Sprint 02

Maycow

05/17

Resumo

Neste relatorio se encontrará maneiras de reduzir erros em programas com maior nível de complexidade utilizando funções e módulos do python

Chapter 10: Debugging

1 Raise exceptions

De maneira geral, utilizar raise exceptions é um modo de dizer ao computador: “pare de rodar o programa nessa função e vá para a declaração except. A declaração no programa é dada da seguinte maneira:

raise Exception(‘mensagem de erro’)

Geralmente utilizamos raise exception em combinação com try/except , sendo o raise Exception('frase de erro qualquer') dentro da Função e fora dela quando formos chamá-la utilizamos um try função / except Exception as 'nome da variável de erro' , depois printamos a variável de erro.veja um exemplo de uso no programa a seguir:

```
def boxPrint(symbol, width, height):
    if len(symbol) != 1:
      raise Exception('Symbol must be a single character string.')
    if width <= 2:
      raise Exception('Width must be greater than 2.')
    if height <= 2:
      raise Exception('Height must be greater than 2.')
    print(symbol * width)
    for i in range(height - 2):
       print(symbol + (' ' * (width - 2)) + symbol)
    print(symbol * width)

for sym, w, h in (('*', 4, 4), ('O', 20, 5), ('x', 1, 3), ('ZZ', 3, 3)):
    try:
       boxPrint(sym, w, h)
for except Exception as err:
       print('An exception happened: ' + str(err))
```

O programa tentará rodar a função caso alguma das condições estabelecidas dentro da função não seja obedecida colocamos a mensagem de erro contida no raise Exception e armazenamos tal mensagem numa variável que nesse programa chama-se err, então printamos a variável onde foi armazenada a string de erro.

2 Getting the Traceback as a string

Traceback é um pacote de informações que traz uma mensagem de erro, a linha onde ocorreu o erro e a sequencia de chamada das funções que causaram erro, quando ele aparece seu programa trava. podemos pegar essas série de informações úteis e escrever num arquivo texto para termos acesso a tais informações num arquivo separado. fazendo da seguinte forma:

```
import traceback
 try:
        raise Exception('This is the error message.')
except:
        errorFile = open('errorInfo.txt', 'w')
        errorFile.write(traceback.format_exc())
        errorFile.close()
        print('The traceback info was written to errorInfo.txt.')
```

3 Assertions

È uma maneira de checar se seu código está fazendo algo errado, use a palavra assert, uma condição, uma virgula e uma frase para ser printada se sua condição for falsa:

```
assert door == 'open', 'The door need to be "open".'
```

Devemos usar quando nosso código não possui erros de execução, mas erros lógicos que não são detectados e quebram o programa imprimindo uma traceback, veja o exemplo onde o programa a seguir implementa a lógica de um semáfaro, porém existe um erro lógico.

```
def switchLights(stoplight):
    for key in stoplight.keys():
        if stoplight[key] == 'green':
            stoplight[key] = 'yellow'
        elif stoplight[key] == 'yellow':
            stoplight[key] = 'red'
        elif stoplight[key] == 'red':
            stoplight[key] = 'green'

switchLights(market_2nd)
```

Imagine que o restante do programa tenha centenas de linhas de código, no fim das contas o programa irá rodar mas não será capaz de regular de maneira correta os semáfaros, porém o uso de um assertion é capaz de ajudar o programador a encontrar tal erro de maneira fácil.

A seguinta linha de código faria o programa quebrar gerando uma traceback indicando onde o erro ocorreu:

```
assert 'red' in stoplight.values(), 'Neither light is red! ' + str(stoplight)
```

Asserrtions servem para o programa em desenvolvimento e não para o produto final, após pronto, quando verificarmos que não existem erros lógios podemos usar a opção -O para desativar o assertion com intuito de melhorar a performace do programa.

4 Logging

O módulo logging serve basicamente para nos exibir mensagens a respeito do que está ocorrendo no nosso programa, imagine que queremos o fatorial de um número e o nosso resultado final está errado, podemos usar o loggin para exibir mensagens na tela a respeito de cada iteração que fizermos, com isso podemos constatar onde nossa lógica falhou. como usar:

```
import logging
logging.basicConfig(level=logging.DEBUG,format='%(asctime)s
%(levelname)s%(message)s')
```

precisamos fazer um import e formatar nossa mensagem de erro, normalmente podemos apenas copiar a linha de código acima, e usar no nosso programa a função logging.debug('mensagem') para printar os passos do programa.
Não devemos usar o print() para substituir o loggin, pois quando acabarmos o programa e estiver tudo ocorrendo conforme o planejado podemos desativar as mensagens printadas pelo loggin usando:

```
logging.disable(logging.CRITICAL)
```

Se usassemos print() teriamos que apagar todos os prints usados para mensagens de erro um por um, pois tais mensagens servem para o programados na fase do desenvolvimento e não para o produto final, devemos deixar no programas apenas mensagens que interessem ao usuário.
Existem 5 categorias no módulo logging, sendo elas:

1- logging.debug(),
2- logging.info() ,
3- logging.warning(),
4- logging.error(),
5- logging.critical(),

Sendo 1 a menos importante em relação a prioridade no programa e 5 a mais importante, usando isso podemos definir uma ordem de prioridade de mensagens passadas pela função logging para concentrar a correção de erros nas partes mais criticas. Dentro da nossa função logging.basicconfig() escolhemos o level minimo de erros que vamos mostrar, se no momento quisermos desprezar o nivel 1 e 2, escreveremos loggin.warning(), quando a função logging.disable for chamada será desativado o nivel escrito dentro dela e todos os outros niveis menores.

podemos salvar nossas mensagens de erro num arquivo txt para manter nossa tela limpa, usando o argumento filename='nome do arquivo' dentro da função basic config:

```
import logging
logging.basicConfig(filename='myProgramLog.txt',level=logging.DEBUG, format='-
-%(asctime)s - %(levelname)s - %(message)s')
```

5 Idle’s Debbuger

O idle do Python possui uma ferramenta que complementa a verificação de erros, tal ferramenta é chamada idle's debbuger, para ativa-la é necessário clicar em debbug e depois em debbuger, após isso uma janela será aberta, é bom que marquemos as quatro opções disponíveis, a janela do idle's debbuger começa na primeira linha do programa e apartir dela podemos rodar linha a linha caso necessário ou podemos rodar o programa rapidamente ate determinado ponto que nos interessa. O idle's debbuger possui 5 cinco botões, sendo eles: go, step, over, out e quit.

O botão step serve para exercutarmos a próxima linha do programa, quando a próxima linha é uma função o idle's debbuger entra na primeira linha da função e a cada step que damos ele percorre a função.

O botão over tem a mesma função que o botão step, porém quando a.próxima linha é uma função o idle's debbuger pula a função e passa para linha imediatamente posterior ao retorno da função.

O botão out serve para sairmos de uma função quando estivermos entrado na mesma com o botão step, após verficarmos determinado erro dentro da função podemos utilizar o botão out para sair o dela e passar para linha imediatamente posterior a linha onde se encontra a chamada da função.

O botão quit serve para interromper e terminar o programa independente da linha onde esteja.

O botão go fará com que o programa seja executado normalmente até encontrar um breakpoint. Breakpoint São linhas que escolhemos como referencia durante o desenvolvimento do código , basta clicar em cima de uma determinada linha com o botão direito e clicar em set breakpoint

que essa linha será uma referência na hora de utilizar o idle's debbuger, o programa executará normalmente até atingir o breakpoint.