Manual de Uso dos Corretores Automáticos de Programas Python

CAPP

Gabriel Ribeiro

Orientado por José Lopes de Siqueira Neto,
Professor da UFMG.

Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
Brasil
16 de julho de 2024

Dedico este trabalho a todos os professores empenhados ao Ensino. Em particular ao meu orientador José Lopes de Siqueira Neto.

"Don't be a drag, just be a queen
Whether you're broke or evergreen
You're black, white, beige, chola descent
You're Lebanese, you're orient
Whether life's disabilities
Left you outcast, bullied or teased
Rejoice and love yourself today
'Cause baby, you were born this way
No matter gay, straight or bi
Lesbian, transgendered life
I'm on the right track, baby
I was born to survive
No matter black, white or beige
Chola or orient made
I'm on the right track, baby
I was born to be brave"

Lady Gaga

Obrigado a todos que acreditaram em mim até aqui.
Essa pesquisa de Iniciação Científica e código a mim muito significam. Neles, pude mostrar meu potencial enquanto matemático, programador e pesquisador, além de iniciar a conquista de um sonho de criança: ser um cientista!
Obrigado por se interessar em meu primeiro trabalho, caro(a) leitor(a).

Carinhosamente, Gabriel R.

Sumário

1 Público Alvo dos Corretores e Usabilidades

A SER ESCRITO

2 Módulos do CAPP-L

Esta secção é dedicada à explicação das funcionalidades dos módulos contidos e usados pelo CAPP-L. Aqui, serão apresentados exemplos de usos e apresentadas algumas interações entre eles.

2.1 Config

Este módulo possui 4 variáveis base, que são utilizadas por todas as partes do Corretor:

- **NOMES_EXES:** É uma lista de strings contendo os nomes de todos os exercícios que devem ser corrigidos.
- **TESTES_EXES:** É uma lista de strings contendo os nomes das variáveis que contém os testes de cada exerício. Isso será melhor explicado na seção Gabarito e Geradoras de Gabarito. É importante ressaltar que, por exemplo, o teste correspondente a <exercício_X> é <teste_X>.
- **NOMES_EXES_INPUT:** É uma lista de strings contendo os nomes dos exerícios que requerem input do usuário, isto é, possuem a função input().
- **NUM_AULA:** É uma string contendo o nome e número da aula, na forma '_aula_X', sendo X um número.

Exemplo de configuração de uma aula:

```
NOMES_EXES_INPUT = ['exercicio_1_1','exercicio_1_2','exercicio_5_1','exercicio_5_2',
                    'exercicio_5_3', 'exercicio_8_1', 'exercicio_8_2']

aula_7 = [1, [2], 2, [2], 3, [2], 4, 5, [3], 6, [2], 7, [2], 8, [2]]

NOMES_EXES = gn.gera_nomes('exercicio_',aula_7)

TESTES_EXES = gn.gera_nomes('teste_',aula_7)

# a função gera_nomes está explicada na secção 1.2 Geradoras deste capítulo.

NUM_AULA = '_aula_7'
```

2.2 Geradoras

Este módulo foi criado para auxiliar na geração de testes para exercícios. Aproveite-o e utilize de suas funções para criar testes aleatórios para as funções de seus alunos, escrever menos ou até mesmo automatizar tarefas.

- **Geradora de Faixas Aleatórias(GFA):**
  Aqui, existem duas funções expecializadas na geração de números aleatórios em determinados intervalos.
  - **gia(entrada: list, semente: int) $\longrightarrow$ list[list]**
    Geradora de Intervalos Aleatórios.
    Função que gera números dentro de intervalos uma única vez.
    entrada: lista, formato: [total1, [inicio1, final1, step1], ...], sendo totalX a quantidade de números que se quer gerar no intervalo X. Inicio e Total definem um intervalo fechado e Step o passo para se dividir o intervalo. Quando a lista que contém [inicioX, finalX, stepX] tem algo (em ordem) omitido, é considerada a configuração padrão [1, 101, 1] para a variável que está faltando. A quantidade de números que se quer gerar deve sempre ser especificada dentro de uma lista. Retorna uma lista de listas, contendo os números gerados nos respectivos intervalos.
    semente: semente para a randomização.

```
>>> ex = [10, [0, 100, 10], 5, [200, 400], 2, [30], 5,
6, [3, 4, 5, 6], 6]
>>> print(gia(ex, 1))
output: [[94, 15, 67, 10, 28, 38, 44, 80, 58, 88],
[389, 222, 237, 360, 372], [22, 23],
[61, 73, 78, 51, 29], [67, 64, 11, 54, 84, 71]]

>>> ex = [10, [200]]
>>> print(gia(ex, 1))
output: [[35, 146, 196, 17, 66, 31, 127, 195, 116, 121]]
```

```
>>> ex = [20]
>>> print(gia(ex, 1))
output: [[36, 89, 14, 52, 75, 48, 95, 74, 68, 77, 33,
10, 7, 15, 23, 28, 29, 85, 38, 46]]
```

- **isr(geradora: list, rep: int = 0) ⟶ list[list[list]]**
  Inicializadora para Sequências com Repetição.
  Função que gera números aleatórios dentro de intervalos repetidas vezes.
  Dada a lista de intervalos, ela chama a função GIA (Geradora de Intervalos Aleatórios), <rep> vezes.
  Retorna uma lista que possui listas, as quais representam os números gerados em cada repetição para os parâmetros repetidos.

  geradora: Lista a ser lida para a geração de números aleatórios. A lista pode conter intervalos da forma n, [inicio, final, step] e/ou n, [inicio, final] e/ou n, [final] e/ou n.
  rep: Parâmetro opcional para repetição da geradora.

```
>>> ex = 3*[10, [0, 100, 10], 5, [200, 400], 2, [30],
5, 6, [3, 4, 5, 6], 6]
>>> print(isr(ex))  # repetição está em ex (3*)
output: [[[80, 21, 6, 58, 16, 47, 61, 86, 91, 33],
[283, 358, 287, 276, 331], [14, 26],
[81, 91, 17, 28, 32], [77, 85, 28, 84, 72, 73]],
[[80, 21, 6, 58, 16, 47, 61, 86, 91, 33],
[283, 358, 287, 276, 331], [14, 26],
[81, 91, 17, 28, 32], [77, 85, 28, 84, 72, 73]],
[[80, 21, 6, 58, 16, 47, 61, 86, 91, 33],
[283, 358, 287, 276, 331], [14, 26],
[81, 91, 17, 28, 32], [77, 85, 28, 84, 72, 73]]]

>>> ex = [10, [0, 100, 10], 5, [200, 400], 2, [30],
5, 6, [3, 4, 5, 6], 6]
>>> print(irs(ex, 3))  # repetição especificada como argumento (3)
output: [[[63, 90, 52, 97, 73, 34, 44, 9, 26, 19],
[280, 363, 304, 257, 320], [21, 4],
[101, 31, 89, 11, 20], [98, 87, 66, 19, 30, 89]],
[[63, 90, 52, 97, 73, 34, 44, 9, 26, 19],
[280, 363, 304, 257, 320], [21, 4],
[101, 31, 89, 11, 20], [98, 87, 66, 19, 30, 89]],
[[63, 90, 52, 97, 73, 34, 44, 9, 26, 19],
[280, 363, 304, 257, 320], [21, 4],
[101, 31, 89, 11, 20], [98, 87, 66, 19, 30, 89]]]
```

- **Geradora Aleatória de Strings (GAS):**
  Essa geradora é especializada em gerar strings de maneira aleatória.

