Modelagem de eventos discretos em SimPy

Simulation in Python

Afonso C. Medina (org.)
Afonso C. Medina
Afonso C. Medina

Primeira edição - 2016

Tabela de conteúdos

Parte I: Introdução

README 1.1

Introdução 1.2

- Introdução ao SimPy 1.2.1
- Instalando o SimPy 1.2.2
- Introdução ao Python 1.2.3
- Solução desafio 1 1.2.4

Criando as primeiras entidades 1.3

- Solução dos desafios 2 e 3 1.3.1

Criando, ocupando e desocupando recursos 1.4

- Exemplo Fila M\/M\/1 1.4.1
- Solução dos desafios 4, 5 e 6 1.4.2

Atributos e variáveis 1.5

- Solução dos desafios 7 e 8 1.5.1

Environments: controlando a simulação 1.6

- Solução dos desafios 9 e 10 1.6.1

Recursos com prioridade 1.7

- Solução dos desafios 11 e 12 1.7.1

Interrupções de processos 1.8

- Interrompendo o processo sem try except 1.8.1
- Solução dos desafios 13 e 14 1.8.2

Selecionando um recurso específico para um processo 1.9

- Solução dos desafios 15 e 16 1.9.1

Enchendo ou esvaziando tanques, caixas ou estoques 1.10

- Solução dos desafios 17 e 18 1.10.1

Criando lotes (ou agrupando) entidades durante a simulação 1.11

- Solução dos desafios 19 e 20 1.11.1

Parte II: Eventos

Events: os diversos tipos de Eventos em SimPy 2.1
Solução dos desafios 21 e 22 2.1.1
Aguardando múltiplos eventos 2.2
Solução dos desafios 23 e 24 2.2.1
Propriedades úteis dos eventos 2.3
Solução do desafio 25 2.3.1
Adicionando funções aos eventos com callbacks 2.4
Solução do desafio 26 2.4.1
Interrupções de eventos 2.5
O que são funções geradoras? (Ou como funciona o SimPy) 2.6
O que são funções geradoras? (Ou como funciona o SimPy?) Parte II 2.7

Parte III: Controlando os Parâmetros de Saída do Modelo

[Um exemplo de simulação e otimização]Parte-III\/(um_exemplo_de_simulacao_e_otimizacao.md) 3.1
Simulação de Agentes em SimPy! 3.2
Gravar Saída em Arquivo ou Planilha 3.3
Construção de modelos de simulação 3.4

Modelagem de eventos discretos em SimPy: See me, Fell me, README

Este livro é uma breve introdução para quem deseja conhecer o ambiente **SimPy** (*Simulation in Python*) para construção de modelos de simulação de eventos discretos em Python.

Propositadamente, o livro é dividido em seções limitadas em tamanho, com características similares a um tutorial. Assim, aqueles que desejam conhecer esta fascinante ferramenta em Python para construção de modelos de simulação, poderão sempre revisitar o material como uma fonte rápida de referência. Mas, antes, um pequeno aviso:

> Estamos pressupondo que você possui conhecimentos de Modelagem e Simulação de Eventos Discretos ou, ao menos, está fazendo algum curso sobre o assunto ou mesmo lendo o prestigioso livro Modelagem e Simulação de Eventos Discretos (Chwif & Medina)!

Portanto, este livro não se pretende como um referencial básico para a simulação de eventos discretos, mas mas sim, um livro sobre uma das linguagens mais facinantes disponíveis para se construir um modelo de simulação de eventos discretos.

Minha sugestão inicial: mentalize e reflita sobre um projeto de simulação que gostaria de desenvolver em SimPy e, a cada seção vencida do livro, desenvolva em paralelo o seu próprio projeto em harmonia com o conteúdo aprendido.

Este tutorial é produto do trabalho colaborativo entre colegas que ensinam, pesquisam ou utilizam profissionalmente ferramentas de Simulação no seu dia-a-dia:

Afonso C. Medina

Parte I: Introdução

Apresentação

Este livro é construído a partir de seções compactas. A ideia é elaborar textos curtos, para que o leitor, a cada seção, tenha uma visão clara do conteúdo apresentado, construindo o conhecimento de maneira sólida e respeitando a sua curva de aprendizado.

SimPy (*Simulation in Python*) é um *framework*[1] para a construção de modelos de simulação de eventos discretos em Python e distribuído segundo a licença MIT. Ele se diferencia dos softwares usuais de simulação, pois não se trata de uma aplicação com objetos prontos, facilmente conectáveis entre si por simples cliques do mouse, como a maioria dos softwares comerciais de simulação. Com o **SimPy**, cabe ao usuário construir um programa de computador em Python que represente seu modelo. Essencialmente, **SimPy** é uma biblioteca de comandos que conferem ao Python o poder de construir modelos de eventos discretos.

Com **SimPy** é possível pode-se construir, além de modelos de simulação discreta, modelos de simulação em “Real Time”, modelos de agentes e até mesmo modelos de simulação contínua. De fato, e como você notará ao longo deste texto, essas possibilidades estão mais associadas ao Python do que propriamente aos recursos fornecidos pelo **SimPy**.

Por que utilizar o SimPy?

Talvez a pergunta correta seja: "por que utilizar o Python?"

Python é hoje, talvez, a linguagem mais utilizada no meio científico e uma breve pesquisa pela Internet vai sugerir artigos, posts e intermináveis discussões sobre os porquês desse sucesso todo. Eu resumiria o sucesso do Python em 3 grandes razões:

- **Facilidade de codificação**. Engenheiros, matemáticos e pesquisadores em geral querem pensar no problema, nem tanto na linguagem e Python cumpre o que promete quando se fala em facilidade. Se ela é fácil de codificar, mais fácil ainda é ler e interpretar um código feito em Python;
- **Bibliotecas! Bibliotecas!** Um número inacreditável de bibliotecas (particularmente para a área científica) está disponível para o programador (e pesquisador).
- **Scripts**. A funcionalidade de trabalhar com scripts[2] ou pequenos trechos de código interpretado (basicamente, Python é uma linguagem script) diminui drasticamente o tempo de desenvolvimento e aprendizado da linguagem.

Além disso, SimPy, quando comparado com pacotes comerciais, é gratuito - o que por si só é uma grande vantagem num mercado em que os softwares são precificados a partir de milhares de dólares - e bastante
flexível, no sentido de que não é engessado apenas pelos módulos existentes.

Sob o aspecto funcional, SimPy se apresenta como uma *biblioteca* em Python e isso significa que um modelo de simulação desenvolvido com ele, terá à disposição tudo que existe de bom para quem programa em Python: o código fica fácil de ler (e desenvolver), o modelo pode ser distribuído como um pacote (sem a necessidade do usuário final instalar o Python para executá-lo), além das diversas bibliotecas de estatística e otimização disponíveis em Python, que ampliam em muito o horizonte de aplicação dos modelos.

Esta disponibilidade de bibliotecas, bem como ser um software livre, torna o SimPy particularmente interessante para quem está desenvolvendo suas pesquisas acadêmicas na área de simulação. O seu modelo provavelmente ficará melhor documentado e portanto mais fácil de ser compreendido, potencializando a divulgação dos resultados de sua pesquisa em dissertações, congressos e artigos científicos.

Prós e contras

Prós:

- Código aberto e livre (licença MIT);
- Diversas funções de bibliotecas de otimização, matemática e estatística podem ser incorporadas ao modelo;
- Permite a programação de lógicas sofisticadas, apoiando-se no Python (e suas bibliotecas);
- Comunidade ativa de desenvolvedores e usuários que mantém a biblioteca atualizada;

Contras:

- Ausência de ferramentas para animação;
- Necessidade de se programar cada processo do modelo;
- Exige conhecimento prévio em Python;
- Não inclui um ambiente visual de desenvolvimento.

Um breve histórico do SimPy

<....>

Onde procurar ajuda sobre o SimPy

Atualmente existem três fontes de consulta sobre o SimPy na internet:

- O próprio site do projeto http://simpy.readthedocs.io, com exemplos e uma detalhada descrição da da Interface de Programação de Aplicações (*Application Programming Interface* - API);
- A lista de discussão de usuários é bastante ativa, com respostas bem elaboradoras;

- O Stack Overflow tem um número razoável de questões e exemplos, mas cuidados pois boa parte do material ainda refere-se à versão 2 do SimPy.

Desenvolvimento deste livro

Este texto foi planejado em formato de seções compactas, de modo que sejam curtas e didáticas – tendo por meta que cada seção não ultrapasse muito além de 500 palavras.

Esta é a primeira edição de um livro sobre uma linguagem que vem apresentando um interesse crescente. Naturalmente, o aumento de usuários - e espera-se que este livro contribua para isso - provocará também o surgimento de mais conhecimento, mais soluções criativas e que deverão ser incorporadas neste livro em futuras revisões.

