Aula 3: Oficina de Programação e Robótica

- Revisão da linguagem do Reeborg
  - Comandos nativos:

    move()

    turn_left()

    pick_beeper()

    put_beeper()

    turn_off()

  - Comandos condicionais:

    if *condição****:***

    comandos a serem executados se a condição for verdadeira

    if *condição****:***

    comandos a serem executados se a condição for verdadeira

    else:

    comandos a serem executados se a condição for falsa

  - Comandos iterativos (ou laços de repetição):

    for i in range(*vezes****):***

    instruções a serem repetidas

    while *condição****:***

    instruções a serem repetidas

  - Definição de métodos:

    def *nome****():***

    instruções que implementam a operação desejada

  - Condições no mundo do Reeborg (funções de teste):

Teste	Teste ao contrário	O que ele verifica:
front_is_clear()	not front_is_clear()	Existe uma parede em frente a Reeborg?
left_is_clear()	not left_is_clear()	Existe uma parede à esquerda de Reeborg?
right_is_clear()	not right_is_clear()	Existe uma parede à direita de Reeborg?
facing_north()	not facing_north()	Reeborg está virado para o Norte?
carries_beepers()	not carries_beepers()	Reeborg está carregando *beepers*?
on_beeper()	not on_beeper()	Reeborg está sobre um *beeper*?

- Exercício para relembrar a aula anterior: Vocês se lembram de como fazer o Reeborg pegar um beeper sem dar erro? Vamos usar essa estratégia nesse exercício. Espalhem alguns beepers pelo mundo e deem alguns beepers para o Reeborg também, se quiserem (13º botão do menu). Vocês devem programar o Reeborg da seguinte forma... Se houver beepers na posição atual do Reeborg, ele pega um beeper; caso contrário, ele coloca um beeper nessa mesma posição (desde que o Reeborg carregue beepers na sua cesta). Neste exercício vocês devem usar o comando ***if-else***!

Aviso de spoiler: esta seção contém revelações sobre o enredo! :D

A explicação a seguir mostra passo-a-passo uma maneira simples de resolver esse problema, com o Reeborg dando uma volta em torno de seu mundo. Vamos começar imaginando que o Reeborg esteja na esquina da rua 1 com a avenida 1, como de costume. Tente não pensar em que lugares você dispôs os beepers, ou em quantos beepers você deu previamente ao robô.

Qual é a 1ª condição que precisamos testar? Se o Reeborg está sobre um beeper. E, se estiver, o que ele deve fazer? Pegar um beeper.

```
1 if on_beeper():
2     pick_beeper()
```

E se o Reeborg não estiver sobre um beeper, o que ele deve fazer? Colocar um beeper naquele lugar. Como você faria isso? Usando outro comando ***if***?

```
1 if on_beeper():
2     pick_beeper()
3 if not on_beeper():
4     put_beeper()
```

Nesse caso, isso não é o correto; precisamos usar um ***else***. O motivo é bem simples, na verdade. Conforme vimos no início da aula, o bloco ***if-else*** é usado quando queremos escolher entre duas opções. Ou seja, se a condição testada no ***if*** for verdadeira, o programa executará somente os comandos dentro dele, e pulará o que estiver dentro do ***else***. Caso contrário, se a condição do ***if*** for falsa, será executado somente o que corresponder ao ***else***. Se usássemos outro ***if***, como no exemplo acima, os dois testes seriam realizados. Imagine que o Reeborg esteja sobre 1 único beeper. De acordo com o 1º teste, ele vai pegar esse beeper. Por isso, essa posição ficará sem beepers. Assim, continuando a execução do programa, o 2º teste também será verdadeiro, e o Reeborg

colocará um beeper de volta. Ou seja, se ambos os testes forem verdadeiros, o Reeborg fará as 2 coisas no mesmo lugar, e não é isso o que queremos! A forma correta é a seguinte, e observe que o ***else*** fica fora do ***if*** (estão na mesma direção):

```
if on_beeper():
    pick_beeper()
else:
    put_beeper()
```

Agora que já testamos a existência ou não de beepers, o que o robô deve fazer em seguida? Não resta mais nada além de andar. E esse comando deve ficar onde, dentro ou fora do ***else***? Fora dele, pois queremos que o robô ande independentemente de haver ou não um beeper na sua posição atual.

```
if on_beeper():
    pick_beeper()
else:
    put_beeper()
move()
```