  - **gas(n: int, modo: str, palindromo: bool = False, semente: int = 0) ⟶ str**
    n: Número de caracteres aleatórios que se deseja gerar.
    modo: Modo de geração, é uma concatenação de Strings:
    * 1) '' -> A string vazia configura a geração para misturar letras maiúsculas e minúsculas.
    * 2) 'm' -> Apenas letras minúsculas na string gerada.
    * 3) 'M' -> Apenas letras maiúsculas na string gerada.
    * 4) '+[]' -> Todas as strings dentro de [] serão colocadas, em ordem aleatória, na string gerada. Exemplo: '+['abc', 'igh]' -> A string gerada possuirá letras maiúsculas e minúsculas, e conterá 'abc' e 'igh'.

    palindromo: Se for True, a string gerada será um palíndromo.
    semente: Opcional, se não for informado será 0.

    Segue alguns exemplos de execução:

```
>>> print(gas(51, '+["abc", "IGH"]', True, 10))
output: dabcWzTKeMqbzsiIIGHhGqjnXCfjROxdxORjfCXnjqGhHGIIiszbqMeKTzWcbad
```

```
>>> print(gas(4, 'M+["pow", "xyz"]', False, 10))
output: OpowFxyzLP

>>> print(gas(23, 'm', True, 10))
output: evvlponfgqioiqgfnoplvve

>>> print(gas(2, '+["algo", "talvez"]', True, 10))
algotalvezddzevlatogla

>>> print(gas(4, 'm+["AAAAAA", "11111"]', True, 10))
output: AAAAAAl11111oo11111lAAAAAA
```

- **Geradora de Nomes:**
  Essa geradora é utilizada para facilitar a geração de nomes de exercícios.
  - **gera_nomes(nome: str, exercicios: list) $\longrightarrow$ list**
    Gera uma lista de strings, formadas pelos argumentos <nome> e exerícicios em <exercicios>. É uma função wrapper das outras contidas em gera_nomes.py.

    nome: str, a string que será usada para gerar os nomes.

    exercicios: list, lista contendo o número dos exercícios para os quais se quer gerar os nomes. É uma lista cujos elementos são inteiros ou listas. Os números representam níveis e as listas seguintes seus subníveis. Por exemplo: [1, [2, 3], [4]] representa o nível 1, com subníveis de 2 a 3, cada com sub-subníveis 1 a 4. Veja alguns exemplos de execução.

```
>>> ex = [10]
>>> print(gera_nomes('teste_', ex))
output: ['teste_10']

>>> ex = [1, [2]]
>>> print(gera_nomes('teste_', ex))
output: ['teste_1_2', 'teste_1_2']

>>> ex = [1, [2, 3], [4]]
>>> print(gera_nomes('teste_', ex))
output: ['teste_1_2_1', 'teste_1_2_2', 'teste_1_2_3', 'teste_1_2_4',
'teste_1_3_1', 'teste_1_3_2', 'teste_1_3_3', 'teste_1_3_4']

>>> ex = [1,[2],2,4,[9],[2],4,[10,10],4,[11,12],[2]]
>>> print(gera_nomes('teste_', ex))
output: ['teste_1_1', 'teste_1_2', 'teste_2', 'teste_4_1_1',
'teste_4_1_2', 'teste_4_2_1', 'teste_4_2_2', 'teste_4_3_1',
'teste_4_3_2', 'teste_4_4_1', 'teste_4_4_2', 'teste_4_5_1',
'teste_4_5_2', 'teste_4_6_1', 'teste_4_6_2', 'teste_4_7_1',
'teste_4_7_2', 'teste_4_8_1',  'teste_4_8_2', 'teste_4_9_1',
'teste_4_9_2', 'teste_4_10', 'teste_4_11_1', 'teste_4_11_2',
'teste_4_12_1', 'teste_4_12_2']
```

2.3 Arquivos

É neste módulo onde é feita a principal tarefa de tratamento dos arquivos submetidos. De forma geral, o algoritmo de tratamento funciona da seguinte maneira:

- O(s) arquivo(s) .zip, baixado(s) do Moodle, contendo as submissões dos alunos é/são colocado(s) na pasta do CAPP-L.
- O módulo Arquivos reconhece o(s) arquivo(s) .zip e o unzipa.
- É criada uma pasta Alunos contendo as respectivas turmas de cada arquivo zip. Dentro dessas turmas, são criadas 5 pastas: Notebooks, Scripts, Intercorrências, Correção e Outros. As turmas de cada zip são reconhecidas por meio de _TX no nome do arquivo.
- Os nomes dos arquivos são modificados para serem iguais ao do aluno que os submeteu.

- Os notebooks (.ipynb) e os Scripts (.py) são movidos para suas respectivas pastas. Qualquer outro tipo de arquivo será movido para a pasta Outros, essa pasta não será utilizada.
- É criado um dicionário roteiro, da forma:

```
roteiro = {'TX': (<converter> = True | False, <corrgirir> = True | False),
           'TY': (...,), ...}
```

- Caso não haja nenhum script em Scripts, os Notebooks, caso exista algum, serão convertidos para arquivos .py. Aqueles notebooks que não puderem ser convertidos, serão copiados para Intercorrências com um novo nome: nome_original_SEM_RAW.ipynb - e terão suas células raw apagadas. Este novo arquivo tentará ser convertido de novo. Caso a conversão para .py falhe de novo, o notebook não será convertido. Se houver Scripts para aquela turma, a conversão de nenhum notebook será realizada.
- Para as turmas que havia scripts e as turmas que tiveram seus notebooks convertidos, em roteiro, seus respectivos <corrigir> se tornam verdadeiros e <converter> se torna falso.

2.4 Gabarito e Geradoras de Gabarito

Neste módulo é realizado uma parte fundamental do processo de correção: manutenção do gabarito - afinal, não é possível corrigir nada sem a presença de algo que diga os resultados esperados. Desta forma, este módulo se dedica a manutenção das variáveis de teste, criação, formulação e execução de gabarito. Ele é composto da seguinte maneira:

- **funcoes_gabarito.py:** Neste arquivo devem estar as funções exigidas dos alunos em suas submissões e seus nomes devem ser aqueles espeficados em NOMES_EXES em <config>. Por exemplo:

```
def exercicio_1_2(n):
"""
Contador modular com for

Parameters
----------
n : int
    Valor do módulo

Returns
-------
None.

"""
resp = ''
while resp != 'N' and resp != 'n':
    resp = ''
    for i in range(n):
        if resp == 'N' or resp == 'n':
            break
        else:
            print(i)
            resp = input('Quer continuar? ([Ss]/Nn) ')
print('Fim')
return None
```

Caso não seja desejado gerar os testes de maneira aleatória, este arquivo também deve conter variáveis com os nomes listados em TESTES_EXES, as quais contém os testes para cada exercício, no qual cada teste deve conter o mesmo número da função exercício a que ele corresponde. Para a função listada anteriormente, segue um exemplo:

```
ent = ['S','s','','S','s','','N']
teste_1_2 = [((1,),ent),((2,),ent),((3,),ent),((4,),ent),((5,),ent)]

teste_4 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 19, 20, 33, 46, 98, 120, 360]
```

Os argumentos de teste devem sempre ser uma tupla de iteráveis, onde o primeiro iterável deve ser os argumentos que se quer passar para a função e o segundo, caso precise, é os inputs de usuário para o qual se quer simular. Isso não é obrigatório quando o exercício não possui inputs (vide teste_4), mas é uma boa prática.

```
teste_X: list[tuple[tuple]] = [((args,), (inputs,)), ...]
```

É importante dizer que o CAPP-L tentará importar e gerar gabaritos para as funções e testes especificados em <config>, caso algo apresente erro, será determinado erro fatal e não será possível executar a correção.

