

Tratamento de Erros, Assertions e Programação Defensiva em Python

Sumário

1. Exceptions (try / except / else / finally)
 2. Custom Exceptions
 3. Assertions
 4. Defensive Programming
 5. Resumo e Checklist de Boas Práticas
-

1. Exceptions (try / except / else / finally)

Definição

Exceções são eventos que ocorrem durante a execução de um programa e interrompem o fluxo normal de instruções. O tratamento de exceções permite que o programa capture e responda a esses eventos de forma controlada, evitando falhas inesperadas.

A estrutura básica de tratamento de exceções em Python consiste em:

- **try**: bloco onde o código potencialmente problemático é executado
- **except**: bloco que captura e trata exceções específicas
- **else**: bloco executado se nenhuma exceção ocorrer
- **finally**: bloco sempre executado, independentemente de exceções

Motivação

Sem tratamento de exceções, um erro interrompe o programa imediatamente, o que pode levar à perda de dados, dificultar o diagnóstico e impedir qualquer tentativa de recuperação.

Um bom tratamento de exceções permite que o software continue operando ou falhe de maneira previsível quando algo dá errado.

Exemplos

Exemplo 1: Tratamento básico

```
def dividir(a, b):
    try:
        resultado = a / b
    except ZeroDivisionError as e:
        raise ValueError("Divisão por zero não é permitida") from e
    else:
        print(f"Divisão realizada com sucesso: {resultado}")
        return resultado
    finally:
        print("Operação de divisão finalizada")

# Testando
dividir(10, 2)    # Sucesso
dividir(10, 0)    # Erro tratado
```

Exemplo 2: Múltiplas exceções

```

def processar_arquivo(nome_arquivo):
    try:
        with open(nome_arquivo, 'r') as arquivo:
            conteudo = arquivo.read()
            numero = int(conteudo)
            return 100 / numero
    except FileNotFoundError:
        print(f"Arquivo '{nome_arquivo}' não encontrado")
    except ValueError:
        print("Conteúdo do arquivo não é um número válido")
    except ZeroDivisionError:
        print("O número no arquivo não pode ser zero")
    except Exception as e:
        raise RuntimeError("Erro inesperado durante o processamento") from e
    finally:
        print("Tentativa de processamento concluída")

```

Exemplo 3: Uso do else

```

def obter_idade():
    try:
        idade = int(input("Digite sua idade: "))
    except ValueError:
        print("Entrada inválida! Por favor, digite um número.")
        return None
    else:
        # Este bloco só executa se não houver exceção
        if idade < 0:
            print("Idade não pode ser negativa")
            return None
        print(f"Idade registrada: {idade}")
        return idade

```

Exemplo 4: Capturando informações da exceção

```

import traceback

def operacao_complexa(dados):
    try:
        resultado = dados['valor'] * 2
        return resultado
    except KeyError as e:
        print(f"Chave ausente: {e}")
        print("Traceback completo:")
        traceback.print_exc()
    except TypeError as e:
        print(f"Erro de tipo: {e}")

```

Exercícios

Exercício 1: Implemente uma função `converter_temperatura(valor, escala)` que converte temperaturas entre Celsius e Fahrenheit. A função deve tratar exceções para valores não numéricos e escalas inválidas.

Exercício 2: Crie uma função `ler_arquivo_json(caminho)` que leia um arquivo JSON e retorne seus dados. Trate as exceções apropriadas (arquivo não encontrado, JSON inválido, permissões).

Exercício 3: Desenvolva uma calculadora que solicite dois números e uma operação ao usuário. Trate todas as exceções possíveis (entradas inválidas, divisão por zero, operações desconhecidas).

Desafios

Desafio 1: Implemente um sistema de retry que tenta executar uma função até 3 vezes antes de desistir, com intervalo crescente entre tentativas. Use tratamento de exceções para controlar o fluxo.

Desafio 2: Crie um contexto manager personalizado usando `__enter__` e `__exit__` que garanta que um recurso seja liberado mesmo em caso de exceção.

Desafio 3: Desenvolva um decorador que capture exceções de funções, registre-as em um arquivo de log com timestamp e re-lance a exceção original.

Aplicações

1. **APIs Web:** Tratamento de falhas de rede, timeout, respostas inválidas
2. **Bancos de Dados:** Conexões perdidas, queries inválidas, transações
3. **Processamento de Arquivos:** Arquivos corrompidos, permissões, espaço em disco
4. **Integração de Sistemas:** Serviços indisponíveis, formatos incompatíveis
5. **Interface com Usuário:** Validação de entrada, feedback de erros

Boas Práticas

1. **Seja específico:** capture exceções específicas, não use `except Exception` indiscriminadamente
2. **Não silencie/ignore erros:** o bloco `except` deve registrar, propagar ou tratar o problema de forma explícita
3. **Use finally para limpeza:** libere recursos (arquivos, conexões) no bloco `finally`
4. **Evite except vazio:** nunca use `except: pass` sem justificativa clara
5. **Documente exceções:** indique em docstrings quais exceções uma função pode lançar
6. **Falhe rápido:** lance exceções cedo quando detectar problemas
7. **Mensagens com contexto:** erros devem trazer informações suficientes para identificar a causa

```
# Ruim
try:
    processar_dados(dados)
except:
    pass

# Bom
try:
    processar_dados(dados)
except ValueError as e:
    logger.error(f"Dados inválidos ao processar: {e}")
    raise
except Exception as e:
    logger.critical(f"Erro inesperado: {e}")
    notificar_equipe(e)
    raise
```

2. Custom Exceptions

Definição

Exceções personalizadas são classes de exceção definidas pelo programador que herdam de `Exception` ou de suas subclasses. Elas permitem criar tipos de erro específicos para o domínio da aplicação, tornando o código mais expressivo e facilitando o tratamento diferenciado de erros.