Na posição seguinte, o Reeborg deve verificar novamente a existência de beepers e andar mais uma posição, e assim por diante. Isto é, ele vai executar esse trecho de código diversas vezes, você percebe? Para tanto, que comando devemos usar? Um ***while***; por quê? Você notou que a última ação do robô em cada iteração será o ***move()***? Isso não parece familiar, como quando queríamos que o robô desse a volta num mundo de tamanho qualquer? Exatamente! Ele vai executar os testes quanto aos beepers e andar enquanto não encontrar uma parede.

```
while front_is_clear():
    if on_beeper():
        pick_beeper()
    else:
        put_beeper()
    move()
```

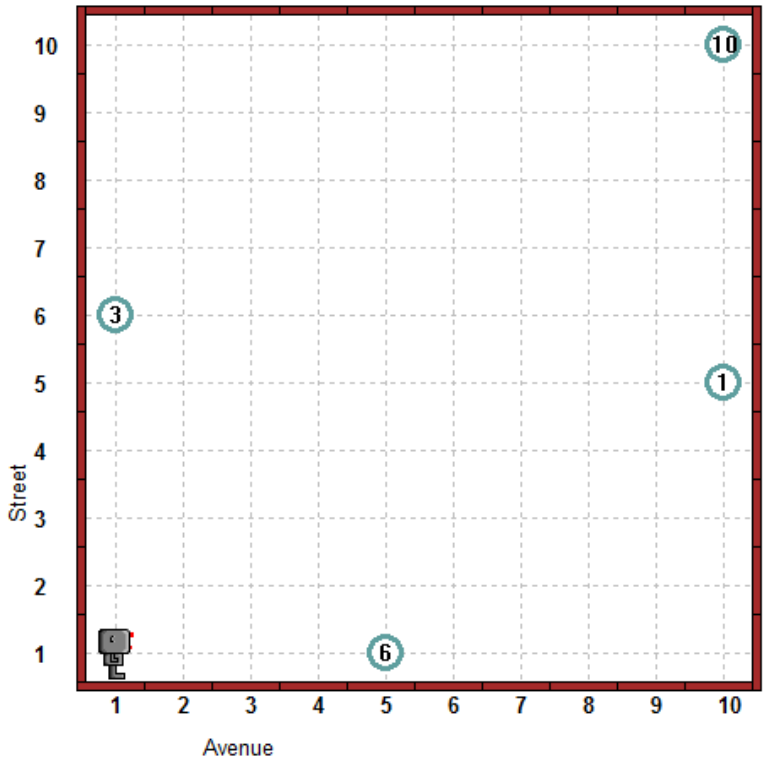
Agora, quando o robô encontrar uma parede a sua frente, sairá do loop dentro desse ***while***. E o que ele deve fazer nesse momento para continuar seu caminho pela borda do mundo? Ele precisa virar à esquerda. Esse comando deve ficar onde: dentro ou fora do ***while***? Como dissemos, ele vai virar à esquerda depois que tiver uma parede à frente. Logo, o comando fica fora do ***while***.

```
1 while front_is_clear():
2     if on_beeper():
3         pick_beeper()
4     else:
5         put_beeper()
6     move()
7 turn_left()
```

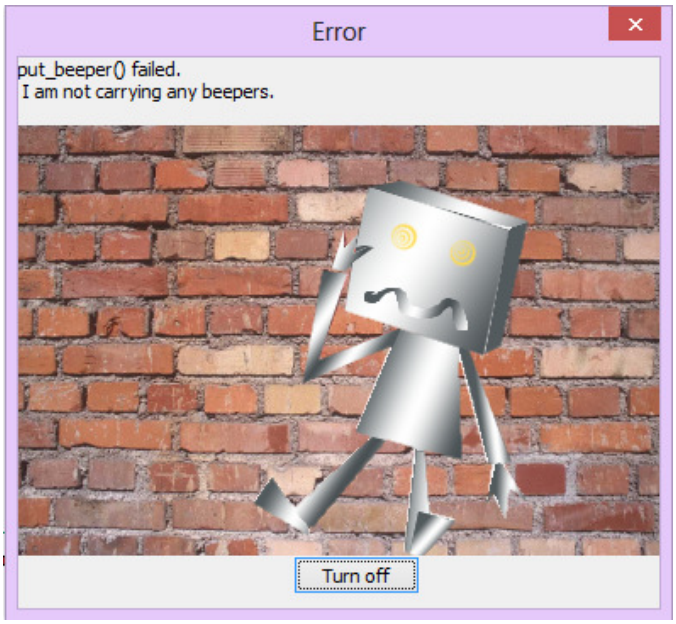
Até agora, conseguimos percorrer apenas 1 lado do mundo. Sabemos que o mundo é quadrado e, portanto, tem exatamente 4 lados. E, para cada lado, queremos repetir exatamente o trecho de código que já escrevemos. Que comando devemos usar, então? Um ***for***, com 4 repetições.

```
1 for i in range(4):
2     while front_is_clear():
3         if on_beeper():
4             pick_beeper()
5         else:
6             put_beeper()
7         move()
8     turn_left()
9 turn_off()
```