- **gera_gabarito.py:** Aqui, são importadas as funções e variáveis (as quais o CAPP-L foi configurado) de funcoes_gabarito.py. Para executarmos as funções e obtermos seus resultados utilizamos a executora de scripts em ICA, afinal, funcoes_gabarito.py é um script python. Caso haja algum erro de importação, a geradora declarará erro. A geradora cria um gabarito com o seguinte corpo:

```
gabarito = {'exercicio_X': {((args,), (inputs,): (output, prints), ...},
            'exercicio_Y': {...}, ...}
```

Onde output é a saída da função exercicio_X para determinado teste e prints é uma lista de strings do que foi printado durante a execução, na ordem que foram impressos.

Ao criar a variável gabarito, geramos um arquivo gabarito.py na pasta ICA.

- **geradora_de_gabaritos_aleatorios.py:** Aqui utilizaremos algumas funções expecificadas em Geradoras para nos ajudar a randomizar a geração dos gabaritos.

  Para que seja possível gerar o gabarito, precisamos informar para a geradora duas coisas: como serão os testes que gostaríamos de gerar para determinados exercícios e quantos gabaritos aleatórios gostaríamos de gerar. Para os testes , a geradora obedece à uma variável denominada manual.

  Manual é um dicionário que assimila o nome dos testes à maneira que eles devem ser gerados. Para isto, ele possui uma estrutura bastante rígida:

  – Geração de Strings: Quando se quer assimilar um teste à uma geração de strings, a GGA utilizará a GAS. Portanto, para determinado teste, associa-se uma tupla da forma:

  ```
  'teste': (número de testes, número de argumentos por teste, 's',
            [tamanho da string, modo de geração da GAS, palíndromo],
            inputs)
  ```

  Os itens da lista são os argumentos que se deseja passar para a GAS e, por isso, devem ser listados em ordem, corretamente e sem omissão.

  – Geração de Números: Quando se quer assimilar um teste à uma geração de números inteiros, a GGA utilizará a GIA. Assim, será associado uma tupla da forma:

  ```
  'teste': ('n', número de testes, [argumentos por teste,
            [ínicio do intervalo, fim do intervalo, passo]],
            inputs)
  ```

Para este exemplo, suponha que <exercício_X> é uma função que requer 2 argumentos inteiros e <exercício_Y> uma função que requer uma string como argumento e requer inputs. Para cada, queremos aplicar 7 e 10 testes, respectivamente. Portanto, escreveríamos em Manual o seguinte:

```
manual = {'teste_X': ('n', 7, [2, [0, 100, 10]]),
          'teste_Y': (10, 1, 's', [5, 'm+["abc"]', False], ['a', 'A', 'b', 'c'])}
```

Com este manual, GGA entende que, para o teste_X, devem ser gerados 7 testes numéricos, onde, em cada teste, devem haver dois números do intevalo de 0 a 100, com passo 10 e, para o teste_Y, que devem ser 10 testes de strings, cada deve ter 1 argumento, no qual cada argumento é: uma string de tamamnho 5, de caracteres minúsculos, contendo "abc" e não é um palíndromo junto à lista de inputs ['a', 'A', 'b', 'c'].

A geradora de gabaritos aleatório cria um arquivo gabarito.py em ICA contendo todos os gabaritos gerados, seus nomes são sufixados pela semente usada em sua geração.

É importante ressaltar que a geração aleatória de gabaritos é totalmente aleatória, para testes mais específicos, recomenda-se a geradora de gabaritos não aleatória. Independente de qual geradora de gabaritos for escolhida, o gabarito será apenas criado pelo CAPP-L caso ele não exista.

2.5 Itens da Correção Automática: ICA

Este módulo é dedicado aos itens necessários para que o corretor em lotes funcione: funções de correção e execução de scripts, assim como a presença do gabarito de correção.

2.5.1 Execução dos Scripts

Análogo à maneira de geração do Gabarito, importamos o script que desejamos corrigir usando importlib, pegamos as funções que queremos usando getattr() e simulamos inputs e prints usando unittest.mock.patch, os argumentos vêm diretamente de gabarito.py, que será importado e guiará a maneira que iremos corrigir os exercícios.

2.5.2 Aplicando Testes Um a Um

```
executa_scripts_1_teste_por_exercicio(script: str, nome_funcao: str,
                                      teste, inputs=None) -> tuple
```

Essa função é responsável por executar os testes e obter seus resultados sobre os scripts. É chamada pelo CAPP-L para executar os Scripts em Lotes, usando multiprocessing, isso será explicado na próxima secção.

O seu algorítmo de execução é bastante simples:

- 1) Dado o caminho para o Script, importamos-o usando importlib.

```
py_code = importlib.import_module(script)
```

- 2) Dado o nome da função, obtemos ela enquanto atributo do script.

```
funcao = getattr(py_code, nome_funcao)
```

- 3) O iterável <teste>, passado como argumento, é utilizado como argumento da função do aluno.
- 4) O iterável <inputs>, caso haja, será aplicados na função.
- 5) A função retorna o que foi obtido dessa triagem, em forma de Tupla, contendo o resultado e o que foi printado.

```
if inputs:
    with unittest.mock.patch(target='builtins.input', side_effect=inputs):
        with unittest.mock.patch(target='builtins.print',
                                 side_effect=lambda s: prints.append(s)):

            resultado = funcao(*teste)
else:
    with unittest.mock.patch(target='builtins.print',
                             side_effect=lambda s: prints.append(s)):

        resultado = funcao(*teste)
```

script: É o caminho para importarmos o arquivo .py do aluno, sem a extensão do arquivo.

```
Alunos.TX.Scripts.nome_do_aluno
```

nome_funcao: Nome da função que desejamos importar para testá-la.

teste: Deve ser um iterável de algum tipo, por padrão, em gabarito, é uma tupla.

inputs: Opcional, deve ser um iterável de algum tipo. Utilizar apenas caso a função tenha inputs.

Retorna uma tupla contendo retorno da função do aluno para dado teste ou o erro obtido, juntamente ao que foi printado durante a execução

2.5.3 Corrigindo o Script Executado

```
compara_respostas(gabarito: dict, respostas_script: dict, turma: str = None,
                  nome_aluno: str = None) -> tuple
```

Essa função compara as respostas obtidas pelos scripts dos alunos com o Gabarito. A nota de cada acerto é $100/n\_testes$. O resultado do script do aluno é transformado em um dicionário de chaves cujos nomes são os exerícios. Cada chave possui um dicionário associado, sendo cada uma delas os testes aplicados, assimiladas ao resultado retornado. Gabarito é de mesma estrutura. A chave da comparação está no gabarito, afinal é nele que está a iteração. Caso o exercício requeira inputs, na comparação, os prints obtidos em <respostas_script> e gabarito serão comparados.