O plano proposto por este texto é caminhar pela seguinte sequência:

- Introdução ao SimPy
- Instalação
- Conceitos básicos: entidades, recursos, filas etc.
- Conceitos avançados: prioridade de recursos, compartilhamento de recursos, controle de filas *Store* de recursos etc.
- Experimentação (replicações, tempo de *warm-up*, intervalos de confiança etc.)
- Aplicações

> [1]. Em português, ainda é comum o estrangeirismo "framework" entre profissionais da Ciência da Computação. ↩

> [2]. Script é uma sequência de comandos executados no interior de algum programa por meio de um interpretador. ↩

Instalando o SimPy

Nossa jornada começa por um tutorial de instalação de alguns programas e bibliotecas úteis para o SimPy. Selecionamos, para começar o tutorial, os seguintes pacotes:

1. Python 3.4
2. Pip
3. SimPy 3.0.10
4. NumPy

Um breve preâmbulo das nossas escolhas: no momento da elaboração deste tutorial, o **Python** já está na versão 3.5.0, mas o **Anaconda** (explicado adiante) ainda fornece a versão 3.4. Se você já possui uma instalação com o Python 3.5, pode instalar o SimPy sem problemas, pois ele é compatível com as versões mais recentes do Python.

Pip é um instalador de bibliotecas e facilita muito a vida do programador.

O **SimPy** 3.0.10 é a versão mais atual no momento em que este tutorial é escrito e traz grandes modificações em relação à versão 2.0.

> Atenção: existe vasto material disponível na Internet para o SimPy. Contudo, um cuidado especial deve ser tomado: grande parte deste material refere-se a versão 2.0, que possui diferenças críticas em relação à versão mais atual. Este texto é para a versão 3 em diante.

Quanto ao **NumPy**, vamos aproveitar o embalo para instalá-lo, pois será muito útil nos nossos modelos de simulação. Basicamente, NumPy acrescenta um tipo de dados (*n-dimensional array*) que facilita a codificação de modelos de simulação, particularmente na análise de dados de saída do modelo.

Passo 1: Anaconda, the easy way

Atenção:

> - Se você já tem o Python e o Pip instalados em sua máquina, pule diretamente para o Passo 3: “Instalando o SimPy”;
> - Se você já tem o Python instalado, mas não o Pip (quem tem o Python +3.4, já tem o Pip instalado), pule para o Passo 2: “Instalando o Pip”

Se esta é a sua primeira vez, nossa sugestão: não perca tempo e instale a distribuição gratuita Anaconda.

Anaconda

Por meio do Anaconda, tudo é mais fácil, limpo e o processo já instala mais de 200 pacotes verificados por toda sorte de compatibilidade, para que você não tenha trabalho algum. (Entre os pacotes instalados está o NumPy que, como explicado, será muito útil no desenvolvimento dos seus modelos).

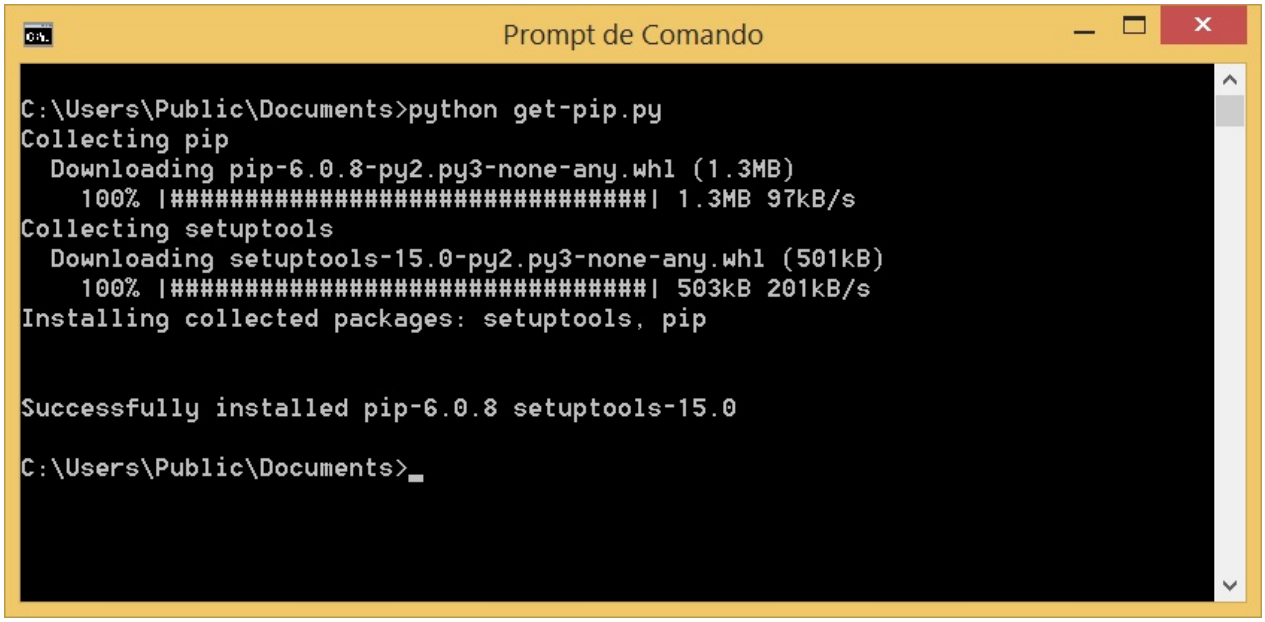
Atualmente eles disponibilizam as versões 2.7, 3.4 e 3.5 do Python (em 32 e 64 bit) na página de downloads.

Baixe o arquivo com a versão desejada (mais uma vez: SimPy roda nas duas versões) e siga as instruções do instalador.

Passo 2: Instalando o Pip (para quem não instalou o Anaconda)

> Se a versão instalada do Python for +3.4 ou você fez o passo anterior, pode pular este passo, pois o pip já deve estar instalado no seu computador.

1. Baixe o pacote `get-pip.py` por este link para o seu computador, salvando-o em uma pasta de trabalho conveniente.
2. Execute `python get-pip.py` na pasta de trabalho escolhida (note a mensagem final de que o pip foi instalado com sucesso).

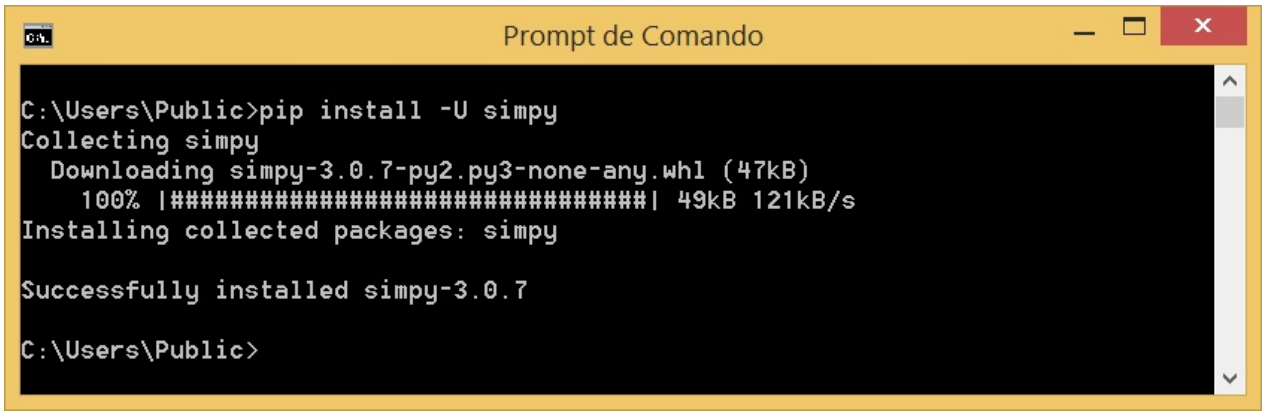


Passo 3: Instalando o Simpy

Instalar o Simpy é fácil!