Motivação

Exceções personalizadas ajudam a deixar claro o tipo de erro ocorrido, permitem tratamento específico e evitam o uso excessivo de exceções genéricas.

Exemplos

Exemplo 1: Exceção simples

```
class SaldoInsuficienteError(Exception):
    """Exceção lançada quando uma conta não tem saldo suficiente"""
    pass

class ContaBancaria:
    def __init__(self, saldo_inicial=0):
        self.saldo = saldo_inicial

    def sacar(self, valor):
        if valor > self.saldo:
            raise SaldoInsuficienteError(
                f"Tentativa de sacar R$ {valor:.2f}, "
                f"mas o saldo é apenas R$ {self.saldo:.2f}"
            )
        self.saldo -= valor
        return self.saldo

# Uso
conta = ContaBancaria(100)
try:
    conta.sacar(150)
except SaldoInsuficienteError as e:
    print(f"Operação negada: {e}")
```

Exemplo 2: Exceção com atributos adicionais

```
class ValidationError(Exception):
    """Erro de validação com detalhes sobre o campo e valor inválido"""

    def __init__(self, campo, valor, mensagem):
        self.campo = campo
        self.valor = valor
        self.mensagem = mensagem
        super().__init__(f"{campo}: {mensagem} (valor fornecido: {valor})")

def validar_usuario(dados):
    if 'email' not in dados:
        raise ValidationError('email', None, 'Campo obrigatório')

    email = dados['email']
    if '@' not in email:
        raise ValidationError('email', email, 'Formato inválido')

    if 'idade' in dados and dados['idade'] < 0:
        raise ValidationError('idade', dados['idade'], 'Não pode ser negativa')

# Uso
try:
    validar_usuario({'email': 'usuario.com', 'idade': -5})
except ValidationError as e:
    print(f"Erro no campo '{e.campo}': {e.mensagem}")
```

Exemplo 3: Hierarquia de exceções

```
class PagamentoError(Exception):
    """Classe base para erros de pagamento"""
    pass

class CartaoRecusadoError(PagamentoError):
    """Cartão de crédito recusado"""
    def __init__(self, motivo):
        self.motivo = motivo
        super().__init__(f"Cartão recusado: {motivo}")

class LimiteExcedidoError(PagamentoError):
    """Limite de crédito excedido"""
    def __init__(self, valor_tentado, limite):
        self.valor_tentado = valor_tentado
        self.limite = limite
        super().__init__(
            f"Limite excedido. Tentativa: R$ {valor_tentado:.2f}, "
            f"Limite: R$ {limite:.2f}"
        )

class GatewayIndisponivelError(PagamentoError):
    """Gateway de pagamento indisponível"""
    pass

def processar_pagamento(valor, cartao):
    if not verificar_gateway():
        raise GatewayIndisponivelError("Serviço temporariamente indisponível")

    if valor > cartao.limite:
        raise LimiteExcedidoError(valor, cartao.limite)

    if cartao.bloqueado:
        raise CartaoRecusadoError("Cartão bloqueado")

    # Processar pagamento...

# Uso
try:
    processar_pagamento(1000, meu_cartao)
```

```
except LimiteExcedidoError as e:  
    print("Tente um valor menor ou outro cartão")  
except CartaoRecusadoError as e:  
    print(f"Problema com o cartão: {e.motivo}")  
except PagamentoError as e:  
    print(f"Erro no pagamento: {e}")
```

Exemplo 4: Exceções com contexto rico

```

from datetime import datetime

class RequisicaoAPIError(Exception):
    """Erro ao fazer requisição para API externa"""

    def __init__(self, endpoint, status_code, mensagem, timestamp=None):
        self.endpoint = endpoint
        self.status_code = status_code
        self.timestamp = timestamp or datetime.now()
        super().__init__(
            f"[{self.timestamp.isoformat()}] "
            f"Erro {status_code} ao acessar {endpoint}: {mensagem}"
        )

    def to_dict(self):
        """Converte exceção para dicionário para logging/serialização"""
        return {
            'endpoint': self.endpoint,
            'status_code': self.status_code,
            'mensagem': str(self),
            'timestamp': self.timestamp.isoformat()
        }

def chamar_api(endpoint):
    resposta = fazer_requisicao(endpoint)
    if resposta.status_code != 200:
        raise RequisicaoAPIError(
            endpoint=endpoint,
            status_code=resposta.status_code,
            mensagem=resposta.texto
        )
    return resposta.json()

```

Exercícios

Exercício 1: Crie uma hierarquia de exceções para um sistema de biblioteca:

`BibliotecaError` (base), `LivroNaoEncontradoError`, `LivroIndisponivelError`, `UsuarioSuspensoError`.

Exercício 2: Implemente exceções personalizadas para validação de CPF: `CPFInvalidoError` com atributos para indicar o tipo de erro (formato, dígito verificador, etc.).

Exercício 3: Desenvolva exceções para um sistema de reservas de hotel que carreguem informações sobre disponibilidade, datas conflitantes e quartos.