gabarito: Gabarito que contém os resultados esperados para determinados testes.
respostas_script: Dicionário de respostas do aluno para cada exercício.
turma: Turma a qual o aluno pertence
nome_aluno: Nome do aluno.
Retorna uma tupla, contendo: nome, nota e acertos do script.
A função, durante a execução, cria um arquivo contendo a correção do script informado. Tal arquivo pode ser de duas formas:
Caso seja informado turma e nome_aluno: será criado um arquivo <nome_aluno>.txt na pasta Alunos/<turma>/Scripts. Contendo a correção do Script.
Independente do caso, a correção é da forma:

```
Aula: <config.NUM_AULA>
Correção de <nome_aluno>

    ========================================================================
    >> exercicio_XXX  # exercício com inputs
    Argumentos       : XXX
    Inputs           : XXX
    ------------------------------------------------------------------------
    Output Esperado  : XXX
    Output Obtido    : XXX
    ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    Prints Esperados : XXX
    Prints Obtidos   : XXX
    ------------------------------------------------------------------------
    Nota obtida      : XXX
    ========================================================================
    >> exercicio_YYY  # exercício sem inputs
    Argumentos       : YYY
    ------------------------------------------------------------------------
    Output Esperado  : YYY
    Output Obtido    : YYY
    ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    Prints           : YYY
    ------------------------------------------------------------------------
    Nota obtida      : YYY

====================================================================
    >>> ACERTOS     : <acertos> / <n_testes>
    >>> NOTA FINAL  : XXX.XXXX

Exercício           Testes    Acertos   Porcentagem
++++++++++++++++++++++++++++++++++++++++++++++++++
exercicio_XXX       XXX       XXX       XXX.XX%
exercicio_YYY       YYY       YYY       YYY.YY%
.                   .         .         .
.                   .         .         .
```

2.6 Estatísticas

Este é o único módulo que sua execução não é utilizada pelo corretor. Aqui, veremos seu conteúdo, sua aplicação será desenvolvida em Gerando Anaálises Estatísticas e Bancos de Dados sobre seus Alunos. Pensando no uso em

sala de aula, é importante manter um histórico de notas dos seus alunos, portando este módulo foi desenvolvido: suas funcionalidades consistem em calcular as médias finais dos alunos e análises de rendimento, gerar planilhas (tabelas CSV) contendo as médias das turmas, notas dos alunos, gerar gráficos sobre o rendimento coletivo e individual.

Envisionando tudo isso, esta secção utiliza de duas funções:

- **cria_csv_por_colunas(dicionario: dict[str: dict[str: Any]]) ⟶ str**
  Essa função é dedicada a criar tabelas CSV por quais elementos deveriam estar em quais colunas. Seu único argumento é um dicionário cujas chaves são strings, estas que estão associadas a outro dicionário, cujas chaves são strings associadas à qualquer outra coisa. Durante a execução deste módulo, esta função recebe um dicionário cujas chaves são nomes de alunos, o dicionário associado aos alunos contém as aulas como chaves e suas notas como valores. Retorna a string da tabela gerada.

  dicionario: Dicionário contendo strings como chaves (nomes das linhas), o valor é outro dicionário (valores das colunas associadas à linha).

  Por exemplo, o argumento:

  ```
  dicionario = {'AlunoA': {'_aula_1': XXXX},
                'AlunoB': {'_aula_1': XXXX, '_aula_2': XXXX, '_aula_3': XXXX},
  ```

  Gera a seguinte string (editada para melhor visualização, porém mantida estrtutura):

  ```
  "Nome dos Alunos ; _aula_1 ; _aula_2 ; _aula_3
   AlunoA          ;   XXXX  ;    00   ;    00
   AlunoB          ;   XXXX  ;   XXXX  ;   XXXX"

   # os valores não citados são tratados como 0.
  ```

- **cria_csv_2linhas(dicionario: dict[str: Any]) ⟶ str**
  Similar à função anterior, esta também gera uma tabela CSV e possui um único argumento - um dicionário. No entanto, esta gera apenas uma tabela de 2 linhas, sendo a primeira o cabeçalho e a segunda, os dados. Essa função é usada pelo Módulo para gerar as planilhas indiduais de cada aluno e a planilha de médias da turma, portanto, o dicionário utilizado é tal cujas chaves são os nomes das aulas e as chaves as notas de determinado aluno em tais.

  dicionario: Dicionário cujas chaves são os nomes das colunas e o valor de tais.

  Segue um exemplo de execução:

  ```
  >>> dicionario = {'_aula_1': XXXX, '_aula_2': YYYY, '_aula_3': ZZZZ}
  >>> print(cria_csv_2linhas(dicionario))
  output: _aula_1 ; _aula_2 ; _aula_3
            XXXX  ;  YYYY   ;  ZZZZ
  ```

Utilizando destas funções, o módulo possui o seguinte algoritmo:

- A cada vez que uma aula é corrigida, é criado, por meio do CAPP-L, um arquivo CSV (.txt) com o nome da aula na pasta CAPP-L/Estatísticas/<turma>/CSVs/. Este arquivo contém os nomes dos alunos, seus acertos e notas.
- Ao ser executado, para cada turma, o módulo lê todas as tabelas CSVs e cria uma pasta em CAPP-L/Estatísticas/<turma> chamada Alunos, nesta pasta são criadas sub-pastas com os nomes dos alunos das respectivas turmas que submeteram, pelo menos, um arquivo em qualquer uma das correções. A pasta de um aluno são criados uma tabela CSV, contendo todas as notas de tal aluno nas aulas que ele participou, e um gráfico de colunas de suas notas.
- O módulo também cria, em CAPP-L/Estatísticas/<turma>, uma pasta contendo os dados da turma. Nesta pasta estão: uma planilha contendo todas as notas de todos os alunos que participaram de pelo menos uma correção, uma planilha contendo as médias de nota da turma para cada aula e um gráfico mostrando as notas médias das turmas em cada aula e um relatório, indicando as notas finais (até a última correção), aulas de maior e menor rendimento e quem já passou no curso (possui nota média final maior que 60).

- O módulo cria, também, em CAPP-L/Estatísticas/<turma>, uma pasta chamada Gráficos das Aulas, contendo os gráficos de barra das notas da turma para cada aula.

Portanto, é gerada uma ostensiva base de dados e planilhas contendo informações de suas turmas e alunos. Sinta-se livre para usar delas para gerar estatísticas e análises mais refinadas do que aquelas realizadas aqui.

3 Configurando o Corretor em Lotes

Nesta secção, veremos como rapidamente configurar o corretor para executar suas correções de uma determinada aula.

3.1 Coloque as variáveis em Config

Vá até config.py e preencha ou escreva as variáveis:

- NUM_AULA = '_aula_X' ⟶ Troque X pelo número da aula que se deseja corrigir
- TESTES_EXES: list[str] ⟶ Coloque na lista as strings nomes dos exercícios que você quer corrigir da forma: 'exercicio_X_Y' (Não é preciso caso o gabarito for gerado com argumentos aletórios)
- NOMES_EXES: list[str] ⟶ Coloque na lista as strings dos nomes das variáveis de teste para cada exercício Por exemplo, o teste de 'exercicio_X_Y' precisa ser nomeado como 'teste_X_Y'.
- NOMES_EXES_INPUT: list[str] ⟶ Coloque na lista as strings dos nomes dos exercícios em NOMES_EXES que possuem a função input.