Digite numa janela cmd:

```
pip install -U simpy
```



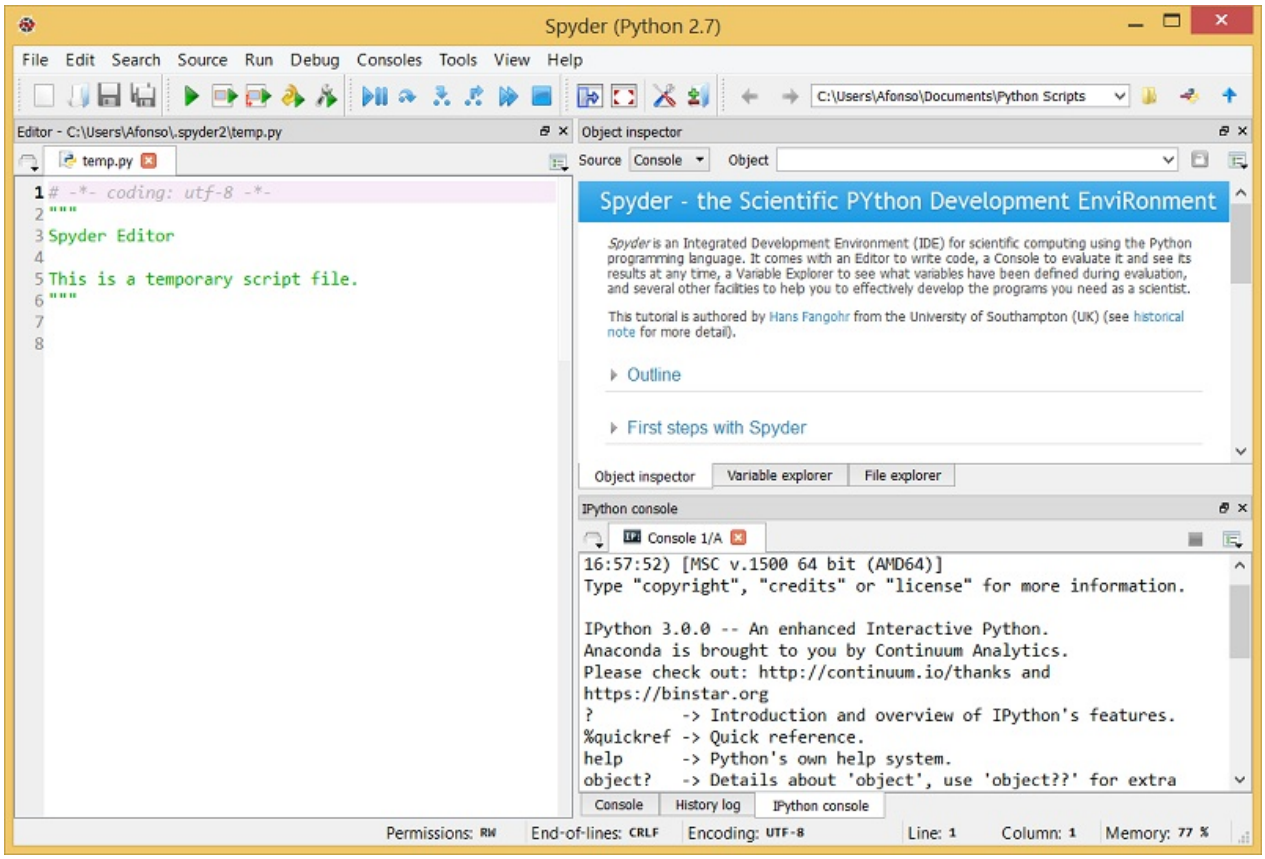
A mensagem "Sucessfully" indica que você já está pronto para o SimPy. Mas, antes disso, tenho uma sugestão para você:

Passo 4: Instalando um Ambiente Integrado de Desenvolvimento (IDE) para aumentar a produtividade no Python

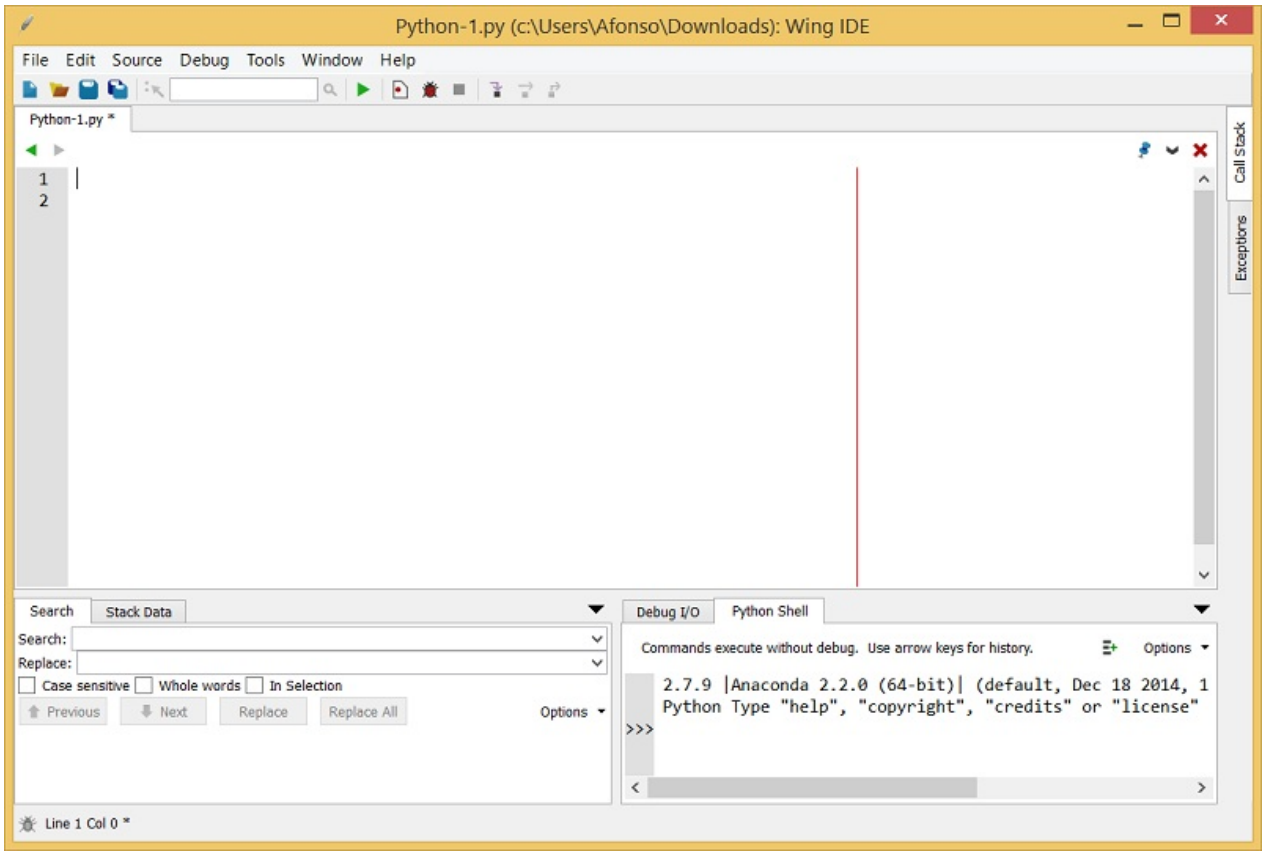
Os IDEs, para quem não conhece, são interfaces que facilitam a vida do programador. Geralmente possuem um editor de textos avançado, recursos de verificação de erros, monitoram os estados de variáveis, permitem o processamento passo-a-passo etc.

Se você instalou o Anaconda, então já ganhou um dos bons: o Spyder, que já está configurado e pronto para o uso. Geralmente (a depender da sua versão do Sistema Operacional) ele aparece como um ícone na área de trabalho. Se não localizar o ícone, procure por Spyder no seu computador (repare no *"y"*) ou digite em uma janela de **cmd** o comando `spyder`.

Aberto, o Spyder fica como a figura a seguir (repare que a janela superior-direita apresenta um tutorial de uso):



Outro IDE muito bom é o Wing IDE 101 que é gratuito (e possui uma versão profissional paga):



> Atenção: se você instalou o Anaconda e pretende utilizar algum IDE que não o Spyder, siga as instruções deste link, para configurar corretamenta seu IDE. (É fácil, não se avexe não!)

Se você chegou até aqui com tudo instalado, o próximo passo é começar para valer com o SimPy!

Na próxima seção, é claro. Precisamos tomar um chazinho depois deste zilhões de bytes instalados.

Começando pelo Python

Antes de começarmos com o SimPy, precisamos garantir que você tenha algum conhecimento mínimo de Python. Se você julga que seus conhecimentos na linguagem são razoáveis, a recomendação é que você pule para a seção seguinte, “Teste seus conhecimentos em Python”.

Se você nunca teve contato com a linguagem, aviso que não pretendemos construir uma "introdução" ou "tutorial" para o Python, simplesmente porque isso é o que mais existe à disposição na internet.

Procure um tutorial rápido (existem tutoriais de até 10 minutos!) e mãos à obra! Nada mais fácil do que aprender o básico de Python.

Sugestões:

1. Um curso muito bom no Udacity (20 horas): Programming Foundations with Python (clicando no botão azul “Access course materials”, você faz o curso de graça, mas sem receber o certificado). A vantagem do Udacity é que você pode começar o curso a hora que quiser;
2. Outro curso muito bom é o da CodeAcademy (13 horas): Python;
3. Outro curso muito bom, este no Coursera: Introdução à Programação Interativa em Python (Parte 1).
4. Eu tenho sempre na minha mesa um ótimo livro de Python: *Summerfield, Mak. Programming in Python 3: a complete introduction. Addison-Wesley Professional. 2012*. Este livro já foi traduzido para o Português e pode ser encontrado na amazon.

Feito o tutorial, curso ou aprendido mesmo tudo sozinho, teste seus conhecimentos para verificar se você sabe o *básico necessário* de Python para começar com o SimPy.

Teste seus conhecimentos em Python: o problema da ruina do apostador

O problema da ruína do apostador é um problema clássico proposto por Pascal em uma carta para Fermat em 1656. A versão aqui apresentada é uma simplificação visando avaliar seus conhecimentos em Python.

> **Desafio 1**: dois apostadores iniciam um jogo de cara ou coroa em que cada um deles aposta $1 sempre em um mesmo lado da moeda. O vencedor leva a aposta total ($2). Cada jogador tem inicialmente $10 disponíveis para apostar. O jogo termina quando um dos jogadores atinge a ruína e não tem mais dinheiro para apostar.