Desafios

Desafio 1: Crie um sistema de exceções que suporte internacionalização (i18n), permitindo que a mensagem seja exibida em diferentes idiomas baseado em um parâmetro de localização.

Desafio 2: Implemente um decorador que converta exceções built-in em exceções personalizadas do domínio, mantendo o traceback original.

Desafio 3: Desenvolva um sistema de exceções que automaticamente registre cada ocorrência em um sistema de monitoramento (como Sentry), incluindo contexto da aplicação.

Aplicações

1. **Frameworks Web:** Exceções para erros HTTP (404, 401, 500, etc.)
2. **ORMs:** Exceções para operações de banco de dados
3. **APIs:** Exceções para diferentes tipos de falha na comunicação
4. **Validação:** Exceções para regras de negócio e validação de dados
5. **Domínio específico:** Exceções que refletem conceitos do negócio

Boas Práticas

1. **Herde de Exception:** Sempre herde de `Exception` ou suas subclasses, nunca de `BaseException`
2. **Nome descritivo:** Use sufixo `Error` ou `Exception` no nome
3. **Docstrings:** Documente quando e por que a exceção é lançada
4. **Hierarquia lógica:** Organize exceções relacionadas em hierarquia
5. **Informação suficiente:** Inclua dados necessários para diagnóstico
6. **Construtor claro:** Aceite parâmetros que facilitem o uso
7. **Serialização:** Para sistemas distribuídos, torne exceções serializáveis

8. Compatibilidade: Mantenha compatibilidade ao evoluir exceções

Convenção recomendada

Use o sufixo `Error` para exceções de domínio (`PagamentoError`) e evite misturar com `Exception` no nome para manter consistência.

```
# Boa estrutura de exceção customizada
class ProcessamentoError(Exception):
    """
    Exceção base para erros de processamento de dados.

    Attributes:
        fonte: Origem dos dados que causou o erro
        detalhes: Informações adicionais sobre o erro
    """

    def __init__(self, mensagem, fonte=None, detalhes=None):
        self.fonte = fonte
        self.detalhes = detalhes or {}
        super().__init__(mensagem)

    def __str__(self):
        msg = super().__str__()
        if self.fonte:
            msg = f"[{self.fonte}] {msg}"
        return msg
```

3. Assertions

Definição

Assertions (afirmações) são verificações que avaliam se uma condição é verdadeira em um determinado ponto do programa. Em Python, usamos a declaração `assert` que lança `AssertionError` se a condição for falsa. Assertions são tipicamente usadas para verificar invariantes do programa e condições que nunca deveriam ser falsas.

Sintaxe: `assert condição, mensagem_opcional`

Atenção

Assertions **NÃO** substituem validação de entrada nem lógica de negócio.

Como podem ser desativadas com `python -O`, nunca dependa de `assert` para garantir comportamento correto em produção.

Motivação

Assertions explicitam hipóteses e invariantes do código e interrompem a execução imediatamente quando essas condições são violadas, o que facilita a detecção e a localização de erros durante a fase de desenvolvimento. Seu uso contribui para a identificação precoce de bugs, simplifica o processo de depuração e torna os invariantes mais claros e documentados no próprio código.

Exemplos

Exemplo 1: Verificação de pré-condições

```
def calcular_raiz_quadrada(numero):
    """Calcula a raiz quadrada de um número não-negativo"""
    assert numero >= 0, f"Número deve ser não-negativo, recebido: {numero}"
    return numero ** 0.5

def dividir_lista(lista, n):
    """Divide uma lista em n partes iguais"""
    assert len(lista) % n == 0, \
        f"Lista de tamanho {len(lista)} não é divisível por {n}"

    tamanho_parte = len(lista) // n
    return [lista[i:i+tamanho_parte] for i in range(0, len(lista), tamanho_parte)]
```

Exemplo 2: Verificação de pós-condições

```
def ordenar_e_remover_duplicatas(lista):
    """Ordena lista e remove duplicatas"""
    resultado = sorted(set(lista))

    # Pós-condições
    assert len(resultado) <= len(lista), "Resultado não pode ser maior que entrada"
    assert resultado == sorted(resultado), "Resultado deve estar ordenado"
    assert len(resultado) == len(set(resultado)), "Não deve haver duplicatas"

    return resultado

def criar_matriz_identidade(n):
    """Cria matriz identidade n x n"""
    matriz = [[1 if i == j else 0 for j in range(n)] for i in range(n)]

    # Verificar propriedades da matriz identidade
    assert len(matriz) == n, "Número de linhas incorreto"
    assert all(len(linha) == n for linha in matriz), "Número de colunas incorreto"
    assert sum(matriz[i][i] for i in range(n)) == n, "Diagonal deve somar n"

    return matriz
```

Exemplo 3: Verificação de invariantes

```

class Fila:
    """Implementação de fila com verificação de invariantes"""

    def __init__(self):
        self._items = []
        self._tamanho = 0
        self._verificar_invariantes()

    def _verificar_invariantes(self):
        """Verifica que os invariantes da classe são mantidos"""
        assert self._tamanho == len(self._items), \
            "Tamanho deve corresponder ao número de itens"
        assert self._tamanho >= 0, "Tamanho não pode ser negativo"

    def enfileirar(self, item):
        self._items.append(item)
        self._tamanho += 1
        self._verificar_invariantes()

    def desenfileirar(self):
        assert not self.vazia(), "Não é possível desenfileirar de fila vazia"
        item = self._items.pop(0)
        self._tamanho -= 1
        self._verificar_invariantes()
        return item

    def vazia(self):
        return self._tamanho == 0

class ContadorRestrito:
    """Contador que deve permanecer dentro de limites"""

    def __init__(self, valor_inicial=0, minimo=0, maximo=100):
        assert minimo <= valor_inicial <= maximo, \
            "Valor inicial fora dos limites"
        self._valor = valor_inicial
        self._min = minimo
        self._max = maximo