  Por exemplo:

```
====================== config.py ======================
NUM_AULA = 1
NOMES_EXES = ['exercicio_1', 'exercicio_2', ...]
TESTES_EXES = ['teste_1', 'teste_2', ...]
NOMES_EXES_INPUT = ['exercicio_2']
```

3.2 Escreva as funções e testes do Gabarito

Vá para a pasta Gabarito e vá para funcoes_gabarito.py, lá, escreva as funções exercícios que você mandou seus alunos resolver. Lembre-se das strings colocadas em config.NOMES_EXES, elas devem ser os nomes das respectivas funções associadas - assim como as strings colocadas em config.TESTES_EXES devem ser colocadas como os testes dos respectivos exercícios. (Liste os testes aqui apenas se você não for usar a geração aleatória.) Por exemplo:

```
====================== funcoes_gabarito.py ======================
def exercicio_1(a: int):
    .
    .
    .
    return ...

teste_1 = [(1,), (2,), (3,), ...] # Coloque aqui os argumentos para os quais
                                  # você quer testar no exercício durante
                                  # a correção

def exercicio_2(a: int, b: str):
    .
    .
    .
    i = input(...)
    .
    .
    .
    return ...

teste_2 = [((1, 'a'), ('x',)), ((2, 'b'), ('y')), ...]
# os testes devem sempre ser uma tupla de até 2 iteráveis. O primeiro iterável sempre será
# o argumento da função, o segundo iterável é o que será passado como input na função input()
.
.
.
```

Existem algumas ressalvas que devem ser tomadas sobre a maneira com que os argumentos devem ser colocados:

- Para exercícios com input, o formato padrão da tupla deve ser seguido à risca.
- Por padrão, cada elemento dentro da lista de testes deve ser uma tupla de até 2 iteráveis. No primeiro iterável estão os argumentos que devem ser passados para a respectiva função. No entanto, caso a função requeira uma lista ou tupla como argumento, deve-se explicitar tal dentro do iterável. Por exemplo, suponha que, para determinado teste X queremos passar n, [a, b, c] como argumentos e depois (d, e, f) será outro argumento, para tanto, escreveríamos o seguinte:

```
teste_X = [((n, [a, b, c]),), (((d, e, f),), ...]
```

- Listas são objetos não-hashable em Python, portanto, caso algum de seus testes contenha listas de alguma forma, elas serão transformadas em tuplas. Veja como o teste do item anterior seria lido:

```
teste_X = [((n, (a, b, c)),), (((d, e, f),), ...]
                # |||||||||
                # | A lista [a, b, c] é considerada como tupla (a, b, c)
```

  Então, suas funções receberiam a tupla de mesmo elementos que a lista inserida, mantenha isso em mente, afinal tuplas possuem propriedades diferentes de listas.
- Os argumentos de teste nem sempre precisam ser do formato citado anteriormente, no entanto, mantê-lo é uma boa prática. Porém, caso você queira passar alguns inteiros ou strings n, m, o como argumentos únicos para um determinado exercício X, as seguintes formas são equivalentes:

```
teste_X = [n, m, o] = [(n,), (m,), (o,)] = [((n,),), ((m,),), ((o,),)]
```

- Caso teste seja uma lista de tuplas da forma:

```
teste_X = [(1, 2, 3), ('a', 'b', 'c')]
```

  Determinada função receberá os inteiros 1, 2 e 3 como argumentos, depois, em outra execução, a, b e c.
- Essas possibilidades trazem mais versatilidade e facilidade na hora de digitar os seus testes, no entanto, é recomendado, fora dos itens citados aqui, seguir o modelo de tupla com até dois elementos iteráveis.

3.3 Configure a Geradora de Gabaritos Aleatórios (Opcional)

Caso queira que os testes do gabarito sejam gerados de maneira aleatória, configure a GBA. Configure as seguintes variáveis no topo do arquivo:

```
# CONFIGURE ================================================================================
n_gab_aleatorios: int = ?                   # quantos gabaritos aleatórios você deseja gerar.
manual: dict[str: tuple] = {}               # coloque o manual das gerações.
# ==========================================================================================
```

Para entender o Manual da Geração, leia: GGA, Geradoras

3.4 Baixe as submissões do Moodle

Baixe as submissões do Moodle da respectiva aula que você configurou acima e coloque o(s) arquivo(s) .zip na mesma pasta que está o arquivo CAPP-L.py. Podemos corrigir quantas turmas você quiser ao mesmo tempo!

3.5 Inicialize o Corretor

Escolha o método de geração de gabarito com a variável:

```
tipo_geracao_gab: str = '' # 'n', para geração normal, 'a' para geração aleatória
```

Quando esta variável é omitida, CAPP-L considerará a geração não aleatória de gabarito. Caso deseja mudar o método de geração de gabarito após um ter sido gerado, deve-se apagar o gabarito gerado, alterar o modelo de geração e executar CAPP-L novamente.

Execute o arquivo CAPP-L.py e veja a magia acontecer!

3.6 Casos de Reconfiguração do Corretor

3.6.1 Houve alguma mudança em funcoes_gabarito.py

Neste caso, é preciso gerar um novo arquivo gabarito.py. Independente do método de geração do(s) gabarito(s), após ter feito as devidas alterações nas funções do gabarito, apague o arquivo gabarito.py em CAPP-I e execute novamente o CAPP-L.

3.6.2 Alteração no modo de geração do(s) gabarito(s)

Quando já existe um gabarito gerado e deseja-se gerar outro, porém de método diferente, é preciso alterar, no CAPP-L a variável <tipo_geracao_gab>, apagar o arquivo gabarito.py em CAPP-I e configurar a geração do novo gabarito como se deseja.

3.6.3 Novo arquivo Zip com novos Notebooks de uma Turma já existente em Alunos

O CAPP-L é capaz apenas de reconhecer e assimilar as turmas dos arquivos zip's àquelas que estão em Alunos, não há comparação de arquivos. Por isso, quando se tem uma nova submissão de uma turma sobre uma mesma aula, apague a pasta correspondente à ela em <Alunos>, ou, se preferir, apague toda a pasta <Alunos>, o corretor recriará as subpastas para as turmas que se deseja corrigir.

4 Corrigindo em Lotes

Aqui, vamos explicar como o multiprocessamento dos scripts funciona e passar explicando o código escrito em CAPP_L.py. A biblioteca principal utilizada para isso é a multiprocessing, esta nos permite usar dos processadores lógicos do computador para realizar várias correções ao mesmo tempo.

4.1 Multiprocessando Scripts

- Para iniciar o multiprocessamento, criamos uma pool de processos
- Depois, fazemos o tratamento dos dados para as funções serem importadas. Para melhor entender o tratamento, é recomendável que seja lido Módulos. Aqui, ele não será explicado devido à extensão do código. No entanto, o leitor é mais que convidado à olhar o código escrito.
- Iteramos pelas turmas, depois pelos seus scripts, e então pelo gabarito, usando apply_async em executa_scripts_1_teste_por_exercicio para aplicar os processos dos exercícios com seus testes na pool.
- Usamos .get() com timeout em 0.5 segundos para receber os resultados da função.
- Usando estes resultados, montamos um dicionário de estrutura igual à gabarito para cada aluno, chamando a função compara_respostas() para gerar o arquivo de correção do aluno. Este arquivo vai para a pasta Alunos/TX/Correção e lá o aluno pode ver para quais argumentos suas funções apresentaram erro ou acertaram, além de saber sua nota.
- Após toda uma turma é corrigida, é criada em sua pasta Alunos/TX uma tabela contendo o nome dos alunos, suas notas e seus acertos, para que o professor rapidamente consiga visualizar a nota de seus alunos e lançá-las no sistema.
- Após as iterações finalizarem, acaba a correção e a pool é fechada.
- Caso a correção não foi possível de ser realizada, será citado no terminal o motivo.