Construa três funções:

1. `transfer(winner, looser, bankroll, tossCount):` transfere o valor do jogador perdedor para o vencedor e imprime na tela o nome do vencedor;
2. `coinToss(bankroll, tossCount):` sorteia o vencedor do cara ou coroa;
3. `run2Ruin(bankroll):` mantém um laço permanente até que um dos jogadores entrem em ruína

Teste o programa com os parâmetros a seguir (você pode utilizar esse código como uma máscara para iniciar o seu programa):

```
import random                   # gerador de números aleatórios

names = ['Chewbacca', 'R2D2']   # jogadores

def transfer(winner, looser, bankroll, tossCount):
    # função que transfere o dinheiro do winner para o looser
    # imprime o vencedor do lançamento  e o bankroll de cada jogador
    pass

def coinToss(bankroll, tossCount):
    # função que sorteia a moeda e chama a transfer
    pass

def run2Ruin(bankroll):
    # função que executa o jogo até a ruina de um dos jogadores
    pass

bankroll = [5, 5]               # dinheiro disponível para cada jogador
run2Ruin(bankroll)              # inicia o jogo
```

Agora é com você: complete o código anterior e descubra se você está pronto para inciar com o SimPy!
(A próxima seção apresenta uma possível resposta para o desafio e, na sequência, tudo enfim, começa.)

Solução do desafio 1

O código a seguir é uma possível solução para o desafio 1 da seção anterior. Naturalmente é possível deixá-lo mais claro, eficiente, obscuro, maligno, elegante, rápido ou lento, como todo código de programação.

O importante é que se você fez alguma que coisa que funcionou, acredito que é o suficiente para começar com o SimPy.

```
import random                   # gerador de números aleatórios

names = ['Chewbacca', 'R2D2']   # jogadores

def transfer(winner, looser, bankroll, tossCount):
    # função que transfere o dinheiro do winner para o looser
    # imprime o vencedor do lançamento  e o bankroll de cada jogador
    bankroll[winner] += 1
    bankroll[looser] -= 1
    print("\nLançamento: %d\tVencedor: %s" % (tossCount, names[winner]))
    print("%s possui: $%d e %s possui: $%d"
            % (names[0], bankroll[0], names[1], bankroll[1]))

def coinToss(bankroll, tossCount):
    # função que sorteia a moeda e chama a transfer
    if random.uniform(0, 1) < 0.5:
        transfer(1, 0, bankroll, tossCount)
    else:
        transfer(0, 1, bankroll, tossCount)

def run2Ruin(bankroll):
    # função que executa o jogo até a ruina de um dos jogadores
    tossCount = 0               # contador de lançamentos
    while bankroll[0] > 0 and bankroll[1] > 0:
        tossCount += 1
        coinToss(bankroll,tossCount)
    winner = bankroll[1] > bankroll[0]
    print("\n%s venceu depois de %d lançamentos, fim de jogo!"
            % (names[winner], tossCount))

bankroll = [5, 5]               # dinheiro disponível para cada jogador
run2Ruin(bankroll)              # inicia o jogo
```

No meu computador, o problema anterior fornece o seguinte resultado:

```
Lançamento: 1   Vencedor: Chewbacca
Chewbacca possui: $6 e R2D2 possui: $4

Lançamento: 2   Vencedor: Chewbacca
Chewbacca possui: $7 e R2D2 possui: $3

Lançamento: 3   Vencedor: Chewbacca
Chewbacca possui: $8 e R2D2 possui: $2

Lançamento: 4   Vencedor: Chewbacca
Chewbacca possui: $9 e R2D2 possui: $1

Lançamento: 5   Vencedor: Chewbacca
Chewbacca possui: $10 e R2D2 possui: $0

Chewbacca venceu depois de 5 lançamentos, fim de jogo!
```

Note que o resultado fornecido deve ser diferente em seu computador, assim como ele se modifica a cada nova rodada no programa. Para que os resultados sejam semelhantes, precisamos utilizar o comando `random.seed(semente)` com um mesmo valor inicial (veja o item 1 na seção "Teste seus conhecimentos a seguir").

Teste seus conhecimentos:

1. Cada vez que você executa o programa, a função `random.uniform(0, 1)` sorteia um novo número aleatório ente 0 e 1, tornando imprevisível o resultado do programa. Utilize a função `random.seed(semente)` para fazer com que a sequência gerada de números aleatórios seja sempre a mesma.
2. Acrescente um laço no programa principal de modo que o jogo possa ser repetido até um número pré definido de vezes. *Simule* 100 partidas e verifique em quantas cada um dos jogadores venceu.

Primeiro passo em SimPy: criando entidades

Algo elementar em qualquer pacote de simulação é uma função para criar entidades dentro do modelo. É o "Alô mundo!" dos pacotes de simulação. Nossa primeira missão será construir uma função que gere entidades com intervalos entre chegadas sucessivas exponencialmente distribuídos, com média de 2 min. Vamos simular o sistema por 10 minutos apenas.

Chamada das bibliotecas `random` e `simpy`

Inicialmente serão necessárias duas bibliotecas do Python: a `random` – biblioteca de geração de números aleatórios – e a `simpy`, que é o próprio SimPy.

Começaremos nosso primeiro modelo de simulação em SimPy chamando as bibliotecas de interesse:

```
import random             # gerador de números aleatórios
import simpy              # biblioteca de simulação

random.seed(1000)         # semente do gerador de números aleatórios
```

Note a linha final `random.seed(1000)`, ela garante que a geração de números aleatórios sempre começará pela mesma semente. Na prática, a sequência de números aleatórios gerados será sempre a mesma, facilitando o processo de verificação do programa.

Criando um `evironment` de simulação

Tudo no SimPy gira em torno de **eventos** criados por funções e todos os eventos ocorrem num **environment**, ou um "ambiente" de simulação criando a partir da função `simpy.Environment()`. Assim, todo programa principal sempre começa com uma chamada ao SimPy, criando um *environment* "env":

```
import random             # gerador de números aleatórios
import simpy              # biblioteca de simulação

random.seed(1000)         # semente do gerador de números aleatórios
env = simpy.Environment() # cria o environment do modelo na variável env
```

Se você executar o programa anterior, nada acontece. No momento, você apenas criou um *environment*, mas não criou nenhum processo, portanto, não existe ainda nenhum evento a ser simulado pelo SimPy.

Criando um gerador de chegadas dentro do `environment`

Vamos escrever uma função `geraChegadas()` que cria entidades no sistema enquanto durar a simulação. Nosso primeiro gerador de entidades terá três parâmetros de entrada: o *environment*, um atributo que representará o nome da entidade e a taxa desejada de chegadas de entidades por unidade de tempo. Para o SimPy, equivale dizer que você vai construir uma *função geradora de eventos* dentro do *environment* criado. No caso, os eventos gerados serão as chegadas de entidades no sistema.

Assim, nosso código começa a ganhar corpo:

```
import random             # gerador de números aleatórios
import simpy              # biblioteca de simulação

def geraChegadas(env, nome, taxa):
    # função que cria chegadas de entidades no sistema
    pass

random.seed(1000)         # semente do gerador de números aleatórios
env = simpy.Environment() # cria o environment do modelo
```

Precisamos informar ao SimPy que a função `geraChegadas()` é, de fato, um processo que deve ser executado ao longo de toda a simulação. Um processo é criado dentro do `environment`, pelo comando:

```
env.process(função_que_gera_o_processo)
```

A chamada ao processo é sempre feita após a criação do `env`, então basta acrescentar uma nova linha ao nosso código:

```
import random             # gerador de números aleatórios
import simpy              # biblioteca de simulação

def geraChegadas(env, nome, taxa):
    # função que cria chegadas de entidades no sistema
    pass

random.seed(1000)         # semente do gerador de números aleatórios
env = simpy.Environment() # cria o environment do modelo
# cria o processo de chegadas
env.process(geraChegadas(env, "Cliente", 2)))
```

Criando intervalos de tempo de espera com `env.timeout(tempo_de_espera)`

Inicialmente, precisamos gerar intervalos de tempos aleatórios, exponencialmente distribuídos, para representar os tempos entre chegadas sucessivas das entidades. Para gerar chegadas com intervalos exponenciais, utilizaremos a biblioteca `random`, bem detalhada na sua documentação, e que possui a função:

```
random.expovariate(lambd)
```

Onde `lambd` é a taxa de ocorrência dos eventos ou, matematicamente, o inverso do tempo médio entre eventos sucessivos. No caso, se queremos que as chegadas ocorram entre intervalos médios de 2 min, a função ficaria:

```
random.expovariate(lambd=1.0/2.0)
```

A linha anterior é basicamente nosso gerador de números aleatórios exponencialmente distribuídos. O passo seguinte será informar ao SimPy que queremos nossas entidades surgindo no sistema segundo a distribuição definida. Isso é feito pela chamada da palavra reservada `yield` com a função do SimPy `env.timeout(intervalo)`, que nada mais é do que uma função que causa um atraso de tempo, um *delay* no tempo dentro do *enviroment* `env` criado:

```
yield env.timeout(random.expovariate(1.0/2.0))
```

Na linha de código anterior estamos executando `yield env.timeout(0.5)` para que o modelo retarde o processo num tempo aleatório gerado pela função `random.expovariate(0.5)`.

