

UNIVERSIDADE FEDERAL DO CARIRI

CENTRO DE CIÊNCIAS E TECNOLOGIA

CIÊNCIA DA COMPUTAÇÃO

CÍCERO IGOR ALVES TORQUATO DOS SANTOS

LABORATÓRIO DE PROGRAMAÇÃO

Juazeiro do Norte - Ceará

2022

***ORGANIZAÇÃO:***

* **MUDANÇAS ESTRUTURAIS**
* **TESTES**
* **GITHUB**

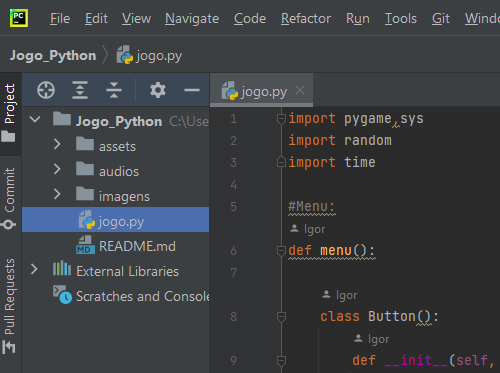
***MUDANÇAS ESTRUTURAIS:***

**INTRODUÇÃO**

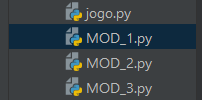
* Na primeira parte do trabalho foi desenvolvido um jogo na linguagem de programação Python e utilizando a biblioteca Pygame. Nesse sentido, agora nosso foco será em melhorar o código desse jogo.
* Executaremos alguns passos que serão importantes para aprimorar cada vez mais o código fonte.

**PRIMEIRA ALTERAÇÃO**

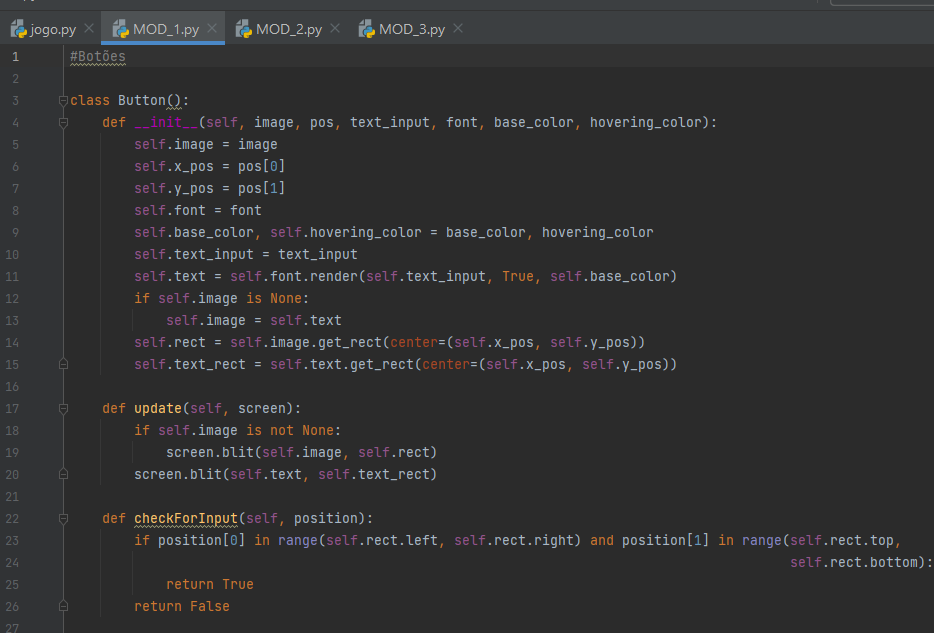
* Tendo em vista todos os conhecimentos adquiridos nas aulas sobre “Modularização” e “Como tornar seu código mais modular?”, podemos aplicar a primeira alteração no nosso projeto.
* Lembrando que o jogo foi desenvolvido em um só arquivo:



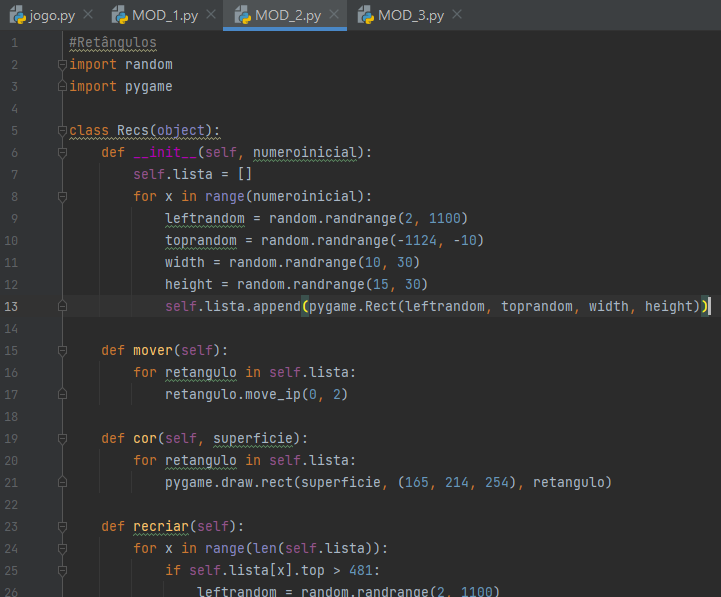
* No código, foram utilizadas classes para os funcionamentos dos botões do jogo, da animação e da criação e movimentação do jogador durante a partida.
* Portanto, nossa primeira alteração consistirá em modularizar o nosso projeto. Dessa forma, reduziremos os números de linhas de código do arquivo principal “jogo.py” e deixaremos o projeto muito mais organizado.
* Módulos:



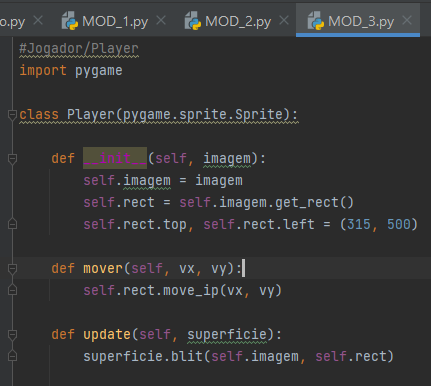
* O MOD\_1 lidará apenas com a classe dos botões.



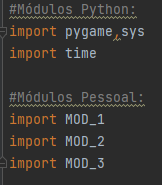
* O MOD\_2 lidará com a classe das superfícies (Retângulos) que o jogador deve desviar no jogo.



* O MOD\_3 lidará com a classe do player (Jogador) e será importante tanto no funcionamento do jogo quanto na animação na tela de menu.

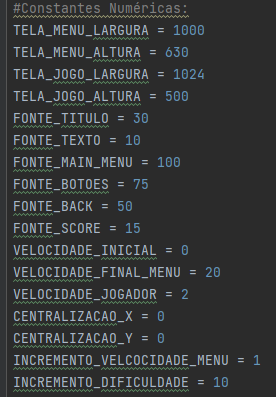


* Finalmente, usaremos todos os módulos no arquivo principal, para isso importaremos cada um deles. Lembrando que, para utilizar o módulo adequadamente devemos digitar seu nome antes de chamar uma função ou classe presente nele.

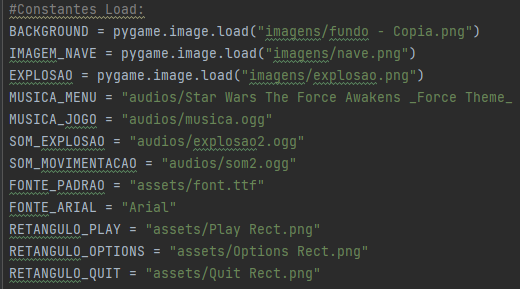
 

**SEGUNDA ALTERAÇÃO**

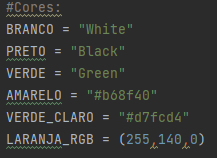
* Além da primeira alteração, o estado final do código do projeto contava com uma série de “números mágicos”, que em programação é o nome dado aos números que aparecem no código sem explicação. Assim, além desses números, outros eventos de cores, textos e carregamento de arquivos estavam no código e isso gera um problema.
* Posteriormente, para fazer alterações nesse código, ou simplesmente para entendê-lo, seria uma tarefa muito complicada se o programador não soubesse o que cada elemento significa. Existe uma maneira de resolver esse problema sem precisar comentar em cada linha e deixar o código mais extenso, que é com a criação de “constantes”.
* Em Python a regra de nomeação das constantes segue um padrão parecido com as de variáveis, com a diferença de que todas as letras são maiúsculas e separadas por underline “\_”.
* Porém, o Python possui tipagem dinâmica e fraca, o que permite que uma variável possa armazenar dados de diferentes tipos em fases diferentes do script. Assim uma constante, em Python, não é bem uma constante, porque pode ser alterada. Mas para resolver o problema no nosso código esse recurso será importante e útil.
* Exemplos das constantes numéricas:



* Exemplos das constantes de carregamento:

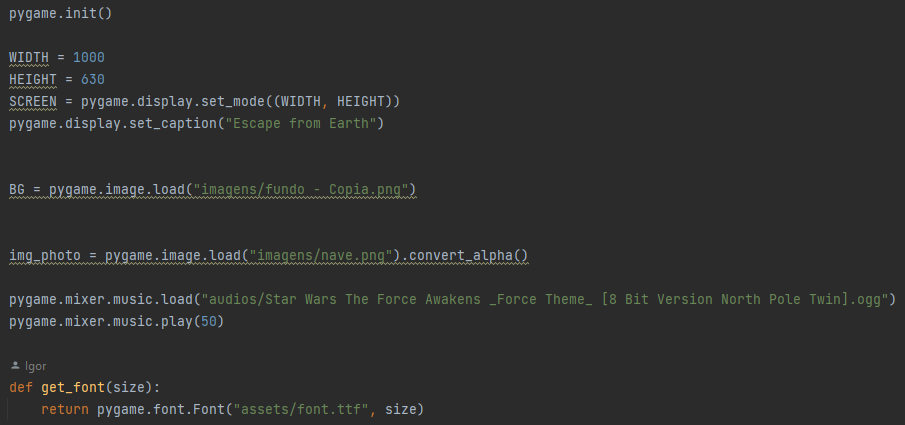


* Exemplos das constantes de cores:

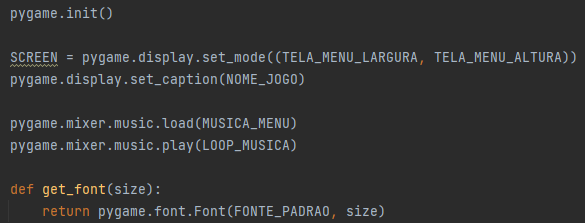


* Diferença do código antes das mudanças e depois das mudanças:

1. Antes:



1. Depois:

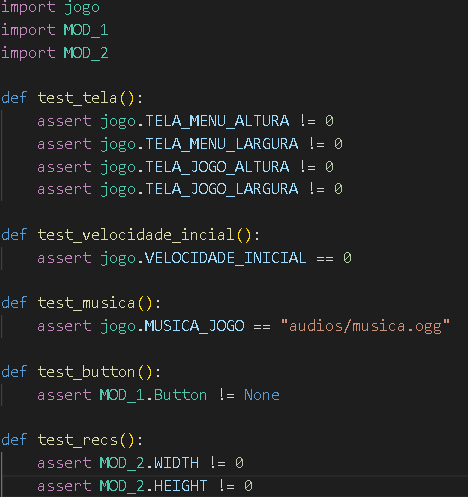


* Enfim, é perceptível que houve uma redução significativa no código após as alterações e para fazer uma mudança em vários elementos o programador só precisa alterar uma variável que está no início do arquivo principal “jogo.py”. Além disso, com a nomeação de cada constante, o entendimento do código melhorou bastante e o código ficou mais simples para possíveis mudanças posteriores.