4.2 Parando Processos em Loop

Estamos corrigindo os exercícios pois as pessoas não são perfeitas, cometem erros. Os alunos não estão fora desta regra, no entanto, existe um erro fatal: o loop. É comum exercícios que apresentam recursão apresentarem TimeoutError devido à uma falta de cláusula base de recursão. Isso, sem o devido cuidado, pode fazer com que o corretor entre em Loop, pois aguardaria eternamente por uma resposta do script. Para evitar isso, foi feito o seguinte:

- Esperamos, no máximo, meio segundo para que o script do aluno apresente algum resultado.
- Caso .get() provoque um erro de timeout, vamos nos estados do processo e pegamos seu evento.
- Quando um processo é lançado para a pool, ele é executado dentro do cache da pool, no entanto. Quando este processo é finalizado ele sai do cache, o mesmo não pode ser dito para processos não finalizados - eles continuam lá. O mesmo ocorre para os exercícios que derem Loop! Eles ficam como 'unset'. Ao pegar seu ThreadingEvent(), podemos usar .set() para que seu estado fique como 'set'! Assim, a pool deixa de aguardar por uma resposta e o processo é liberado para executar outro exercício.
- Quando ocorre o erro de Timeout, o aluno recebe o erro de Timeout como saída do Script.

  Dê uma olhada no código:

```
try:
    resultado = processo.get(timeout=0.5)        # esperamos, no máximo meio segundo;
except TimeoutError as e:                        # caso dê erro de Timeout:
    resultado = (e, [])                          # o aluno fica com o erro;
    event = processo.__getstate__()['_event']    # pegamos o evento;
    event.set()                                  # o processo para.
```

É importante dizer que .get() retorna os resultados conforme eles chegam, porém, quando timeout é especificado, há uma espera de no máximo <timeout> segundos até que o resultado seja retornado. Portanto, para cada teste que entra em loop, o tempo de correção cresce em $0.5 \times n_{loops}$. No entanto, caso o nível das funções requeridas sejam de maior nível computacional ou de maior lentidão, esse tempo é completamente alterável - ajuste-o de modo que se encaixe melhor com suas aulas. Para alterar o tempo de timeout, basta procurar o trecho de código anterior em CAPP_L.py e, em timeout=, colocar depois do sinal de igual o tempo, em segundos, que você gostaria que o corretor esperasse até considerar Erro por Timeout.

4.3 Resultado da Correção

Os arquivos de texto gerados pela correção são lançados para cada turma dentro da pasta Correção de cada turma, lá estão as correções individuais de cada aluno, onde eles podem ver seus erros e acertos. Para o professor, dentro da pasta de cada turma, é criado um arquivo Notas_dos_Alunos, que contém uma tabela com os nomes, notas e acertos de cada aluno. Além disso, dentro do módulo Estatísticas, é criada uma pasta da turma que foi corrigida, caso ela não exista. Dentro desta pasta, é criada uma subpasta chamada CSVs, lá, ficam as tabelas de notas de todas as aulas corrigidas de determinada turma.

4.4 Recorrigindo uma Turma

Se, por quaisquer motivos, for preciso recorrigir a submissão de uma turma, é bastante simples: basta executar novamente o corretor. Os arquivos de texto já criados serão atualizados com os novos resultados.
Vale ressaltar que, ao recorrigir uma turma, o arquivo CSV, em Estatísticas/<turma>, de tal aula será atualizado e terá os nomes dos alunos que participaram desta correção, qualquer aluno que participou na correção anterior, mas não nesta, terá seus dados, desta aula, perdidos. Ao se corrigir novamente uma aula, não é preciso apagar nenhum arquivo em Estatísticas.
No entanto, alguns motivos requerem reconfigurar o corretor, leia Casos de Reconfiguração do Corretor para mais informações.

5 Geração de Análises Estatísticas e Criação de Bancos de Dados sobre seus Alunos

Quando uma aula é corrigida, é no módulo Estatísticas que alguns dos resultados da correção vêm parar. Conforme mais aulas são corrigidas mais dados sobre os seus alunos são adquiridos e mais útil se torna este módulo. Desevolvido para lidar com as notas de seus alunos e gerar gráficos mostrando os rendimentos individuais e gerais, esta seccção é dedicada às coleta estatística e geração de dados do corretor.

5.1 Executando o Módulo

Para gerar as análises e planilhas é bastante simples: em CAPP-L/Estatísticas execute o arquivo Analizadora.py e, simples assim, todas as pastas serão criadas baseadas nas turmas que já foram corrigidas e possuem os seus arquivos CSVs. Para entender o que será criado e como veja o algoritmo da Analizadora

5.2 Instruções de Uso

Este módulo é voltado para quando se possui um de dois casos:

- Algumas aulas já foram corrigidas e deseja-se obter as planilhas dos alunos e turmas e seus gráficos.
- Todas as aulas já foram realizadas e corrigidas e deseja-se obter suas planilhas, gráficos e notas finais.

Isso porque a geração de gráficos diz pouco quando seu volume de aulas é menor, assim como as notas finais e médias ainda terão grande alteração. No entanto, não há nada que lhe impessa de executar a Analizadora, pelo contrário, aproveite ao máximo o que o corretor lhe disponibiliza, leitor(a).

Caso você já tenha executado a Analizadora.py antes e deseja executá-la novamente, há algumas coisas que devem ser mantidas em mente:

- As planilhas CSV dos seus alunos e turmas são atualizadas com base nos arquivos presentes na pasta CSVs de cada turma. Quando uma correção é feita, caso seu arquivo CSV não exista na pasta de determinada turma, ele será criado, caso contrário, ele será atualizado. Tenha em mente que a atualização destes dados não mantém os dados anteriores, logo, caso um aluno que participou da primeira correção não participe desta, seus dados serão perdidos.
- A execução da analizadora gira em dos arquivos CSVs de cada turma nas suas respectivas pasta CSVs. Não há problema adicionar arquivos novos de maneira externa ou modificar os que já existem, no entanto, deve-se manter o formato e o padrão dos nomes. É recomendado utilizar o formato: _aula_X como nome de suas aulas. A analizadora é apenas uma leitora e manipuladora de arquivos.
- Os gráficos gerados são arquivos png e têm a data geração marcada no nome do arquivo. Gráficos gerados no mesmo dia são sempre atualizados para a última execução.
- Caso você queira apagar tudo que já foi gerado, para uma determinada turma, pela execução da Analizadora, a única pasta que não pode ser apagada é <turma>/CSVs.

5.3 Utilize as tabelas e dados de maneira esperta!

A SER ESCRITO

6 O Corretor Individual (CAPP-I)

Contrário, porém aliado, ao Corretor em Lotes, o Corretor Individual busca ser uma ferramenta rápida e para a execução de testes em Scripts Python, porém de maneira individual. Sua usabilidade foi pensada voltada para o aluno, para que ele pudesse executar e testar suas funções que serão submetidas em aula ou até mesmo avaliar suas funções e ver suas notas, caso um gabarito seja disponível.

6.1 Algoritmo do Corretor Individual

O corretor usa da função que executa scripts, contida em ICA, porém aqui está sob uma pasta de novo nome: FCA: funções de correção automática. O Script que se deseja aplicar os testes é importado, então, usando da configuração do CAPP-I, são importadas as funções que se deseja testar e os testes especificados são aplicados.