Oportunamente, discutiremos mais a fundo qual o papel do palavra `yield` (*spoiler*: ela não é do SimPy, mas originalmente do próprio Python). Por hora, considere que ela é apenas uma maneira de **criar eventos** dentro do `env` e que, caso uma função represente um processo, obrigatoriamente ela precisará conter o comando `yield *alguma coisa*`, bem como o respectivo `environment` do processo.

> Atenção: uma função criada no Python (com o comando `def`) só é tratada como um **processo** ou **gerador de eventos** para o SimPy, caso ela contenha ao menos uma linha de código com o comando `yield`. Mais adiante, a seção "O que são funções geradoras" explica em mais detalhe o funcionamento do `yield`.

Colocando tudo junto na função `geraChegadas()`, temos:

```
import random             # gerador de números aleatórios
import simpy              # biblioteca de simulação

def geraChegadas(env, nome, taxa):
    # função que cria chegadas de entidades no sistema
    contaChegada = 0
    while True:
        yield env.timeout(random.expovariate(1.0/taxa))
        contaChegada += 1
        print("%s %i chega em: %.1f " % (nome, contaChegada, env.now))

random.seed(1000)         # semente do gerador de números aleatórios
env = simpy.Environment() # cria o environment do modelo
# cria o processo de chegadas
env.process(geraChegadas(env, "Cliente", 2)))
```

O código deve ser autoexplicativo: o laço `while` é **infinito** enquanto dure a simulação; um contador, `contaChegada`, armazena o total de entidades geradas e a função `print`, imprime na tela o instante de chegada de cada cliente. Note que, dentro do `print`, existe uma chamada para a **hora atual de simulação** `env.now`.
Por fim, uma chamada a função `random.seed()` garante que os números aleatórios a cada execução do programa serão os mesmos.

Executando o modelo por um tempo determinado com `env.run(until=tempo_de_simulacao)`

Se você executar o código anterior, nada acontece novamente, pois ainda falta informarmos ao SimPy qual o tempo de duração da simulação. Isto é feito pelo comando:
`env.run(until=tempo_de_simulação)`
No exemplo proposto, o tempo de simulação deve ser de 10 min.

```
import random             # gerador de números aleatórios
import simpy              # biblioteca de simulação

def geraChegadas(env, nome, taxa):
    # função que cria chegadas de entidades no sistema
    contaChegada = 0
    while True:
        yield env.timeout(random.expovariate(1/taxa))
        contaChegada += 1
        print("%s %i chega em: %.1f " % (nome, contaChegada, env.now))

random.seed(1000)         # semente do gerador de números aleatórios
env = simpy.Environment() # cria o environment do modelo
env.process(geraChegadas(env, "Cliente", 2)) # cria o processo de chegadas
env.run(until=10) # roda a simulação por 10 unidades de tempo
```

Ao executar o programa, temos a saída:

```
Cliente 1 chega em: 3.0
Cliente 2 chega em: 5.2
Cliente 3 chega em: 5.4
Cliente 4 chega em: 6.3
Cliente 5 chega em: 7.6
Cliente 6 chega em: 9.1
```

Agora sim!

Note que `env.process(geraChegadas(env))` é um comando que **torna** a função `geraChegadas()` um **processo** ou um **gerador de eventos** dentro do `Environment env`. Esse processo só começa a ser executado na linha seguinte, quando `env.run(until=10)` informa ao SimPy que todo processo pertencente ao `env` deve ser executado por um **tempo de simulação** igual a 10 minutos.

Conceitos desta seção

Conteúdo	Descrição
`env = simpy.Environment()`	cria um `Environment` de simulação
`random.expovariate(lambd)`	gera números aleatórios exponencialmente distribuídos, com taxa de ocorrência (eventos/unidade de tempo) igual a `lambd`
`yield env.timeout(time)`	gera um atraso dado por `time`
`random.seed(seed)`	define o gerador de sementes aleatórias para um mesmo valor a cada nova simulação
`env.process(geraChegadas(env))`	inicia a função `geraChegadas` como um *processo* em `env`
`env.run(until=tempoSim)`	executa a simulação (executa todos os processos criados em `env`) pelo tempo `tempoSim`
`env.now`	retorna o instante atual da simulação

Desafios (soluções na próxima seção)

> **Desafio 2:** é comum que os comandos de criação de entidades nos softwares proprietários tenham a opção de limitar o número máximo de entidades geradas durante a simulação. Modifique a função `geraChegadas` de modo que ela receba como parâmetro `numeroMaxChegadas` e limite a criação de entidades a este número.
>
> **Desafio 3:** modifique a função `geraChegadas`
> de modo que as chegadas entre entidades sejam distribuídas segundo uma distribuição triangular de moda 1, menor valor 0,1 e maior valor 1,1.

Solução dos desafios 2 e 3

> **Desafio 2:** é comum que os comandos de criação de entidades nos softwares proprietários tenham a opção de limitar o número máximo de entidades geradas durante a simulação. Modifique a função `geraChegadas` de modo que ela receba como parâmetro o `numeroMaxChegadas` e limite a criação de entidades a este número.

Neste caso, o *script* em Python é autoexplicativo, apenas note que limitei o número de chegadas em 5 e fiz isso antes da chamada do processo gerado pela função `geraChegadas()` :

```
import random     # gerador de números aleatórios
import simpy      # biblioteca de simulação

def geraChegadas(env, nome, taxa, numeroMaxChegadas):
    # função que cria chegadas de entidades no sistema
    contaChegada = 0
    while (contaChegada < numeroMaxChegadas:
        yield env.timeout(random.expovariate(1/taxa))
        contaChegada += 1
        print("%s %i chega em: %.1f " % (nome, contaChegada, env.now))

random.seed(1000)   # semente do gerador de números aleatórios
env = simpy.Environment() # cria o environment do modelo
# cria o processo de chegadas
env.process(geraChegadas(env, "Cliente", 2, 5))
env.run(until=10) # executa a simulação por 10 unidades de tempo
```

> **Desafio 3:** modifique a função `geraChegadas` de modo que as chegadas entre entidades sejam distribuídas segundo uma distribuição triangular de moda 1, menor valor 0,1 e maior valor 1,1.

Neste caso, precisamos verificar na documentação da biblioteca `random` , quais são nossas opções. A tabela a seguir, resume as distribuições disponíveis:

Função	Distribuição
`random.random()`	gera números aleatórios no intervalo [0.0, 1.0)
`random.uniform(a, b)`	uniforme no intervalo [a, b]
`random.triangular(low, high, mode)`	triangular com menor valor *low*, maior valor *high* e moda *mode*
`random.betavariate(alpha, beta)`	beta com parâmetros *alpha* e *beta*
`random.expovariate(lambd)`	exponencial com média 1/*lambd*
`random.gammavariate(alpha, beta)`	gamma com parâmetros *alpha* e *beta*
`random.gauss(mu, sigma)`	normal com média *mu* e desvio padrão *sigma*
`random.lognormvariate(mu, sigma)`	lognormal com média *mu* e desvio padrão *sigma*
`random.normalvariate(mu, sigma)`	equivalente à random.gauss, mas um pouco mais lenta
`random.vonmisesvariate(mu, kappa)`	distribuição de von Mises com parâmetros *mu* e *kappa*
`random.paretovariate(alpha)`	pareto com parâmetro *alpha*
`random.weibullvariate(alpha, beta)`	weibull com parâmetros *alpha* e *beta*

A biblioteca NumPy, que veremos oportunamente, possui mais opções para distribuições estatísticas. Por enquanto, o desafio 3 pode ser solucionado de maneira literal:

```
import random     # gerador de números aleatórios
import simpy      # biblioteca de simulação

def geraChegadas(env, nome, numeroMaxChegadas):
    # função que cria chegadas de entidades no sistema
    contaChegada = 0
    while (contaChegada < numeroMaxChegadas:
        yield env.timeout(random.triangular(0.1,1,1.1))
        contaChegada += 1
        print("%s %i chega em: %.1f " % (nome, contaChegada, env.now)))

random.seed(1000)         # semente do gerador de números aleatórios
env = simpy.Environment() # cria o environment do modelo
# cria o processo de chegadas
env.process(geraChegadas(env, "Cliente, 5))
env.run(until=10)
```

Dica

Os modelos de simulação com muitos processos de chegadas e atendimento, tendem a utilizar diversas funções diferentes de distribuição de probabilidades, deixando as coisas meio confusas para o programador.

Uma dica bacana é criar uma função que armazene todas as distribuições do modelo em um único lugar, como uma prateleira de distribuições.

Por exemplo, imagine um modelo em SimPy que possui 3 processos: um exponencial com média 10 min, um triangular com parâmetros (10, 20, 30) min e um normal com média 0 e desvio 1 minuto. A função `distribution()` a seguir, armazena todos os geradores de números aleatórios em um único local:

```
import random

def distributions(tipo):
    return {
        'arrival': random.expovariate(1/10.0),
        'singing': random.triangular(10, 20, 30),
        'applause': random.gauss(10, 1),
    }.get(tipo, 0.0)
```

O próximo exemplo testa como chamar a função:

```
tipo = 'arrival'
print(tipo, distributions(tipo))

tipo = 'singing'
print(tipo, distributions(tipo))

tipo = 'applause'
print(tipo, distributions(tipo))
```

O qual produz a saída:

```
arrival 6.231712146858156
singing 22.192356552471104
applause 10.411795571842426
```

Essa foi a nossa dica do dia!