```

```
def _verificar_invariante(self):
    assert self._min <= self._valor <= self._max, \
        f"Valor {self._valor} fora dos limites [{self._min}, {self._max}]"

def incrementar(self, delta=1):
    self._valor += delta
    self._verificar_invariante()

def decrementar(self, delta=1):
    self._valor -= delta
    self._verificar_invariante()
```

Exemplo 4: Assertions em algoritmos

```
def busca_binaria(lista, alvo):
    """Busca binária com verificação de pré-condição"""
    assert lista == sorted(lista), "Lista deve estar ordenada para busca binária"

    esquerda, direita = 0, len(lista) - 1

    while esquerda <= direita:
        meio = (esquerda + direita) // 2

        # Invariante: se alvo existe, está entre esquerda e direita
        assert esquerda <= meio <= direita

        if lista[meio] == alvo:
            return meio
        elif lista[meio] < alvo:
            esquerda = meio + 1
        else:
            direita = meio - 1

    return -1


def merge(esquerda, direita):
    resultado = []
    i = j = 0

    while i < len(esquerda) and j < len(direita):
        if esquerda[i] <= direita[j]:
            resultado.append(esquerda[i])
            i += 1
        else:
            resultado.append(direita[j])
            j += 1

    resultado.extend(esquerda[i:])
    resultado.extend(direita[j:])
    return resultado
```

```

def merge_sort(lista):
    """Merge sort com verificação de propriedades"""
    if len(lista) <= 1:
        return lista

    meio = len(lista) // 2
    esquerda = merge_sort(lista[:meio])
    direita = merge_sort(lista[meio:])

    # Verificar que sublistas estão ordenadas
    assert esquerda == sorted(esquerda), "Sublista esquerda deve estar ordenada"
    assert direita == sorted(direita), "Sublista direita deve estar ordenada"

    resultado = merge(esquerda, direita)

    # Pós-condição: resultado ordenado e mesmo tamanho
    assert resultado == sorted(resultado), "Resultado deve estar ordenado"
    assert len(resultado) == len(lista), "Tamanho deve ser preservado"

    return resultado

```

Exercícios

Exercício 1: Implemente uma função `calcular_media(numeros)` que calcule a média de uma lista. Use assertions para verificar que a lista não está vazia e que todos os elementos são numéricos.

Exercício 2: Crie uma classe `Retangulo` com atributos `largura` e `altura`. Use assertions no construtor e nos setters para garantir que dimensões são sempre positivas.

Exercício 3: Implemente uma função `encontrar_mediana(lista)` que encontra a mediana de uma lista. Use assertions para verificar pré e pós-condições.

Desafios

Desafio 1: Crie um decorador `@assert_types` que verifica os tipos dos argumentos e do retorno de uma função baseado em type hints.

Desafio 2: Implemente uma classe `MatrizQuadrada` que mantém invariantes matemáticos (determinante, traço, etc.) e os verifica após cada operação.

Desafio 3: Desenvolva um sistema de "design by contract" onde classes podem declarar invariantes, pré-condições e pós-condições que são verificadas automaticamente.

Aplicações

1. **Desenvolvimento:** Verificação de lógica durante implementação
2. **Testes:** Verificação de estados intermediários em testes
3. **Algoritmos:** Garantia de invariantes em estruturas de dados
4. **Prototipagem:** Validação rápida de suposições
5. **Documentação:** Explicitar contratos e suposições

Boas Práticas

1. **Use para bugs, não para validação:** Assertions são para detectar bugs do programador, não para validar entrada de usuário
2. **Nunca capture AssertionError:** Deixe assertions falharem e corrigir o bug
3. **Assertions não substituem testes:** São complementares aos testes automatizados
4. **Mensagens claras:** Sempre inclua mensagem explicativa
5. **Não tenha efeitos colaterais:** Assertions não devem modificar estado
6. **Verifique invariantes:** Use assertions para documentar e verificar invariantes de classes
7. **Podem ser desabilitadas:** Não use para lógica essencial do programa

```

# Ruim - assertion com efeito colateral
assert (x := processar_dados()) > 0

# Bom - sem efeito colateral
x = processar_dados()
assert x > 0, f"Resultado do processamento deve ser positivo, obtido: {x}"

# Ruim - validação de entrada com assertion
def sacar(valor):
    assert valor > 0, "Valor deve ser positivo" # NUNCA FAÇA ISSO
    self.saldo -= valor

# Bom - validação explícita
def sacar(self, valor):
    if valor <= 0:
        raise ValueError("Valor deve ser positivo")
    assert self.saldo >= valor, "Invariante: saldo sempre não-negativo"
    self.saldo -= valor

```

4. Defensive Programming

Definição

Programação defensiva é uma abordagem de desenvolvimento que antecipa possíveis problemas e implementa mecanismos para preveni-los ou detectá-los precocemente. O objetivo é criar código que funcione corretamente mesmo em circunstâncias inesperadas ou com entradas inválidas.