6.2 Configurando o Corretor

- Na veriável config, a qual é um dicionário, as chaves devem ser strings cujos nomes são as funções, presentes no arquivo .py que você deseja testar.
- Os valores de cada chave devem ser iteráveis: listas ou tuplas, cada elemento do seu iterável é teste que será passado para a função designada pela sua chave.
- O formato dos testes é idêntico ao formato de como em configurar testes do CAPP-L. Deve-se usar uma tupla de no máximo 2 iteráveis, no entanto, os adendos também são válidos.

6.3 Executando e Corrigindo suas funções

Após definir a variável config com as funções que quer corrigir e os testes que deseja aplicar em cada uma, coloque o arquivo .py na mesma pasta que o CAPP-I está e depois execute o CAPP-I. Será gerada um arquivo chamado Resultado.txt, contendo um relatório das entradas e saídas dos testes realizados. **Caso haja algum outro arquivo estranha, além do seu script .py, na pasta do CAPP-I, a correção pode não ser feita, portanto, deixe apenas os arquivos originais e o script python que você deseja testar.**
Caso seja disponibilizado o gabarito de alguma aula, é possível usar o corretor individual para corrigir o script desejado com base neste gabarito. Para que isso aconteça, coloque o arquivo gabarito.py na pasta do CAPP-I e execute o corretor individual. Isso gerará o mesmo arquivo de correção, desta vez contendo as notas do aluno.

7 Análises de Bibliotecas de Processos

Durante o desenvolvimento do corretor, foram testadas 4: Concurrent.Futures, Pebble, Subprocess e Multiprocessing, essa útima se tornou a definitiva para a execução do corretor. Aqui, faremos umas análises das tentativas.

7.1 Concurrent.Futures

Tratando-se de uma biblioteca extremamente otimizada para lidar com processos, encontramos um problema: matar ou parar os processos em Loop. Quando, em um dos processos da Pool é lançada uma tarefa, é gerado um processo filho e este é que gostaríamos de encerrar a execução - no entanto, ao tentar matá-lo usando seu PID obtém-se BrokenPoolError() e todo o corretor para de ser executado. Dessa forma, precisaríamos de outra biblioteca para lidar com as possibilidades de Loop. No entanto, teoricamente, qualquer script submetido tem risco de sua execução ter tempo indeterminado, assim, a melhor decisão foi não usar Concurrent.Futures.

7.2 Pebble

A biblioteca Pebble trata-se de uma melhoria, por assim dizer, da biblioteca anterior. Afinal, nesta, é possível matar processos conforme for preciso. Porém, por se tratar de uma biblioteca não muito usual, foi temido que ela, eventualmente, parasse de receber atualizações de compatibilidade, atrapalhando assim a execução do corretor. Portanto, havíamos duas opções: procurar uma biblioteca *"mainstream"* que, certamente, teria menos chances de parar de receber atualizações, ou tentar a sorte e correr o risco de, eventualmente, ter de procurar outra biblioteca. Bom, o escolhido foi a primeira opção, afinal, escolhendo a segunda, a primeira continuaria nos planos, logo, sobraram duas opções: Subprocess e Multiprocessing.

7.3 Subprocess

A biblioteca Subprocess permitiu com que construíssemos os processos à mão, conectando Pipes aos seus stdin, stdout e stderr, permitindo um total controle sobre o processo que estava sendo executado. No entanto, como, por padrão, a bibliteca executa comandos por meio do Prompt, precisávamos construir um Mini Script que rodava scripts.

```
process = Popen(args=["python", "-c",
                  f"import sys; import {nome_script.replace('.py', '')}\n"
                  f"resultado_teste = ({nome_script}.{nome_exercicio}(*{teste}),)\n"
                  "print('__RESULTADO__: ', resultado_teste)"],
             stdin=PIPE, stdout=PIPE, stderr=PIPE, encoding='utf-8', text=True)

for input in u_inputs:
    process.stdin.write(str(u_input) + '\n')
    process.stdin.flush()

try:
    output, error = process.communicate(timeout=2.0)

except:
    process.kill()
    error = sys.exc_info()
    output = ''

if '__RESULTADO__:' in output:
    indice_resultado = output.find('__RESULTADO__:') + 15
    # 15 pois é o findable + caracter de espaço
    output = output[indice_resultado:]
```

Esse código, permite com que executemos qualquer script. Seu algoritmo é o seguinte:

- Primeiro, simulando um prompt de comando, importamos o script do aluno e sua função. Pegamos sua função e lançamos o teste, depois, imprimos o resultado.
- Ao imprimir o resultado, fazemos com que, antes dele, haja uma sequência de caracteres específica, que será utilizada para encontrarmos o resultado da função. Neste caso, o utilizado foi ___RESULTADO___.
- Nos comunicamos com o processo para pegar seu resultado e seu erro, caso dê timeout, matamos ele diretamente.

- Então, se for possível encontrar a sequência de caracteres especificada, consideraremos tudo que vier depois dela como resposta do programa.

Uma melhor e mais profunda explicação dessa solução pode ser encontrada no meu Github por meio das postagens:

- Multiprocessing Sum With User Input - Aqui, utilizamos Multiprocessing.Pool munido de subprocess.Popen para multiprocessar a função soma com input de usuário.
- Multiprocessing Student Functions with Input for Correction - Aqui ocorrem testes análogos, porém levemente mais avançados.

Essa solução parecia promissora, afinal, caso o processo entrasse em Loop e .communicate() gerasse erro de timeout, poderíamos facilmente terminar o processo, porém provou-se demasiado lenta, devido à quantidade de subprocessos que seriam abertos, afinal, é um novo script importando e executando um script que já existe. Além disso, gostaríamos de utilizar a biblioteca unittest.Mock, pois ela permite que facilmente peguemos os prints do programa. Isso é útil para exercícios que é requerida a impressão de algo na tela. Decidimos então, executar testes com a biblioteca que havia sobrado: multiprocessing.

7.4 Multiprocessing

Multiprocessing provou-se melhor devido, unicamente, à sua capacidade de parar o processo por meio de ThreadingEvent().set() em seu estado. Isso permitiu com que pudéssemos executar os scripts dos alunos para quaisquer argumentos, sem que o corretor parasse de funcionar e o script que entra em loop tivesse de ser retirado da correção. Assim, a nota recebida pelo teste que entrou em loop é 0; além disso, não perturbamos os processos originais da Pool.

8 Modelos Alternativos de Correção

Nesta secção está a explicação do método de correção e sugestões de implementações para outros tipos de correção. Fica como exercício para o leitor, caso queira ou ache necessário, implementar quaisquer mudanças no código original.

8.1 Modelo Original de Correção

O modelo de correção é bastante maniqueísta: ou está certo ou está errado. A comparação das respostas é feita de modo que só é contabilizado em dois casos:

- Se há input(s), então tanto o output deve ser igual em forma e tipo ao output de gabarito, assim como a lista de prints deve ser a mesma.
- Se não há input, então o output deve ser igual em forma e tipo ao output de gabarito.

De fato, pode-se debater sobre a existência de meio-acerto, algo que aqui não é contabilizado. Portanto, seguem algumas sugestões de implementação.

