Tratando erros

Exceções

Principais conceitos a serem abordados

- Programação defensiva
  - Antecipar o que pode sair errado
- Lançamento e tratamento de exceção
- Informe de erro

Algumas causas das situações de erros lógicos

- Implementação incorreta.
  - Não atende à especificação.
- Solicitação de objeto inapropriado.
  - Por exemplo, índice inválido.
- Estado do objeto inconsistente ou inadequado.
  - Por exemplo, surgindo devido à extensão de classe.

Nem sempre erro do programador

- Erros surgem freqüentemente do ambiente:
  - URL incorreto inserido; e
  - interrupção da rede.
- Processamento de arquivos é particularmente propenso a erros:
  - arquivos ausentes; e
  - falta de permissões apropriadas.

Programação defensiva

- Interação cliente—servidor.
  - Um servidor deve assumir que os clientes são bem-comportados?
  - Ou ele deve assumir que os clientes são potencialmente hostis?
- Diferenças significativas na implementação são requeridas.

Questões a serem resolvidas

- Qual é o número de verificações por um servidor nas chamadas de método?
- Como informar erros?
- Como um cliente pode antecipar uma falha?
- Como um cliente deve lidar com uma falha?

Projeto exemplo: catálogo de endereços

- Aplicação que armazena detalhes de contato pessoal – nome, endereço e número de telefone – para um número arbitrário de pessoas.
- Os detalhes de contato são indexados no catálogo de endereços pelo nome e pelo número de telefone
- Classes AddressBook e ContactDetails

Um exemplo

- Na criação de um objeto `AddressBook`.
- Na tentativa de remover uma entrada.
- Resulta em um erro em tempo de execução.
  - De quem é a ‘falha’?
- Antecipação e prevenção são preferíveis a apontar um culpado.

Valores dos argumentos

- Argumentos representam uma séria ‘vulnerabilidade’ para um objeto servidor.
  - Argumentos do construtor inicializam o estado.
  - Argumentos do método contribuem freqüentemente com o comportamento.
- Verificação de argumento é uma das medidas defensivas.

Verificando a chave

```
def removeDetails(key):

    if(keyInUse(key)):
        details = book.get(key)
        book.remove(details.getName())
        book.remove(details.getPhone())
        numberOfEntries -= 1
```

Informe de erro do servidor

- Como informar argumentos inválidos?
  - No usuário?
    - Há usuários humanos?
    - Eles podem resolver o problema?
  - No objeto cliente?
    - Retorna um valor de diagnóstico.
    - *Lança uma exceção.*

Retornando um diagnóstico

```
def removeDetails(key):
    if(keyInUse(key)):
        ContactDetails details = book.get(key)
        book.remove(details.getName())
        book.remove(details.getPhone())
        numberOfEntries -= 1
        return True

    else:
        return False
```

Respostas do cliente

- Testar o valor de retorno.
  - Tente recuperar o erro.
  - Evite a falha do programa.
- Ignorar o valor de retorno.
  - Não pode ser evitado.
  - Possibilidade de levar a uma falha do programa.
- Exceções são preferíveis.

Princípios do lançamento de exceções

- Um recurso especial de linguagem.
- Nenhum valor de retorno ‘especial’ é necessário.
- Erros não podem ser ignorados no cliente.
  - O fluxo normal de controle é interrompido.
- Ações específicas de recuperação são encorajadas.

Lançando uma exceção (1)

- A instrução **raise** permite ao programador forçar a ocorrência de um determinado tipo de exceção.
- O argumento de **raise** indica a exceção a ser levantada.

Lançando uma exceção (2)

- Um objeto de exceção é construído:
  - `ValueError("...");`
- O objeto exceção é lançado:
  - `raise …`
- `raise ValueError("aqui tem um valor inválido")`

Lançando uma exceção (3)

```
“””
 * Pesquisa um nome ou um número de telefone e retorna
 * os detalhes do contato correspondentes.
 * @param key O nome ou número a ser pesquisado.
 * @return Os detalhes correspondentes a chave, ou null
 *         se não houver nenhuma correspondência.
 * @raise KeyError se a chave for null.
“””
def getDetails(key):

    if key == None:
        raise KeyError(“None key in getDetails")

    return book.get(key)
```

Categorias de exceção

- Exceções verificadas:
  - subclasse de `Exception`;
  - utilizadas para falhas iniciais; e
  - onde a recuperação talvez seja possível.
  - Exemplo: gravação de arquivo em disco cheio
- Exceções não-verificadas:
  - utilizadas para falhas não-antecipadas; e
  - onde a recuperação não é possível.
  - Normalmente, erro de programa

Exceções não-verificadas

- A utilização dessas exceções ocorre de forma ‘não-verificada’ (poucas regras) pelo compilador.
- Causam o término do programa se não capturadas.
  - Essa é a prática normal.
- `RuntimeError` é o exemplo padrão.