Na prática, isso significa validar entradas, considerar estados inválidos e garantir que falhas ocorram de forma previsível.

Motivação

Em ambientes de produção, o código enfrenta situações fora do planejado:

- **Entradas inválidas:** Usuários fornecem dados inesperados
- **Estado inconsistente:** Sistemas externos podem falhar

- **Concorrência:** Múltiplas threads podem causar condições de corrida
- **Recursos limitados:** Memória, disco e rede podem se esgotar
- **Bugs futuros:** Mudanças no código podem introduzir problemas

Esse tipo de abordagem reduz falhas causadas por entradas inesperadas e estados inconsistentes.

Exemplos

Exemplo 1: Validação de entrada

```
def calcular_idade(ano_nascimento):
    """
    Calcula idade a partir do ano de nascimento.
    Implementação defensiva com múltiplas validações.
    """
    from datetime import datetime

    # Validação de tipo
    if not isinstance(ano_nascimento, int):
        raise TypeError(f"Ano deve ser inteiro, recebido: {type(ano_nascimento)}")

    ano_atual = datetime.now().year

    # Validação de range
    if ano_nascimento < 1900:
        raise ValueError(f"Ano muito antigo: {ano_nascimento}")

    if ano_nascimento > ano_atual:
        raise ValueError(f"Ano no futuro: {ano_nascimento}")

    idade = ano_atual - ano_nascimento

    # Pós-condição
    assert 0 <= idade <= 150, f"Idade calculada fora do esperado: {idade}"

    return idade

def processar_email(email):
    """Processa email com validações defensivas"""
    # Validar tipo e não-nulo
    if not isinstance(email, str):
        raise TypeError("Email deve ser string")

    # Normalizar entrada
    email = email.strip().lower()

    # Validar não-vazio
    if not email:
        raise ValueError("Email não pode ser vazio")
```

```
# Validação básica de formato
if '@' not in email or '.' not in email.split('@')[-1]:
    raise ValueError(f"Formato de email inválido: {email}")

# Validar comprimento razoável
if len(email) > 254: # RFC 5321
    raise ValueError("Email muito longo")

return email
```

Exemplo 2: Valores padrão seguros

```
def criar_relatorio(dados, formato='pdf', incluir_graficos=True,
                    destino=None, timeout=30):
    """
    Cria relatório com valores padrão seguros e validações.
    """

    # Validar formato
    formatos_validos = {'pdf', 'html', 'csv'}
    if formato not in formatos_validos:
        raise ValueError(
            f"Formato '{formato}' inválido."
            f"Opcões: {', '.join(formatos_validos)}"
        )

    # Garantir que dados não é None
    if dados is None:
        raise ValueError("Dados não podem ser None")

    # Copiar dados para evitar modificação do original
    dados = dados.copy() if hasattr(dados, 'copy') else list(dados)

    # Validar timeout
    if not isinstance(timeout, (int, float)) or timeout <= 0:
        raise ValueError(f"Timeout inválido: {timeout}")

    # Usar destino padrão seguro se não fornecido
    if destino is None:
        from pathlib import Path
        destino = Path.home() / 'relatorios' / f'relatorio.{formato}'
        destino.parent.mkdir(parents=True, exist_ok=True)

    # Processar relatório...
    return destino

def conectar_banco(host='localhost', porta=5432, usuario=None,
                  senha=None, timeout=10, retry=3):
    """
    Conecta a banco de dados com configurações defensivas.
    """

    # Nunca use valores mutáveis como padrão
```

```
# Validações antes de tentar conexão

if usuario is None:
    raise ValueError("Usuário é obrigatório")

if senha is None:
    raise ValueError("Senha é obrigatória")

if not (1 <= porta <= 65535):
    raise ValueError(f"Porta inválida: {porta}")

if retry < 0:
    raise ValueError("Número de tentativas não pode ser negativo")

# Implementar lógica de conexão com retry...
pass
```

Exemplo 3: Cópia defensiva e imutabilidade

```
class ConfiguracaoSistema:  
    """  
        Gerencia configurações do sistema com proteção defensiva.  
    """  
  
    def __init__(self, config_inicial=None):  
        # Nunca armazene referência direta a objetos mutáveis externos  
        if config_inicial is None:  
            self._config = {}  
        else:  
            # Cópia profunda para prevenir modificações externas  
            import copy  
            self._config = copy.deepcopy(config_inicial)  
  
        # Validar configurações  
        self._validar_config()  
  
    def _validar_config(self):  
        """Valida estrutura e valores da configuração"""  
        chaves_obrigatorias = {'host', 'porta', 'modo'}  
        if not all(chave in self._config for chave in chaves_obrigatorias):  
            faltando = chaves_obrigatorias - self._config.keys()  
            raise ValueError(f"Configurações obrigatórias faltando: {faltando}")  
  
    def obter(self, chave, padrao=None):  
        """  
            Obtém valor de configuração com retorno defensivo.  
            Nunca retorna referência direta a objetos mutáveis.  
        """  
        valor = self._config.get(chave, padrao)  
  
        # Retornar cópia se for mutável  
        if isinstance(valor, (list, dict, set)):  
            import copy  
            return copy.deepcopy(valor)  
  
        return valor  
  
    def definir(self, chave, valor):
```

```

"""Define valor com validação"""
if not isinstance(chave, str) or not chave:
    raise ValueError("Chave deve ser string não-vazia")

# Armazenar cópia para prevenir modificações externas
if isinstance(valor, (list, dict, set)):
    import copy
    valor = copy.deepcopy(valor)

self._config[chave] = valor

def obter_config_completa(self):
    """Retorna cópia completa da configuração"""
    import copy
    return copy.deepcopy(self._config)

def processar_lista_usuarios(usuarios):
    """
    Processa lista de usuários sem modificar original.
    """

    # NUNCA modifique parâmetros mutáveis diretamente
    # Crie uma cópia primeiro
    usuarios_processados = []

    for usuario in usuarios:
        # Criar novo dicionário ao invés de modificar
        usuario_novo = {
            'nome': usuario.get('nome', '').strip().title(),
            'email': usuario.get('email', '').strip().lower(),
            'ativo': usuario.get('ativo', True)
        }
        usuarios_processados.append(usuario_novo)

    return usuarios_processados