8.2 Sugestões de Implementação

8.2.1 Nota proporcional à semelhança

Neste caso, deve-se pensar em um peso que divida a nota do output e dos prints (caso seja contabilizado). Suponha que o peso $0 < \rho < 1$ sobre a nota do resultado (output), e $1 - \rho$ seja o peso sobre o que foi emitido na tela (prints). Portanto, tem-se:

- que a nota sobe os outputs é calculada por:

$$nota_{exercicio} \times \rho \times \frac{n\_acertos_{output}}{n\_total_{output}} = nota_{output}$$

- e que a nota sobre os prints é calculada por:

$$nota_{exercicio} \times (1 - \rho) \times \frac{n\_acertos_{prints}}{n\_total_{prints}} = nota_{prints}$$

- A nota final do exercício então seria:

$$nota_{final} = nota_{output} + nota_{prints}$$

$$nota_{final} = \left(nota_{exercicio} \times \rho \times \frac{n\_acertos_{output}}{n\_total_{output}}\right) + \left(nota_{exercicio} \times (1 - \rho) \times \frac{n\_acertos_{prints}}{n\_total_{prints}}\right)$$

$$nota_{final} = nota_{exercicio} \left(\frac{n\_acertos_{output}}{\rho \times n\_total_{output}} + \frac{n\_acertos_{prints}}{(1 - \rho) \times n\_total_{prints}}\right)$$

Suponha que o exercício vale 4 pontos os pesos sejam de 50%, o número total de acertos sejam de 5 e 10 para output e print respectivamente. Portanto, para cada elemento do output do aluno que é igual ao output do gabarito, contabiliza-se:

$$\frac{2 \times n\_acertos_{output}}{5} = nota_{output}$$

Analogamente, para os prints, obter-se-ia:

$$\frac{n\_acertos_{prints}}{5} = nota_{prints}$$

A nota final do exercício então seria dada pela equação:

$$nota_{final} = nota_{output} + nota_{prints}$$

$$nota_{final} = \frac{2(n\_acertos_{output} + 2 \times n\_acertos_{prints})}{5}$$

8.2.2 Margens de Erro e Intervalos de Tolerância

Similarmente, é possível propor, para exercícios mais avançados, uma margem de erro aceitável, onde, se a resposta do aluno estiver dentro dessa margem de erro, é obtido um acerto total. No entanto, é bastante interessante pensar um intervalo mínimo de aceitação e um máximo de tolerância, no qual, quanto mais próxima a resposta do aluno está do intervalo mínimo, sua nota é inversamente proporcional à distância. Por exemplo:

Seja r o valor númerico da resposta esperada e tome $\delta > \epsilon > 0$ Considere $I = [r - \epsilon, r + \epsilon]$ o intervalo mínimo onde o aluno obterá nota total n e $J = (r - \delta, r + \delta)$ o intervalo máximo onde a nota será proporcional à distância.

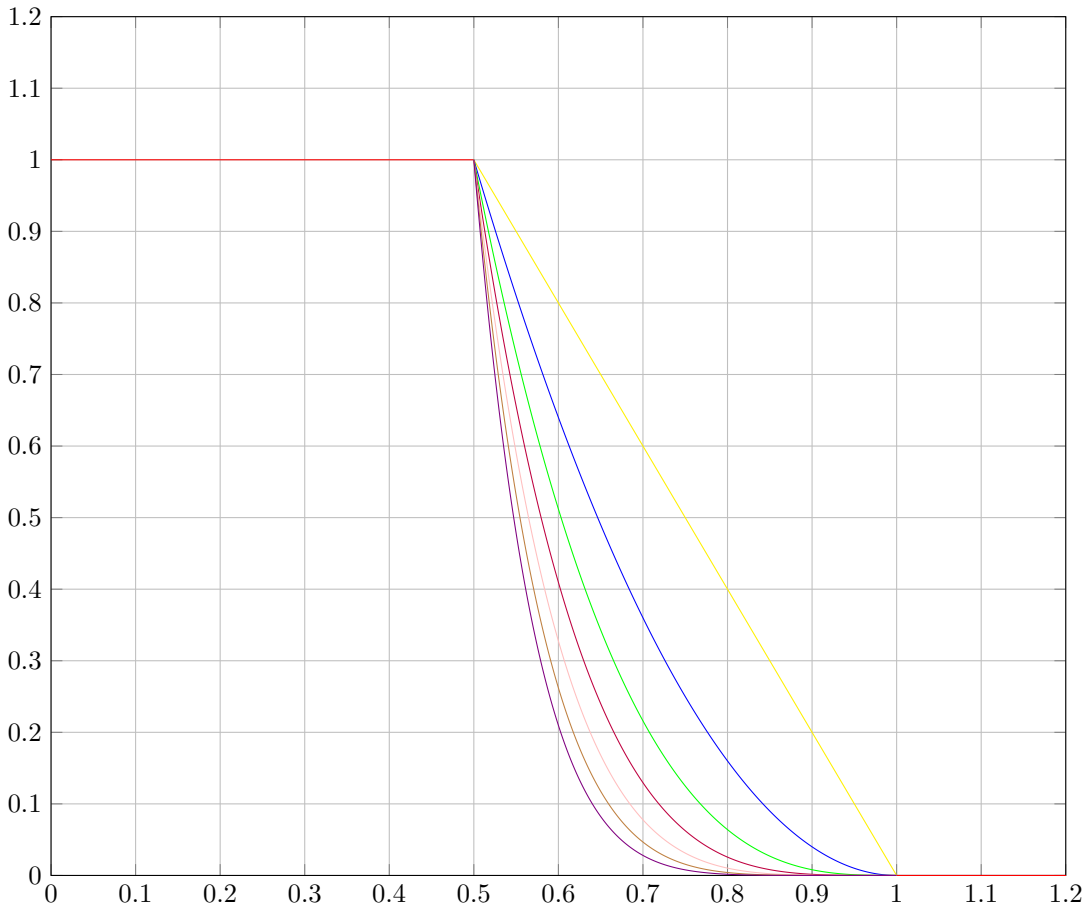
$$f_n(x) = \begin{cases} 1 & \text{se } d(r, I) \leq \epsilon \\ (2\frac{nota_{exercício}}{\delta - \epsilon} - 2\frac{nota_{exercício}}{\epsilon - \delta}x)^n & \text{se } \epsilon < d(r, I) < \delta \\ 0 & \text{se } d(r, J) \geq 0 \end{cases}$$

sendo n o quão rápido a nota deveria decrescer no intervalo máximo.

Para poupar esforços, vamos considerar que a nota do exercício é 1, $\epsilon = 0.5$ e $\delta = 1$: Nossa $f_n(x)$, então, seria:

$$f_n(x) = \begin{cases} 1 & \text{se } d(r, I) \leq \epsilon \\ (2 - 2x)^n & \text{se } \epsilon < d(r, I) < \delta \\ 0 & \text{se } d(r, J) \geq 0 \end{cases}$$

Observe o gráfico com $f_1(x), f_2(x), f_3(x), f_4(x), f_5(x) f_6(x), f_7(x)$



Vale ressaltar que, conforme n cresce, $f_n(x)$ fica cada vez mais assintótica e, quando $n \longrightarrow \infty, f_n(x) \longrightarrow H(s)$ (função de Heavyside) ponto-a-ponto, com:

$$H(s) = \begin{cases} 1 & \text{se } x \leq 0.5 \\ 0 & \text{se } x > 0.5 \end{cases}$$

Ou seja, tome cuidado, pois temos uma sequência de funções contínuas convergindo para uma função descontínua. (Isso ocorre pois a convergência não é uniforme.) Portanto, quanto maior for o peso da distância, menos uniforme ele se torna, logo, busque não deixar n extremamente grande.