> Fique a vontade para implementar funções de geração de números aleatórios ao seu gosto. Note, e isso é importante, que **praticamente todos os seus modelos de simulação em SimPy precisarão deste tipo de função!**

Uma última observação:

> Atenção à eficiência do código. Como a finalidade deste texto é prioritariamente didática, as chamadas às funções de geração de números aleatórios respeitam a lógica do aprendizado. Contudo, tais chamadas não são exatamente eficientes... Por exemplo, você consegue descobrir qual das duas chamadas a seguir é a mais eficiente e por quê?

```
while True:
    yield env.timeout(random.expovariate(1.0/2.0))
```

Ou:

```
while True:
    yield env.timeout(random.expovariate(0.5))
```

Resposta: note que, no primeiro caso, a cada novo número gerado é realizada uma operação de divisão. No segundo caso, isso não ocorre, deixando o tempo de processamento bem mais rápido.

Teste seus conhecimentos:

1. Acrescente ao programa inicial, uma função `distribution` como a proposta na Dica do Dia e faça o tempo entre chegadas sucessivas de entidades chamar a função para obter o valor correto.
2. Considere que 50% das entidades geradas durante a simulação são do sexo feminino e 50% do sexo masculino. Modifique o programa para que ele sorteie o gênero dos clientes. Faça esse sorteio dentro da função `distribution` já criada.

Juntando tudo em um exemplo: a fila M/M/1

A fila M/M/1 (ver Chwif e Medina, 2015 representa um sistema simples em que clientes chegam para atendimento em um servidor de fila única, com intervalos entre chegadas sucessivas exponencialmente distribuídos e tempos de atendimentos também exponencialmente distribuídos.

Para este exemplo, vamos considerar que o tempo médio entre chegadas sucessivas é de 1 min (ou seja, uma taxa de chegadas de 1 cliente/min) e o tempo médio de atendimento no servidor é de 0,5 min (ou seja, uma taxa de atendimento de 2 clientes/min). Como um experimento inicial, o modelo deve ser simulado por 5 minutos apenas.

Geração de chegadas de entidades

Partindo da função `geraChegadas,` codificada na seção "Primeiro passo em SimPy: criando entidades", precisamos criar uma função ou processo para ocupar, utilizar e desocupar o servidor. Criaremos uma função `atendimentoServidor` responsável por manter os clientes em fila e realizar o atendimento.

Inicialmente, vamos acrescentar as constantes `TEMPO_MEDIO_CHEGADAS` e `TEMPO_MEDIO_ATENDIMENTO`, para armazenar os parâmetros das distribuições dos processos de chegada e atendimento da fila. Adicionalmente, vamos criar o recurso `servidorRes` com capacidade de atender 1 cliente por vez.

```
import random                           # gerador de números aleatórios
import simpy                            # biblioteca de simulação

TEMPO_MEDIO_CHEGADAS = 1.0              # tempo médio entre chegadas sucessivas de clientes
TEMPO_MEDIO_ATENDIMENTO = 0.5           # tempo médio de atendimento no servidor

def geraChegadas(env):
    # função que cria chegadas de entidades no sistema
    contaChegada = 0
    while True:
        # aguardo um intervalo de tempo exponencialmente distribuído
        yield env.timeout(random.expovariate(1.0/TEMPO_MEDIO_CHEGADAS))
        contaChegada += 1
        print('%.1f Chegada do cliente %d' % (env.now, contaChegada))

random.seed(25)                                 # semente do gerador de números aleatórios
env = simpy.Environment()                       # cria o environment do modelo
servidorRes = simpy.Resource(env, capacity=1)   # cria o recurso servidorRes
env.process(geraChegadas(env))                  # incia processo de geração de chegadas

env.run(until=5)                                # executa o modelo por 10 min
```

Realizando o atendimento no servidor

Se você executar o script anterior, o recurso é criado, mas nada acontece, afinal, não existe ainda nenhum processo requisitando o recurso.

Precisamos, portanto, construir uma nova função que realize o *processo* de atendimento. Usualmente, um processo qualquer tem ao menos as 4 etapas a seguir:

1. Solicitar o servidor;
2. Ocupar o servidor;
3. Executar o atendimento por um tempo com distribuição conhecida;
4. Liberar o servidor para o próximo cliente.

A função `atendimentoServidor`, a seguir, recebe como parâmetros o `env` atual, o `nome` do cliente e a recurso `servidorRes` para executar todo o processo de atendimento.

```
import random                           # gerador de números aleatórios
import simpy                            # biblioteca de simulação

TEMPO_MEDIO_CHEGADAS = 1.0              # tempo médio entre chegadas sucessivas de clientes
TEMPO_MEDIO_ATENDIMENTO = 0.5           # tempo médio de atendimento no servidor

def geraChegadas(env):
    # função que cria chegadas de entidades no sistema
    contaChegada = 0
    while True:
        # aguardo um intervalo de tempo exponencialmente distribuído
        yield env.timeout(random.expovariate(1.0/TEMPO_MEDIO_CHEGADAS))
        contaChegada += 1
        print('%.1f Chegada do cliente %d' % (env.now, contaChegada))

def atendimentoServidor(env, nome, servidorRes):
    # função que ocupa o servidor e realiza o atendimento
    # solicita o recurso servidorRes
    request = servidorRes.request()

    # aguarda em fila até a liberação do recurso e o ocupa
    yield request
    print('%.1f Servidor inicia o atendimento do %s' % (env.now, nome))

    # aguarda um tempo de atendimento exponencialmente distribuído
    yield env.timeout(random.expovariate(1.0/TEMPO_MEDIO_ATENDIMENTO))
    print('%.1f Servidor termina o atendimento do %s' % (env.now, nome))

    # libera o recurso servidorRes
    yield servidorRes.release(request)

random.seed(25)                                 # semente do gerador de números aleatórios
env = simpy.Environment()                       # cria o environment do modelo
servidorRes = simpy.Resource(env, capacity=1)   # cria o recurso servidorRes
env.process(geraChegadas(env))                  # incia processo de geração de chegadas

env.run(until=5)                                # executa o modelo por 10 min
```

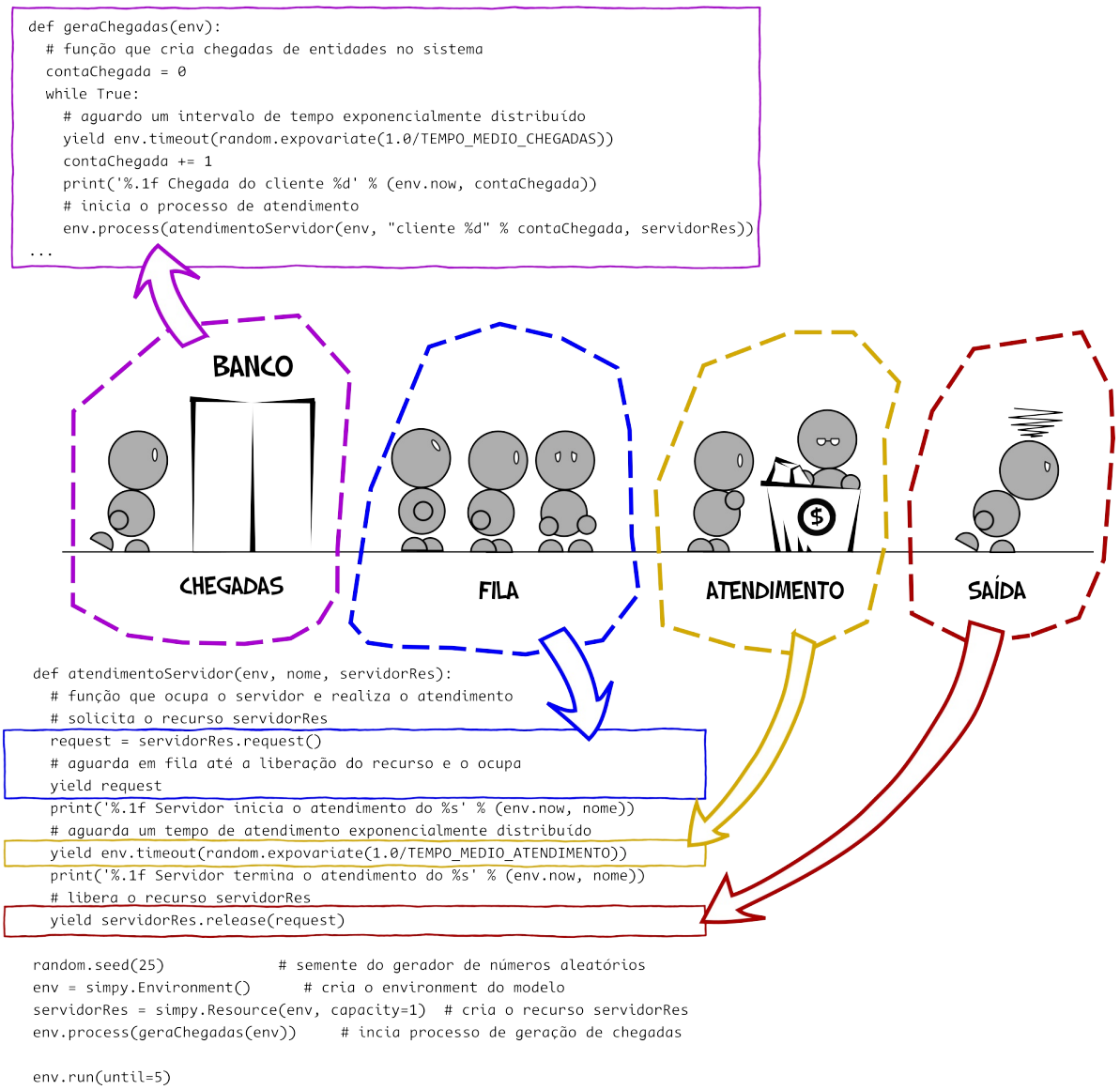
Neste momento, nosso script possui uma função geradora de clientes e uma função de atendimento dos clientes, mas o bom observador deve notar que não existe conexão entre elas. Em SimPy, *e vamos repetir isso a exaustão*, **tudo é processado dentro de um** `environment`. Assim, o atendimento é um *processo* que deve ser iniciado por cada cliente *gerado* pela função `criaChegadas.` Isto é feito por uma chamada a função `env.process(atendimentoServidor(...)).`