```

Exemplo 4: Verificações de tipo e estado

```
class ContaBancaria:  
    """Implementação defensiva de conta bancária"""  
  
    def __init__(self, titular, saldo_inicial=0):  
        # Validação no construtor  
        if not isinstance(titular, str) or not titular.strip():  
            raise ValueError("Titular deve ser string não-vazia")  
  
        if not isinstance(saldo_inicial, (int, float)):  
            raise TypeError("Saldo inicial deve ser numérico")  
  
        if saldo_inicial < 0:  
            raise ValueError("Saldo inicial não pode ser negativo")  
  
        self._titular = titular.strip()  
        self._saldo = float(saldo_inicial)  
        self._ativa = True  
        self._historico = []  
  
    def _verificar_conta_ativa(self):  
        """Verifica se conta está ativa antes de operações"""  
        if not self._ativa:  
            raise RuntimeError("Operação não permitida: conta inativa")  
  
    def depositar(self, valor):  
        """Deposita valor com validações defensivas"""  
        self._verificar_conta_ativa()  
  
        # Validar tipo  
        if not isinstance(valor, (int, float)):  
            raise TypeError(f"Valor deve ser numérico, recebido: {type(valor)}")  
  
        # Validar range  
        if valor <= 0:  
            raise ValueError(f"Valor deve ser positivo: {valor}")  
  
        # Validar limite razoável (proteção contra overflow)  
        if valor > 1_000_000_000:  
            raise ValueError("Valor excede limite permitido por transação")
```

```
# Verificar se não causará overflow
novo_saldo = self._saldo + valor
import math
if math.isinf(novo_saldo): # Detecção de overflow
    raise OverflowError("Operação resultou em infinito")

self._saldo = novo_saldo
self._historico.append(f"Depósito: +{valor:.2f}")

return self._saldo

def sacar(self, valor):
    """Saca valor com validações defensivas"""
    self._verificar_conta_ativa()

    if not isinstance(valor, (int, float)):
        raise TypeError("Valor deve ser numérico")

    if valor <= 0:
        raise ValueError("Valor deve ser positivo")

    if valor > self._saldo:
        raise ValueError(
            f"Saldo insuficiente. "
            f"Disponível: {self._saldo:.2f}, Solicitado: {valor:.2f}"
        )

    self._saldo -= valor
    self._historico.append(f"Saque: -{valor:.2f}")

return self._saldo

def obter_historico(self):
    """Retorna cópia do histórico para prevenir modificação"""
    return self._historico.copy()

def dividir_com_segurança(a, b):
    """
```

```
Divisão com múltiplas camadas de proteção.

"""

# Validação de tipos
if not isinstance(a, (int, float)):
    raise TypeError(f"Dividendo deve ser numérico: {type(a)}")

if not isinstance(b, (int, float)):
    raise TypeError(f"Divisor deve ser numérico: {type(b)}")

# Verificar divisão por zero
if b == 0:
    raise ZeroDivisionError("Não é possível dividir por zero")

# Verificar valores muito pequenos que podem causar instabilidade
if abs(b) < 1e-10:
    raise ValueError(f"Divisor muito próximo de zero: {b}")

resultado = a / b

# Verificar resultado válido
import math
if math.isnan(resultado):
    raise ValueError("Operação resultou em NaN")

if math.isinf(resultado):
    raise OverflowError("Operação resultou em infinito")

return resultado
```