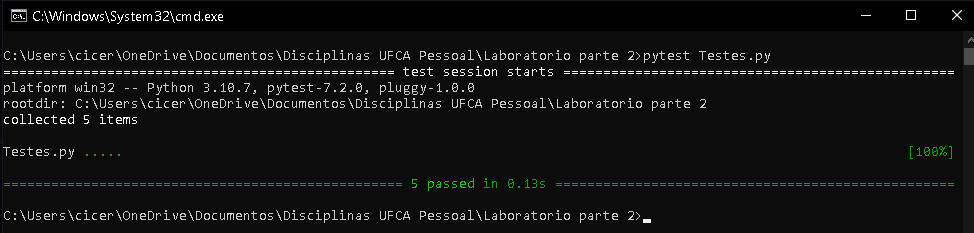
***TESTES:***

**CAIXA PRETA (5) :**

* Iniciando a testagem do nosso código, o *pytest* é um recurso que nos ajudará a fazer os testes iniciais no nosso projeto.
* No projeto, criamos um arquivo chamado “Testes.py”. É nele que criaremos funções que irão checar alguns pontos cruciais para o funcionamento do jogo.
* Lembrando que, para o jogo funcionar corretamente são necessárias uma série de elementos definidas de forma coesa e correta. Então os testes serão utilizados para verificar a integridade de pontos específicos do código que precisam, além de estarem certos, seguir um certo padrão.
* Para os cinco primeiros testes decidi explorar esses pontos importantes: A tela, a velocidade da nave, a música, os botões e as superfícies (Retângulos).
* Todos esses cinco elementos não podem ter nenhum defeito para que o jogo entregue algo mínimo ao usuário. As dimensões da tela devem ser diferentes de zero; A velocidade inicial da nave deve ser nula inicialmente para que o usuário possa alterá-la e para que a animação funcione; A música deve ser reproduzida; Os botões devem estar em pleno funcionamento para que o usuário possa clicar sobre ele.
* Arquivo:



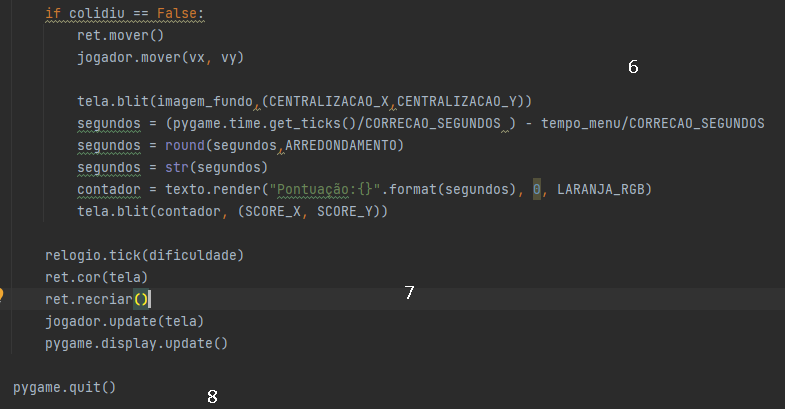
* Executando os testes:



**CAIXA BRANCA (8):**

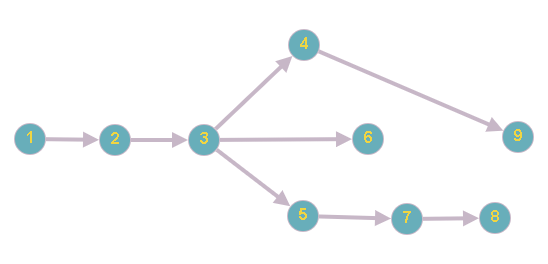
* O teste da caixa branca, se preocupa com a estrutura interna do software, assim precisaremos analisar o código fonte. Com o código, é importante fazer a testagem de todos os caminhos possíveis que o programa pode rodar. Tendo em vista essas observações, dividi o código em alguns blocos para que os testes possam ser realizados.
* Na função *main()* do jogo , podemos realizar uma divisão em 9 blocos:

1. Inicialização do *pygame* e variáveis.
2. Condicional *while sair != True.*
3. Evento do *pygame* usando o laço de repetição *for.*
4. Condicional *pygame.event.get()* que checa se o usuário fechou o programa.
5. Condicional *colidiu == False* que checa os eventos de teclado.
6. Condicional *colisão* que checa se houve colisão.
7. Condicional *colidiu == False* que realiza a movimentação do jogador, dos retângulos e apresenta a pontuação.
8. Comandos que realizam a recriação dos retângulos, aumento da dificuldade do jogo, apresentação do jogador e atualização de frames do display.
9. Evento de saída *pygame.quit().*
10. Evento do *pygame* usando o laço de repetição *for.*



Exemplo dos Blocos (6,7,8)

* Com os blocos devidamente enumerados, podemos montar um grafo que representará o fluxo que o algoritmo segue. Dessa forma, saberemos a quantidade de testes que será preciso ser executado para que o jogo funcione corretamente. Utilizarei o método de cobertura de instruções para saber se todas elas foram executadas.