A função `geraChegadas` deve ser alterada, portanto, para receber como parâmetro o recurso `servidorRes,` criado no corpo do programa e para iniciar o processo de atendimento por meio da chamada à função `env.process`, como representado a seguir:

```
def geraChegadas(env):
    # função que cria chegadas de entidades no sistema
    contaChegada = 0
    while True:
        # aguardo um intervalo de tempo exponencialmente distribuído
        yield env.timeout(random.expovariate(1.0/TEMPO_MEDIO_CHEGADAS))
        contaChegada += 1
        print('%.1f Chegada do cliente %d' % (env.now, contaChegada))

        # inicia o processo de atendimento
        env.process(atendimentoServidor(env, "cliente %d" % contaChegada, servidorRes))
```

Agora execute o script e voilá!

```
0.5 Chegada do cliente 1
0.5 Servidor inicia o atendimento do cliente 1
1.4 Servidor termina o atendimento do cliente 1
3.1 Chegada do cliente 2
3.1 Servidor inicia o atendimento do cliente 2
3.3 Chegada do cliente 3
4.1 Servidor termina o atendimento do cliente 2
4.1 Servidor inicia o atendimento do cliente 3
4.1 Servidor termina o atendimento do cliente 3
4.3 Chegada do cliente 4
4.3 Servidor inicia o atendimento do cliente 4
4.5 Servidor termina o atendimento do cliente 4
```



Uma representação alternativa para a ocupação e desocupação de recursos

A sequência de ocupação e desocupação do recurso pode ser representada de maneira mais compacta com o laço `with`:

```
def atendimentoServidor(env, nome, servidorRes):
    # função que ocupa o servidor e realiza o atendimento
    # solicita o recurso servidorRes
    with servidorRes.request() as request:
        # aguarda em fila até a liberação do recurso e o ocupa
        yield request
        print('%.1f Servidor inicia o atendimento do %s' % (env.now, nome))

        # aguarda um tempo de atendimento exponencialmente distribuído
        yield env.timeout(random.expovariate(1.0/TEMPO_MEDIO_ATENDIMENTO))
        print('%.1f Servidor termina o atendimento do %s' % (env.now, nome))
```

No script anterior a ocupação e desocupação é garantida dentro do `with`, deixando o código mais compacto e legível. Contudo, a aplicação é limitada a problemas de ocupação e desocupação simples de servidores (veja um contra-exemplo no Desafio 6).

Existem muitos conceitos a serem discutidos sobre os scripts anteriores e, garanto, que eles serão destrinchados nas seções seguintes.

Por hora, e para não esticar demais a atividade, analise atentamente os resultados da execução do script e avance para cima dos nossos desafios.

Conteúdos desta seção

Conteúdo	Descrição
`with servidorRes.request() as req:`	forma compacta de representar a sequência de ocupação e desocupação do recurso: `request(), yield` e `release().` Tudo que está dentro do with é realizado com o recurso ocupado

Desafios

> **Desafio 4:** para melhor compreensão do funcionamento do programa, imprima na tela o tempo de simulação e o números de clientes em fila. Quantos clientes existem em fila no instante 4.5?
>
> **Desafio 5:** calcule o tempo de permanência em fila de cada cliente e imprima o resultado na tela. Para isso, armazene o instante de chegada do cliente na fila em uma variável `chegada.` Ao final do atendimento, armazene o tempo de fila, numa variável `tempoFila` e apresente o resultado na tela.
>
> **Desafio 6:** um problema clássico de simulação envolve ocupar e desocupar recursos na seqüência correta. Considere uma lavanderia com 4 lavadoras, 3 secadoras e 5 cestos de roupas. Quando um cliente chega, ele coloca as roupas em uma máquina de lavar (ou aguarda em fila). A lavagem consome 20 minutos (constante). Ao terminar a lavagem, o cliente retira as roupas da máquina e coloca em um cesto e leva o cesto com suas roupas até a secadora, num processo que leva de 1 a 4 minutos distribuídos uniformemente. O cliente então descarrega as roupas do cesto diretamente para a secadora, espera a secagem e vai embora. Esse processo leva entre 9 e 12 minutos, uniformemente distribuídos. Construa um modelo de simulação que represente o processo anterior.

Solução dos desafios 4, 5 e 6

> **Desafio 4**: imprima na tela o tempo de simulação e o números de clientes em fila. Quantos clientes existem em fila no instante 4.5?

Para solução do desafio, basta lembrarmos que a qualquer momento, o conjunto de entidades em fila pelo recurso é dado por `servidorRes.queue` e, portanto, o número de entidade em fila é facilmente obtido pela expressão:

```
len(servidorRes.queue)
```

Foi acrescentada uma chamadas à função `print,` de modo a imprimir na tela o número de clientes em fila ao término do atendimento de cada cliente:

```
def atendimentoServidor(env, nome, servidorRes):
    # função que ocupa o servidor e realiza o atendimento
    # solicita o recurso servidorRes
    request = servidorRes.request()

    # aguarda em fila até a liberação do recurso e o ocupa
    yield request
    print('%.1f Servidor inicia o atendimento do %s' % (env.now, nome))

    # aguarda um tempo de atendimento exponencialmente distribuído
    yield env.timeout(random.expovariate(1.0/TEMPO_MEDIO_ATENDIMENTO))
    print('%.1f Servidor termina o atendimento do %s. Clientes em fila: %i'
            % (env.now, nome, len(servidorRes.queue)))

    # libera o recurso servidorRes
    yield servidorRes.release(request)
```

Executado o código, descobrimos que no instante 5,5 min, temos 2 clientes em fila:

```
0.5 Chegada do cliente 1
0.5 Servidor inicia o atendimento do cliente 1
1.4 Servidor termina o atendimento do cliente 1. Clientes em fila: 0
3.1 Chegada do cliente 2
3.1 Servidor inicia o atendimento do cliente 2
3.3 Chegada do cliente 3
4.1 Servidor termina o atendimento do cliente 2. Clientes em fila: 1
4.1 Servidor inicia o atendimento do cliente 3
4.1 Servidor termina o atendimento do cliente 3. Clientes em fila: 0
4.3 Chegada do cliente 4
4.3 Servidor inicia o atendimento do cliente 4
4.5 Servidor termina o atendimento do cliente 4. Clientes em fila: 0
```

Portanto, existem 0 cliente em fila no instante 4,5 minutos, nas condições simuladas (note a semente de geração de números aleatórios igual a 2).

> **Desafio 5**: calcule o tempo de permanência em fila de cada cliente e imprima o resultado na tela. Para isso, armazene o instante de chegada do cliente na fila em uma variável `chegada`. Ao final do atendimento, armazene o tempo de fila, numa variável `tempoFila` e apresente o resultado na tela.

