Chapter 04 - Exception Handling

April 4, 2021

0.1 Raising an Unexpected

- `raise TypeError("")` or `raise ValueError("")`
- you can extend built in methods like the append method

[1]:
```
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)
```

0.2 The Effects of an Exception

- exceptions stop program execution immediately

0.3 Handling Exceptions

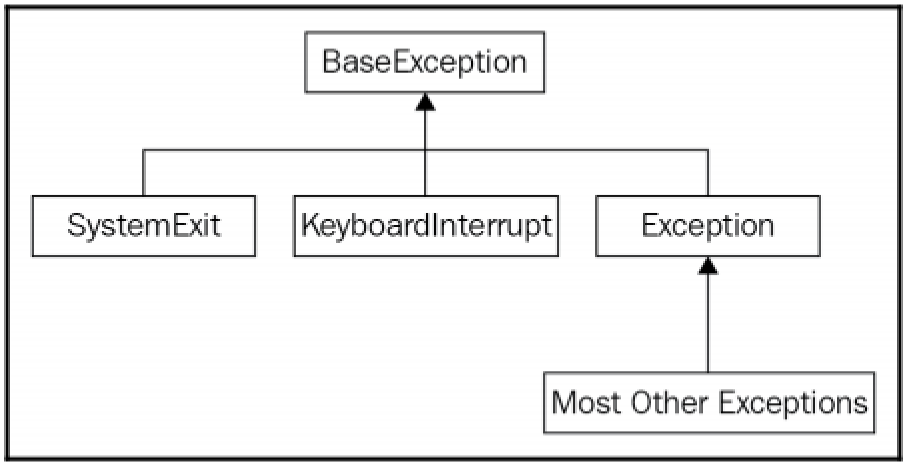
- use `try/except`
- dont just use `except` but with some exception
- you can also use `finally`
  - to clean up an open database connection
  - closing an open file
  - sending a closing handshake over the network
- if you have an `else` block, it will not be executed if you have an `exception`
-

0.4 The Exception Hierarchy

- `Exception` inheir form a class called `BaseException`
- all exceptions must extend the `BaseException` class or one of its subclasses

- there are two key built-in exception classes: `systemExit` and `KeyboardInterrupt` that derive directly from `BaseException` instead of `Exception`
- `SystemExit` exception is raised whenever the program exits naturally, typically because we called the `sys.exit` function somewhere in our code
  - the exception is designed to allow us to clean up code before the program ultimately exits
- `KeyboardInterrupt` exception is common in command-line programs

    – occurs when user stops programs
    – always responds by stopping the program
    – it should handle any cleanup tasks inside the finally block



- 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, including the two special ones
- using `except` without those other two may by mistake call `systemExit` or `KeybordInterrupt`

0.5 Defining our Own Exceptions

[6]:
```python
class InvalidWithdrawl(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

# raise InvalidWithdrawl(25, 50)

try:
    raise InvalidWithdrawl(25, 50)
except InvalidWithdrawl as e:
    print("I am sorry, but your withdrawl is "
          "more than your balance by "
          f"${e.overage()}"
         )
```

```
I am sorry, but your withdrawl is more than your balance by $25
```

- beauty of exceptions comes into light if you are creating your own framework, library or API that is intended for access by other programmers

- python programmers tend to *ask forgiveness rather than permission* and not waste CPU cycle looking ofr an unusual situation that might not arise in the normal path
- just try and exception and go from there
- exceptions could be used to pass information to the backend