E fim, certo? Nosso programa está feito. Nos lembramos de desligar o Reeborg, para ele não reclamar (linha 9). Agora podemos rodar o programa. Vou usar o seguinte mundo, e meu Reeborg vai começar com 4 beepers na sacola:



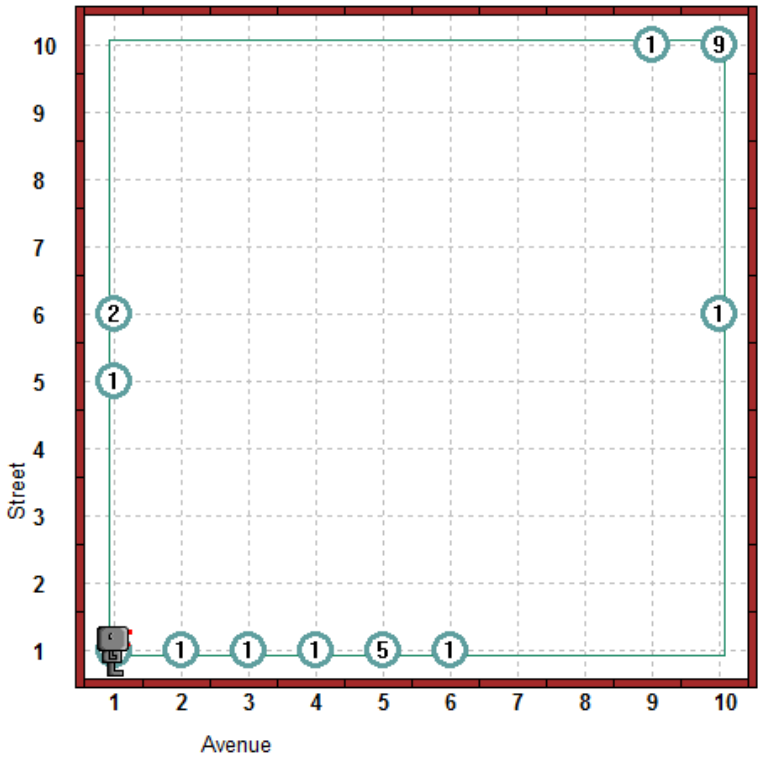
Eu obtive um erro, você também? “put_beeper() failed. I am not carrying any beepers.”



Isso significa que a sacola do Reeborg ficou vazia e nós não percebemos. Mas no enunciado do problema foi dito que o robô deveria coloca um beeper na sua posição atual desde que ele carregasse beepers na sua cesta, e nós não testamos isso! Naquela tabela das funções de teste das condições do mundo existe o método ***carries_beepers()***, que justamente verifica isso. Como precisamos fazer um teste, que comando devemos usar? Um ***if***. Precisa ser um bloco ***if-else***? Não, porque não precisamos decidir entre 2 ações, é apenas um teste simples. Assim, a versão correta do nosso programa é a seguinte:

```
1  for i in range(4):
2      while front_is_clear():
3          if on_beeper():
4              pick_beeper()
5          else:
6              if carries_beepers():
7                  put_beeper()
8              move()
9      turn_left()
10 turn_off()
```

O estado final do mundo que eu usei, após rodar o programa, foi o seguinte:



 # --------- Fim do passo-a-passo --------- #

- Vocês se lembram de que ressaltamos o espaçamento antes de algumas linhas na aula passada? Que tipos de linhas precisam disso?
    - Dentro dos métodos, dos comandos condicionais e dos comandos iterativos.
- Já tentaram tirar esse espaçamento? O que acontece?
- Esse espaçamento antes de certas linhas de um código é o que chamamos de ***indentação***. A indentação tem vários níveis (geralmente contamos a partir do zero, desde a extrema esquerda), e ressalta a hierarquia entre os elementos do código. Geralmente, uma tabulação a mais equivale a um nível de indentação a mais.
    - Essa organização do código nos ajuda a identificar os ***blocos de comando***, que começam com um ***cabeçalho***. No nosso caso, esse cabeçalho é uma linha que termina com "**:**". É possível termos blocos de comando aninhados, ou seja, blocos com sub-blocos. Assim, todas as linhas dentro do mesmo bloco possuem nível igual ou maior de indentação, nunca menor.

  - Para algumas linguagens de programação, a indentação é apenas uma forma de organizar o código. No nosso caso, a indentação correta é essencial! Caso contrário, nossos programas não funcionarão.

- Todos os programas que fizemos até agora foram relativamente simples. Mas como agir com problemas mais complexos? Imaginem que vocês querem construir uma casa de alvenaria e já possuem todo o material necessário. Não basta juntar todo esse material, é preciso combiná-lo e estabelecer uma ordem para a construção. Para isso, não é mais fácil dividir a obra em etapas? Por exemplo, primeiro fazemos a planta da casa, que é a nossa meta; depois fazemos fundação, colunas, paredes, teto, encanamento, fiação; colocamos os revestimentos, janelas e portas, pintamos as paredes.
  - E o que esse exemplo enorme tem a ver com programação? É que a maneira mais eficaz de resolver um problema complexo é dividi-lo em subproblemas sucessivamente mais simples. Você começa quebrando a tarefa como um todo em partes mais simples. Algumas dessas tarefas podem, elas próprias, precisar de subdivisão adicional, e assim por diante, até chegarmos a um nível de detalhe suficiente para sermos capazes de compreender minuciosamente o problema. Este processo é chamado de refinamento sucessivo ou decomposição.
    - No caso de um programa, o nível "mais baixo" de detalhe é aquele que o computador compreende diretamente, composto das instruções da linguagem que estamos usando.
    - Como saber se as etapas têm um bom tamanho? Elas devem ser fáceis de explicar! Uma indicação de que você foi bem sucedido é ser capaz de encontrar nomes simples para as tarefas.
- E os pedreiros, quantas vezes eles precisam aprender a executar cada uma dessas etapas? Será que a cada janela e porta que eles colocam, precisam que alguém os ensine de novo como fazer? Não, eles aprendem um jeito de fazer essas etapas num momento anterior à obra, e ao longo dela não precisam reaprendê-lo! Mas e se uma janela for arredondada na parte de cima e outra for reta? Não importa, eles aprenderam como fazer cada uma delas antes de começar a obra, e são capazes de fazer quantas dessas janelas forem necessárias!

    - O que isso quer dizer? Vocês lembram que podemos criar métodos? Eles servem (e muito!) justamente para isso: ensinar uma única vez ao computador como desempenhar certa tarefa. Assim, toda vez que você quiser executá-la, basta chamar o método ao longo do código.

- Ah, mas os meus pedreiros aprenderam direitinho a fazer janelas! De todos os tipos! Mas só se elas estiverem a ¼ da lateral esquerda da parede, 1,2m distante do chão, com 1m de altura, e só no 1º andar!... Vocês querem contratar essa equipe? Não, porque eles não sabem ser *genéricos*.
    - Vocês lembram quando ensinamos o Reeborg a dar a volta em torno de um mundo de qualquer tamanho, e não somente num mundo 5x5? O que fizemos foi tornar essa tarefa genérica. As etapas devem ser o mais genéricas quanto possível. Por que isso é bom? Porque ferramentas de programação são reutilizadas o tempo todo. Se os seus métodos realizarem tarefas genéricas, eles serão muito mais fáceis de reutilizar.
- Ah, eu conheço um pedreiro ótimo em fazer chang e mun! (*/tchán/*: janelas em coreano; portas) Pode contratá-lo sem medo! Vocês entenderam alguma coisa??? Não seria melhor se ele usasse termos fáceis de compreender? O mesmo vale para os seus códigos! Se você tem um método que faz o trabalho esperado, mas cujo nome não faz sentido no contexto do problema, até você mesmo pode não entender o que é aquele trecho quando olhar o programa depois de um tempo.