Exemplo 5: Tratamento de recursos e contextos

```
import logging
from contextlib import contextmanager

class GerenciadorArquivo:
    """Gerenciador defensivo de arquivos"""

    def __init__(self, caminho, modo='r'):
        # Validações
        if not isinstance(caminho, str) or not caminho.strip():
            raise ValueError("Caminho deve ser string não-vazia")

        modos_validos = {'r', 'w', 'a', 'rb', 'wb', 'ab'}
        if modo not in modos_validos:
            raise ValueError(f"Modo inválido: {modo}")

        self.caminho = caminho
        self.modo = modo
        self._arquivo = None
        self._aberto = False

    def abrir(self):
        """Abre arquivo com tratamento de erros"""
        if self._aberto:
            raise RuntimeError("Arquivo já está aberto")

        try:
            self._arquivo = open(self.caminho, self.modo)
            self._aberto = True
        except FileNotFoundError:
            logging.error(f"Arquivo não encontrado: {self.caminho}")
            raise
        except PermissionError:
            logging.error(f"Sem permissão para acessar: {self.caminho}")
            raise
        except Exception as e:
            logging.error(f"Erro ao abrir arquivo: {e}")
            raise

    return self._arquivo
```

```
def fechar(self):
    """Fecha arquivo com segurança"""
    if self._arquivo and self._aberto:
        try:
            self._arquivo.close()
        except Exception as e:
            logging.warning(f"Erro ao fechar arquivo: {e}")
    finally:
        self._aberto = False
        self._arquivo = None

def __enter__(self):
    return self.abrir()

def __exit__(self, exc_type, exc_val, exc_tb):
    self.fechar()
    return False # Não suprimir exceções

@contextmanager
def conexao_segura(config):
    """Context manager para conexões com limpeza garantida"""
    conexao = None
    try:
        # Validar configuração antes de conectar
        if not isinstance(config, dict):
            raise TypeError("Configuração deve ser dicionário")

        if 'host' not in config:
            raise ValueError("Host é obrigatório na configuração")

        # Estabelecer conexão
        conexao = estabelecer_conexao(config)
        yield conexao

    except Exception as e:
        logging.error(f"Erro na conexão: {e}")
        raise
```

```
finally:
    # Garantir limpeza mesmo em caso de erro
    if conexao:
        try:
            conexao.fechar()
        except Exception as e:
            logging.warning(f"Erro ao fechar conexão: {e}")

def processar_com_timeout(funcao, timeout=5):
    """
    Executa função com timeout para prevenir travamentos.

    AVISO: Disponível apenas em sistemas Unix (Linux/macOS).
    Não funciona no Windows.
    """
    import signal

    def handler(signum, frame):
        raise TimeoutError(f"Função excedeu timeout de {timeout}s")

    # Validações
    if not callable(funcao):
        raise TypeError("Primeiro argumento deve ser função")

    if not isinstance(timeout, (int, float)) or timeout <= 0:
        raise ValueError("Timeout deve ser positivo")

    # Configurar alarme
    signal.signal(signal.SIGALRM, handler)
    signal.alarm(timeout)

    try:
        resultado = funcao()
        signal.alarm(0) # Cancelar alarme
        return resultado
    except TimeoutError:
        logging.error(f"Função {funcao.__name__} excedeu timeout")
        raise
```

Exercícios

Exercício 1: Implemente uma função `validar_senha(senha)` que verifica se uma senha atende critérios de segurança (comprimento mínimo, caracteres especiais, números, maiúsculas). Use programação defensiva para validar todas as entradas.

Exercício 2: Crie uma classe `CacheSeguro` que armazena dados em memória com validação de chaves, limites de tamanho e proteção contra modificação de valores armazenados.

Exercício 3: Desenvolva uma função `mesclar_dicionarios(*dicts)` que mescla múltiplos dicionários defensivamente, sem modificar os originais e tratando conflitos de chaves.

Desafios

Desafio 1: Implemente um decorador `@safe_execution` que envolve funções com múltiplas camadas de proteção: validação de tipos baseada em type hints, timeout, tratamento de exceções, logging e retry automático.

Desafio 2: Crie uma classe `TransacaoSegura` que implementa o padrão de transação com commit/rollback, garantindo consistência de estado mesmo em caso de falhas parciais.

Desafio 3: Desenvolva um sistema de validação configurável por schema (similar ao JSON Schema) que valida estruturas de dados complexas com mensagens de erro detalhadas.

Aplicações

1. **APIs públicas:** Validação rigorosa de todas as entradas de usuários
2. **Sistemas financeiros:** Proteção contra erros de arredondamento e overflow
3. **Processamento de dados:** Validação de integridade e formato
4. **Sistemas críticos:** Múltiplas camadas de verificação e fallbacks
5. **Integração de sistemas:** Validação de contratos entre componentes

Boas Práticas

1. Validação em camadas

- Valide na entrada (API, UI)

- Valide na lógica de negócio
- Valide antes de persistir dados

2. Falhe rápido e claro

- Detecte problemas cedo
- Forneça mensagens de erro específicas
- Inclua contexto suficiente para debugging

3. Valores padrão seguros

- Use valores que não causem efeitos colaterais
- Nunca use listas/dicts mutáveis como padrão
- Documente valores padrão claramente

4. Proteção de estado

- Use propriedades com validação em setters
- Faça cópias defensivas de objetos mutáveis
- Mantenha invariantes de classe

5. Tratamento de recursos

- Sempre use context managers para recursos
- Garanta limpeza mesmo em caso de erro
- Implemente timeouts para operações demoradas

6. Logging adequado

- Registre erros com contexto completo
- Use níveis de log apropriados
- Não exponha informações sensíveis em logs

7. Documentação

- Documente pré-condições e pós-condições
- Liste exceções que podem ser lançadas
- Explique comportamento em casos extremos

```
# Exemplo completo de função defensiva bem implementada

import logging
from typing import List, Optional
from decimal import Decimal, InvalidOperation

def calcular_total_compra(
    itens: List[dict],
    desconto_percentual: float = 0.0,
    taxa_entrega: Optional[float] = None
) -> Decimal:
    """
    Calcula total de compra com validações defensivas completas.
    """

    Args:
        itens: Lista de dicionários com 'preco' e 'quantidade'
        desconto_percentual: Desconto em porcentagem (0-100)
        taxa_entrega: Taxa de entrega opcional
    
```