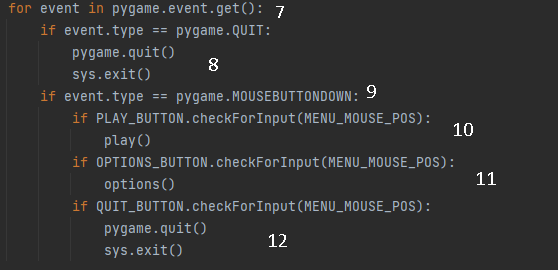
**

* Testes:

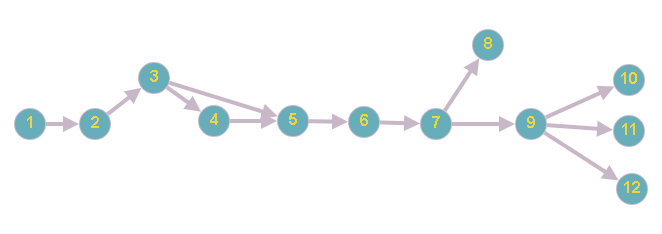
| **Análise** | **Nós Percorridos** |
| --- | --- |
| sair == True (No bloco 3) | *{1,2,3,4,9}* |
| sair != True e colidiu == False (Condição normal de jogo) | *{1,2,3,6}* |
| sair !=True e colidiu == True (Condição de derrota) | *{1,2,3,5,7,8}* |

* Podemos concluir que todos os fluxos de caminho de funcionamento do jogo estão corretos e funcionando à medida que o jogador desvia dos obstáculos ou opta por clicar em *quit*.
* Agora vamos fazer o mesmo procedimento para o menu. Na função *main\_menu()* vamos dividir o código em blocos e enumerá-los para posteriormente criarmos um grafo com os caminhos possíveis e testar cada um deles.

1. Definição das variáveis que serão utilizadas na animação.
2. Condicional *while True.*
3. Incremento das variáveis de velocidade da animação e apresentação do background.
4. Condicional *YVEL == VELOCIDADE\_FINAL\_MENU* que limita a velocidade das naves da animação.
5. Definição das variáveis dos botões e apresentação da animação.
6. Evento de mudança de cor do botão ao ser sobreposto pelo mouse com laço *for.*
7. Evento do *pygame* com laço *for.*
8. Condicional de *quit* ao clicar no “*x”* do canto superior da janela.
9. Condicional de mouse do *pygame* que checa se o usuário clicou em algum botão.
10. Condicional que checa se o usuário clicou no botão que executa *play().*
11. Condicional que checa se o usuário clicou no botão que executa *options().*
12. Condicional que checa se o usuário clicou no botão que executa *quit().*

**

Exemplo dos Blocos (7,8,9,10,11,12)



* Testes:

| **Análise** | **Nós Percorridos** |
| --- | --- |
| event.type == pygame.QUIT (Quando for ativo esse evento no bloco 8) | *{1,2,3,4,5,6,7,8}* |
| PLAY\_BUTTON.checkForInput(MENU\_MOUSE\_POS) (Ao clicar em jogar) | *{1,2,3,4,5,6,7,9,10}* |
| OPTIONS\_BUTTON.checkForInput(MENU\_MOUSE\_POS) (Ao clicar em manual) | *{1,2,3,4,5,6,7,9,11}* |
| QUIT\_BUTTON.checkForInput(MENU\_MOUSE\_POS) (Ao clicar em sair) | *{1,2,3,4,5,6,7,9,12}* |
| if YVEL != 20 and XVEL != 20 (Teste de clicar rápido nos botões sem esperar a animação concluir) | *{1,2,3,5,6,7,8}* |

* Pode-se concluir que o nó 4 aparece em todos os caminhos mesmo que seu nó anterior, 3 , possa ser direcionado ao vértice 5 e isso faz sentido devido a limitação da posição imposta às naves da animação. Nesse sentido, já que elas se movimentam até a condicional imposta pelo bloco 4 essa condição sempre será checada para que ocorra esse controle da animação. Somente em um caso muito específico ela não será checada, que é quando o usuário não espera pela animação do menu e clica em algum dos botões.

***GITHUB:***

* A nova versão do sistema com as mudanças e as suítes de teste estão no repositório do seguinte link: https://github.com/IgorTorquatto/Jogo\_Python