O efeito de uma exceção

- O que acontece quando uma exceção é lançada?
- Há dois efeitos a considerar:
  - Efeito no método em que a exceção é lançada e o
  - efeito no chamador

Efeito no lançador

- O método de lançamento termina prematuramente.
- Nenhum valor de retorno é retornado.
- Controle não retorna ao ponto da chamada do cliente.
  - Portanto, o cliente não pode prosseguir de qualquer maneira.

Verificação de argumento

```
def getDetails(key):

    if(key == None):
        raise KeyError("null key in getDetails")

    if(len(key)== 0):
        raise ValueError("Empty key passed to getDetails")

    return book.get(key)
```

Efeito no chamador

- Considere a seguinte chamada improvisada a getDetails:

```
details = addresses.getDetails(None)
// a seguinte instrução não será alcançada
phone = details.getPhone()
```

- Um cliente deve ‘capturar’ a exceção, se não o programa terminará bruscamente.

Tratamento de exceções

- Exceções verificadas devem ser capturadas.
- O compilador assegura que a utilização dessas exceções seja fortemente controlada.
- Se utilizadas apropriadamente, é possível recuperar-se das falhas.

O bloco try

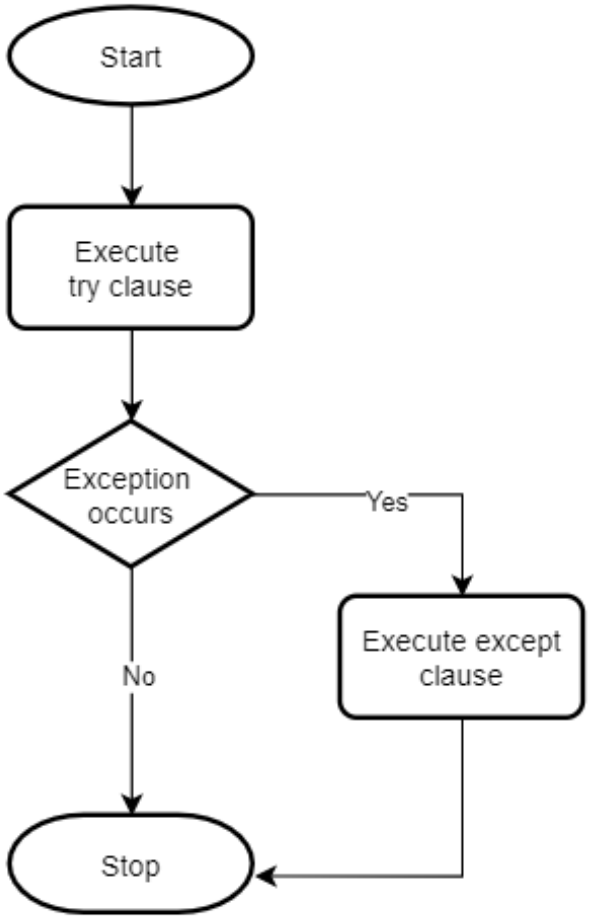
- Clientes que capturam uma exceção devem proteger a chamada com um bloco try:

```
try:
    Proteja uma ou mais instruções aqui.
except:
    Informe da exceção e recuperação aqui.
else:
    se não ocorrer exceção então execute este bloco
```