Returns:

Total da compra como Decimal

Raises:

TypeError: Se tipos de entrada forem inválidos
 ValueError: Se valores estiverem fora de ranges válidos
 KeyError: Se itens não tiverem chaves necessárias

Examples:

```
>>> itens = [{'preco': 10.0, 'quantidade': 2}]
>>> calcular_total_compra(itens)
Decimal('20.00')

"""
# 1. Validação de tipos
if not isinstance(itens, list):
    raise TypeError(f"itens deve ser lista, recebido: {type(itens)}")

if not isinstance(desconto_percentual, (int, float)):
    raise TypeError(f"desconto deve ser numérico, recebido: {type(desconto_percentual)}")

```

```
if taxa_entrega is not None and not isinstance(taxa_entrega, (int, float)):
    raise TypeError(f"taxa_entrega deve ser numérica ou None")

# 2. Validação de valores
if not itens:
    raise ValueError("Lista de itens não pode ser vazia")

if not (0 <= desconto_percentual <= 100):
    raise ValueError(f"Desconto deve estar entre 0 e 100: {desconto_percentual}")

if taxa_entrega is not None and taxa_entrega < 0:
    raise ValueError(f"Taxa de entrega não pode ser negativa: {taxa_entrega}")

# 3. Processar com Decimal para evitar erros de ponto flutuante
try:
    subtotal = Decimal('0')

    for i, item in enumerate(itens):
        # Validar estrutura do item
        if not isinstance(item, dict):
            raise TypeError(f"Item {i} deve ser dicionário")

        if 'preco' not in item or 'quantidade' not in item:
            raise KeyError(f"Item {i} deve ter 'preco' e 'quantidade'")

        # Converter e validar
        preco = Decimal(str(item['preco']))
        quantidade = int(item['quantidade'])

        if preco < 0:
            raise ValueError(f"Item {i}: preço não pode ser negativo")

        if quantidade <= 0:
            raise ValueError(f"Item {i}: quantidade deve ser positiva")

        subtotal += preco * quantidade

    # 4. Aplicar desconto
    desconto_decimal = Decimal(str(desconto_percentual)) / Decimal('100')
```

```

total = subtotal * (Decimal('1') - desconto_decimal)

# 5. Adicionar taxa de entrega
if taxa_entrega is not None:
    total += Decimal(str(taxa_entrega))

# 6. Validar resultado final
if total < 0:
    logging.warning("Total calculado é negativo, ajustando para 0")
    total = Decimal('0')

# 7. Arredondar para 2 casas decimais
total = total.quantize(Decimal('0.01'))

return total

except InvalidOperation as e:
    raise ValueError(f"Erro ao processar valores numéricos: {e}")

except Exception as e:
    logging.error(f"Erro inesperado no cálculo: {e}")
    raise

```

Resumo e Checklist de Boas Práticas

Checklist de Code Review

Ao revisar código, verifique:

- [] Todas as entradas são validadas?
- [] Tipos são verificados quando necessário?
- [] Ranges de valores são validados?
- [] Exceções apropriadas são usadas?
- [] Mensagens de erro são claras e açãoáveis?
- [] Recursos são liberados adequadamente?
- [] Há tratamento para casos extremos?

- [] Código falha de forma segura?
- [] Invariantes são mantidos?
- [] Há logging apropriado?
- [] Documentação menciona exceções?
- [] Cópias defensivas são feitas quando necessário?

Hierarquia de Defesas

1. **Prevenção:** Design que previne erros (tipos, interfaces claras)
2. **Validação:** Verificação de entrada e estado
3. **Detecção:** Assertions e invariantes
4. **Tratamento:** Exceções bem tratadas
5. **Recovery:** Fallbacks e estados seguros
6. **Monitoramento:** Logging e alertas

Princípios Fundamentais

SOLID aplicado ao tratamento de erros:

- **Single Responsibility:** Cada exceção tem propósito único
- **Open/Closed:** Hierarquias de exceções extensíveis
- **Liskov Substitution:** Subtipos mantêm contratos de exceção
- **Interface Segregation:** Interfaces declaram exceções específicas
- **Dependency Inversion:** Dependa de abstrações de erro, não implementações

Fail-Safe vs Fail-Fast:

- Use **fail-fast** durante desenvolvimento e testes para expor bugs rapidamente.
- Use **fail-safe** em produção apenas quando houver um plano claro de recuperação.
- Fail-safe sem monitoramento transforma erros em problemas silenciosos.

Use fail-fast durante desenvolvimento para detectar bugs rapidamente. Use fail-safe em produção para manter o sistema operacional quando possível.

Conclusão

O tratamento adequado de erros e a adoção de práticas de código seguro são muito importantes para a qualidade do software. A combinação de:

- **Exceções bem definidas** usadas para comunicar falhas de forma clara
- **Exceções customizadas** para domínio específico
- **Assertions** limitadas à verificação de invariantes internos
- **Programação defensiva** para prevenir problemas

Resulta em código com menor taxa de falhas em produção, diagnósticos mais rápidos, contratos explícitos e menor custo de manutenção ao longo do tempo.

Contrato *consiste de um conjunto de regras formais e verificáveis que definem como o código deve ser usado e o que ele garante em troca.*

Código seguro não se resume a reagir a erros. Ele começa no projeto, reduz a chance de falhas e deixa claro como o sistema deve se comportar quando algo sai do esperado.