisionError: int division or modulo by zero

>>> lst = [1,2,3]
>>> print(lst[3])
IndexError: list index out of range

>>> lst + 2
TypeError: can only concatenate list (not "int") to list

>>> lst.add
AttributeError: 'list' object has no attribute 'add'

>>> d = {'a': 'hello'}
>>> d['b']
KeyError: 'b'

>>> print(this_is_not_a_var)
NameError: name 'this_is_not_a_var' is not defined
```

Raising an exception

```
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)
```

This class extends the list built-in and overrides the append method to check two conditions that ensure the item is an even integer. We first check if the input is an instance of the int type, and then use the modulus operator to ensure it is divisible by two. If either of the two conditions is not met, the raise keyword causes an exception to occur. The raise keyword is simply followed by the object being raised as an exception.

Raising an exception stops all program execution immediately, right up through the function call stack until it is either handled or forces the interpreter to exit with an error message.

Handling an exception

We handle exceptions by wrapping any code that might throw one (whether it is exception code itself, or a call to any function or method that may have an exception raised inside it) inside a *try...except* clause. The most basic syntax looks like this:

```
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"

try:
        no_return()
except:
        print("I caught an exception")

print("executed after the exception")
```

The above code catches any type of exception. If our code can raise more exceptions, the above code cannot make a distinction between them. We can specify the type of exception to be handled in the except clause with the following syntax:

```
def division(divider):
        try:
                return 100 / divider
        except ZeroDivisionError:
                return "Zero is not a good idea!"

print(division(0))              -> Zero is not a good idea!
print(division(50.0))           -> 2.0
print(division("hello"))        -> TypeError: unsupported operand type(s) for /: 'int' and 'str'.
```

We can catch two or more different exceptions and handle them with the same code.

```
def division(divider):
        try:
                return 100 / divider
        except (ZeroDivisionError, TypeError):
                return "Enter a number other than zero"
```

We can write separate exception handler clauses to each exception type.

```
def division(number):
    try:
        if number == 13:
            raise ValueError("13 is an unlucky number")
        return 100 / number
    except ZeroDivisionError:
        return "Enter a number other than zero"
    except TypeError:
        return "Enter a numerical value"
    except ValueError:
        return "No, not 13!"
```

Within an exception handler clause we can use the raise command to raise the exception again and write the original stack trace on the console.

```
def division(number):
    try:
        return 100 / number
    except ZeroDivisionError:
        return "Enter a number other than zero"
    except TypeError:
        print("Enter a numerical value")
        raise
```

If we stack exception clauses like we did in the preceding example, only the first matching clause will be run, even if more than one of them fits. Since exceptions are objects, beware of inheritance between exception types. For example, if we catch Exception before we catch TypeError, then only the Exception handler will be executed, because TypeError is an Exception by inheritance. This can come in handy in cases where we want to handle some exceptions specifically,cand then handle all remaining exceptions as a more general case. We can simply catch Exception after catching all the specific exceptions and handle the general case there. As a rule of thumb, the order of the except clauses normally goes from most specific to most generic.

```
def division(number):
    try:
        if number == 13:
            raise ValueError("13 is an unlucky number")
        return 100 / number
    except TypeError:
        return "Enter a numerical value"
    except Exception:
        return "Enter a number other than 0 or 13"
```

Sometimes, when we catch an exception, we need a reference to the exception object itself. This most often happens when we define our own exceptions with custom arguments, but can also be relevant with standard exceptions. Most exception classes accept a set of arguments in their constructor, and we might want to access those attributes in the exception handler. If we define our own exception class, we can even call custom methods on it when we catch it. The syntax for capturing an exception as a variable uses the 'as' keyword:

```
try:
        raise ValueError("This is an argument")
except ValueError as e:
        print("The exception arguments were", e.args)
```

We can execute code regardless of whether or not an exception has occurred using the **finally** clause. This is extremely useful when we need to perform certain tasks after our code has finished running (even if an exception has occurred). Common example codes here are cleaning up an open database connection, or closing an open file.

```
try:
        ...
except:
        ...
finally:
        ...
```

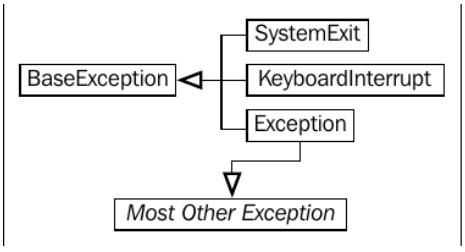
The finally clause is also very important when we execute a return statement from inside a try clause. The finally clause will still be executed before the value is returned.

The code that should be executed only when no exception is raised, should be placed in the **else** clause.

```
try:
        ...
except:
        ...
else:
        ...
finally:
        ...
```

The code that goes after the entire try-except block will be executed even if an exception is caught and handled.

Exception hierarchy



When we use the except: clause without specifying any type of exception, it will catch all subclasses of BaseException; which is to say, it will catch all exceptions.

Creating custom exception

We can define new exceptions of our own inherited from the Exception class. The name of the class is usually designed to communicate what went wrong, and we can provide arbitrary arguments in the initializer to include additional information.

Simple exception:

```
class InvalidWithdrawal(Exception):
    pass
```

Raising the exception:

```
raise InvalidWithdrawal("You don't have $50 in your account")
```

We are able to pass an arbitrary number of arguments into the exception. Often a string message is used, but any object that might be useful in a later exception handler can be stored. The **Exception.__init__** method is designed to accept any arguments and store them as a tuple in an attribute named **args**.
For example, here is an exception whose initializer accepts the current balance and the amount the user wanted to withdraw. In addition, it adds a method to calculate how overdrawn the request was:

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

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

Usage:

```
try:
    raise InvalidWithdrawal(25, 50)
except InvalidWithdrawal as e:
    print("Your withdrawal is more than your balance by ${}".format(e.overage()))
```

Note:
Exceptional situations can be handled in if...else constructs, too. But it is not advised to do this.

```
def divide_with_exception(number, divisor):
    try:
        print("{} / {} = {}".format(number, divisor, number / divisor * 1.0))
    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("{} / {} = {}".format(number, divisor, number / divisor * 1.0))
```

Python programmers tend to follow a model of 'Ask forgiveness rather than permission', which is to say, they execute code and then deal with anything that goes wrong. The main reason for this is that it shouldn't be necessary to burn CPU cycles looking for an unusual situation that is not going to arise in the normal path through the code. Therefore, it is wise to use exceptions for exceptional circumstances, even if those circumstances are only a little bit exceptional. Furthermore, exception syntax is also effective for flow control. Like an if statement, exceptions can be used for decision making, branching, and message passing.