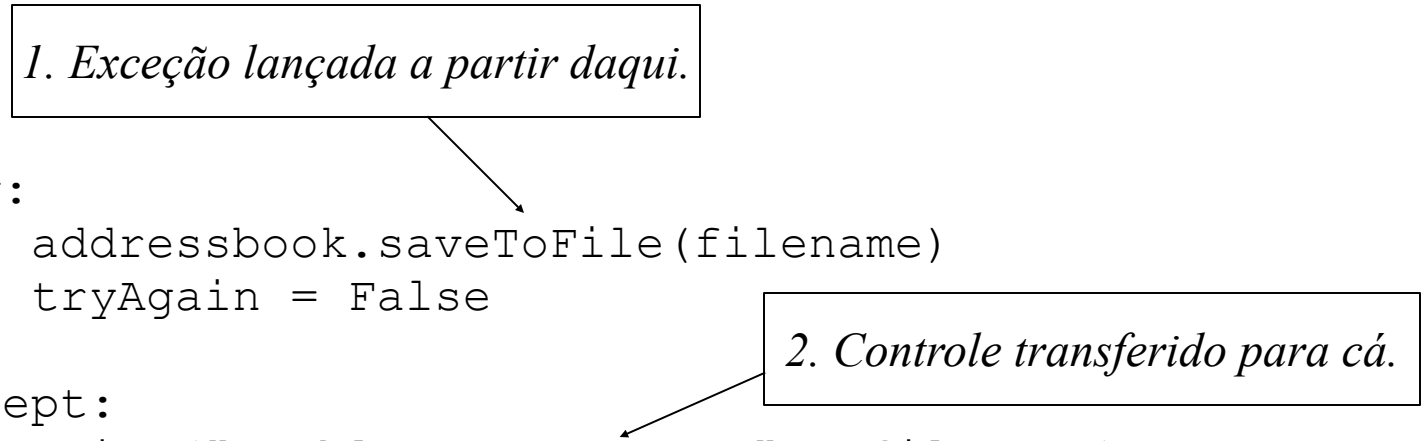
O bloco try

```
try:
    # code that may cause error
except:
    # handle errors
```

O bloco try


Start
Execute
try clause
Exception
occurs
Yes
Execute except
clause
No
Stop

O bloco try




```
try:
    addressbook.saveToFile(filename)
    tryAgain = False

except:
    print("Unable to save to " + filename)
    tryAgain = True

else:
    pass
```

Capturando múltiplas exceções

```
try:
    ...
    ref.process();
    ...

except EOFError:
    // Toma a ação apropriada para uma exceção
    // de final de arquivo alcançado.
    ...

except IOError:
    // Toma a ação apropriada para uma exceção
    // de final de arquivo alcançado.
    ...
else:
```

A cláusula finally

```
try:
    Proteja uma ou mais instruções aqui.


finally:
    Realize quaisquer ações aqui comuns quer ou não
    uma exceção seja lançada.
```

A cláusula finally

- Uma cláusula finally é executada mesmo se uma instrução de retorno for executada na cláusula try.
- Ainda há uma exceção não-capturada ou propagada via a cláusula finally.

Definindo novas classes de exceção

- Estenda `Exception` ou `RuntimeError`.
- Defina novos tipos para fornecer melhores informações diagnósticas.
  - Inclua informações sobre a notificação e/ou recuperação.

```
class NoMatchingDetailsException(Exception):

    def __init__(self, key):

        self.__key = key


    def getKey(self):

        return self.__key


    def toString(self):

        return "No details matching '" + self.__key +
               "' were found.";
```

Recuperação após erro

- Clientes devem tomar nota dos informes de erros.
  - Verifique o valor de retorno.
  - Não ‘ignore’ exceções.
- Inclua o código para a tentativa de recuperação.
  - Freqüentemente isso exigirá um loop.

Tentativa de recuperação

```
// Tenta salvar o catálogo de endereços.
successful = False
attempts = 0;
while !successful and attempts < MAX_ATTEMPTS:
    try:
        addressbook.saveToFile(filename)
        successful = True

    except IOError:
        print("Unable to save to " + filename)
        attempts++
        if(attempts < MAX_ATTEMPTS):
            filename = um nome de arquivo alternativo;



if(!successful):
    Informa o problema e desiste;
```

Prevenção de erro

- Clientes podem freqüentemente utilizar os métodos de pesquisa do servidor para evitar erros.
  - Ter clientes mais robustos significa que os servidores podem ser mais confiáveis.
  - Exceções não-verificadas podem ser utilizadas.
  - Simplifica a lógica do cliente.
- Pode aumentar o acoplamento cliente/servidor.

Revisão (1)

- Erros em tempo de execução surgem por várias razões.
  - Uma chamada cliente inadequada a um objeto servidor.
  - Um servidor incapaz de atender a uma solicitação.
  - Erro de programação no cliente e/ou servidor.

Revisão (2)

- Erros em tempo de execução freqüentemente levam a falhas de programa.
- Programação defensiva antecipa erros — tanto no cliente como no servidor.
- Exceções fornecem um mecanismo de informe e recuperação.

Revisão (3)

- Exceções comuns em Python:
  - Exception
  - RuntimeError
  - OSError
  - ValueError
  - TypeError
  - NameError
  - ZeroDivisionError