A ideia deste desafio é que você se acostume com esse cálculo tão trivial quanto importante dentro da simulação: o tempo de permanência de uma entidade em algum local. Neste caso, o local é uma fila por ocupação de um recurso.
A lógica aqui é a de um cronometrista que deve disparar o cronômetro na chegada do cliente e pará-lo ao início do atendimento.
Assim, ao chegar, criamos uma variável `chegada` que armazena o instante atual fornecido pelo comando `env.now` do SimPy:

```
def atendimentoServidor(env, nome, servidorRes):
    # função que ocupa o servidor e realiza o atendimento
    # armazena o instante de chegada do cliente
    chegada = env.now
    # solicita o recurso servidorRes
    request = servidorRes.request()
```

Agora, inciado o atendimento (logo após o `yield` que ocupa o recurso), a variável `tempoFila` armazena o tempo de permanência em fila. Como num cronômetro, o tempo em fila é calculado pelo instante atual do cronômetro menos o instante de disparo dele já armazenado na variável `chegada`:

```
def atendimentoServidor(env, nome, servidorRes):
    # função que ocupa o servidor e realiza o atendimento
    # armazena o instante de chegada do cliente
    chegada = env.now
    # solicita o recurso servidorRes
    request = servidorRes.request()

    # aguarda em fila até a liberação do recurso e o ocupa
    yield request
    # calcula o tempo em fila
    tempoFila = env.now - chegada
```

Para imprimir o resultado, basta simplesmente alterar a chamada à função `print` na linha seguinte, de modo que o código final da função `atendimentoServidor` fica:

```
def atendimentoServidor(env, nome, servidorRes):
    # função que ocupa o servidor e realiza o atendimento
    # armazena o instante de chegada do cliente
    chegada = env.now
    # solicita o recurso servidorRes
    request = servidorRes.request()

    # aguarda em fila até a liberação do recurso e o ocupa
    yield request
    # calcula o tempo em fila
    tempoFila = env.now - chegada
    print('%.1f Servidor inicia o atendimento do %s. Tempo em fila: %.1f'
            % (env.now, nome, tempoFila))

    # aguarda um tempo de atendimento exponencialmente distribuído
    yield env.timeout(random.expovariate(1.0/TEMPO_MEDIO_ATENDIMENTO))
    print('%.1f Servidor termina o atendimento do %s. Clientes em fila: %i'
            % (env.now, nome, len(servidorRes.queue)))

    # libera o recurso servidorRes
    yield servidorRes.release(request)
```

Agora, a execução do programa mostra na tela o tempo de espera de cada cliente:

```
0.5 Chegada do cliente 1
0.5 Servidor inicia o atendimento do cliente 1. Tempo em fila: 0.0
1.4 Servidor termina o atendimento do cliente 1. Clientes em fila: 0
3.1 Chegada do cliente 2
3.1 Servidor inicia o atendimento do cliente 2. Tempo em fila: 0.0
3.3 Chegada do cliente 3
4.1 Servidor termina o atendimento do cliente 2. Clientes em fila: 1
4.1 Servidor inicia o atendimento do cliente 3. Tempo em fila: 0.8
4.1 Servidor termina o atendimento do cliente 3. Clientes em fila: 0
4.3 Chegada do cliente 4
4.3 Servidor inicia o atendimento do cliente 4. Tempo em fila: 0.0
4.5 Servidor termina o atendimento do cliente 4. Clientes em fila: 0
```

> **Desafio 6:** um problema clássico de simulação envolve ocupar e desocupar recursos na seqüência correta. Considere uma lavanderia com 4 lavadoras, 3 secadoras e 5 cestos de roupas. Quando um cliente chega, ele coloca as roupas em uma máquina de lavar (ou aguarda em fila). A lavagem consome 20 minutos (constante). Ao terminar a lavagem, o cliente retira as roupas da máquina e coloca em um cesto e leva o cesto com suas roupas até a secadora, num processo que leva de 1 a 4 minutos distribuídos uniformemente. O cliente então descarrega as roupas do cesto diretamente para a secadora, espera a secagem e vai embora. Esse processo leva entre 9 e 12 minutos, uniformemente distribuídos. Construa um modelo que represente o sistema descrito.

A dificuldade do desafio da lavanderia é representar corretamente a sequência de ocupação e desocupação dos recursos necessários de cada cliente. Se você ocupá-los/desocupá-los na ordem errada, fatalmente seu programa apresentará resultados inesperados.

Como se trata de um modelo com vários processos e distribuições, vamos seguir a Dica da seção "Solução dos desafios 2 e 3" e construir uma função para armazenar as distribuições do problema, organizando nosso código:

```
import random
import simpy

def distributions(tipo):
    # função que armazena as distribuições utilizadas no modelo
    return {
        'chegadas': random.expovariate(1.0/5.0),
        'lavar': 20,
        'carregar': random.uniform(1, 4),
        'descarregar': random.uniform(1, 2),
        'secar': random.uniform(9, 12),
    }.get(tipo, 0.0)
```

Como já destacado, a dificuldade é representar a sequência correta de processos do cliente: ele chega, ocupa uma lavadora, lava, ocupa um cesto, libera uma lavadora, ocupa uma secadora, libera o cesto, seca e libera a secadora. Se a sequência foi bem compreendida, a máscara a seguir será de fácil preenchimento:

```
import random
import simpy

contaClientes = 0             # conta clientes que chegaram no sistema

def distributions(tipo):
    # função que armazena as distribuições utilizadas no modelo
    return {
        'chegadas': random.expovariate(1.0/5.0),
        'lavar': 20,
        'carregar': random.uniform(1, 4),
        'descarregar': random.uniform(1, 2),
        'secar': random.uniform(9, 12),
    }.get(tipo, 0.0)

def chegadaClientes(env, lavadoras, cestos, secadoras):
    # função que gera a chegada de clientes
    global contaClientes

    pass

    # chamada do processo de lavagem e secagem
    pass

def lavaSeca(env, cliente, lavadoras, cestos, secadoras):
    # função que processa a operação de cada cliente dentro da lavanderia

    # ocupa a lavadora
    pass

    # antes de retirar da lavadora, pega um cesto
    pass

    # libera a lavadora, mas não o cesto
    pass

    # ocupa a secadora antes de liberar o cesto
    pass

    # libera o cesto mas não a secadora
    pass

    # pode liberar a secadora
    pass

random.seed(10)
env = simpy.Environment()
lavadoras = simpy.Resource(env, capacity = 3)
cestos = simpy.Resource(env, capacity = 2)
secadoras = simpy.Resource(env, capacity = 1)
env.process(chegadaClientes(env, lavadoras, cestos, secadoras))

env.run(until = 40)
```

O programa a seguir apresenta uma possível solução para o desafio, já com diversos comandos de impressão:

```
import random
import simpy

contaClientes = 0           # conta clientes que chegaram no sistema

def distributions(tipo):
    # função que armazena as distribuições utilizadas no modelo
    return {
        'chegadas': random.expovariate(1.0/5.0),
        'lavar': 20,
        'carregar': random.uniform(1, 4),
        'descarregar': random.uniform(1, 2),
        'secar': random.uniform(9, 12),
    }.get(tipo, 0.0)

def chegadaClientes(env, lavadoras, cestos, secadoras):
    # função que gera a chegada de clientes
    global contaClientes

    contaClientes = 0
    while True:
        contaClientes += 1
        yield env.timeout(distributions('chegadas'))
        print("%.1f chegada do cliente %s" %(env.now, contaClientes))
         # chamada do processo de lavagem e secagem
        env.process(lavaSeca(env, "Cliente %s" % contaClientes, lavadoras, cestos, secadoras))

def lavaSeca(env, cliente, lavadoras, cestos, secadoras):
    # função que processa a operação de cada cliente dentro da lavanderia
    global utilLavadora, tempoEsperaLavadora, contaLavadora

    # ocupa a lavadora
    req1 = lavadoras.request()
    yield req1
    print("%.1f %s ocupa lavadora" %(env.now, cliente))
    yield env.timeout(distributions('lavar'))

    # antes de retirar da lavadora, pega um cesto
    req2 = cestos.request()
    yield req2
    print("%.1f %s ocupa cesto" %(env.now, cliente))
    yield env.timeout(distributions('carregar'))

    # libera a lavadora, mas não o cesto
    lavadoras.release(req1)
    print("%.1f %s desocupa lavadora" %(env.now, cliente))

    # ocupa a secadora antes de liberar o cesto
    req3 = secadoras.request()
    yield req3
    print("%.1f %s ocupa secadora" %(env.now, cliente))
    yield env.timeout(distributions('descarregar'))

    # libera o cesto mas não a secadora
    cestos.release(req2)
    print("%.1f %s desocupa cesto" %(env.now, cliente))
    yield env.timeout(distributions('secar'))

    # pode liberar a secadora
```

```
    print("%.1f %s desocupa secadora" %(env.now, cliente))
    secadoras.release(req3)



random.seed(10)
env = simpy.Environment()
lavadoras = simpy.Resource(env, capacity=3)
cestos = simpy.Resource(env, capacity=2)
secadoras = simpy.Resource(env, capacity=1)
env.process(chegadaClientes(env, lavadoras, cestos, secadoras))

env.run(until=40)
```

A execução do programa anterior fornece como saída:

```
4.2 chegada do cliente 1
4.2 Cliente 1 ocupa lavadora
12.6 chegada do cliente 2
12.6 Cliente 2 ocupa lavadora
24.2 Cliente 1 ocupa cesto
27.2 Cliente 1 desocupa lavadora
27.2 Cliente 1 ocupa secadora
28.8 Cliente 1 desocupa cesto
32.6 Cliente 2 ocupa cesto
36.3 Cliente 2 desocupa lavadora
38.7 Cliente 1 desocupa secadora
38.7 Cliente 2 ocupa secadora
```

Teste seus conhecimentos:

1. A fila M/M/1 possui expressões analíticas conhecidas. Por exemplo, o tempo médio de permanência no sistema é dado pela expressão: $$W = \frac{1} {\mu - \lambda}$$. Valide seu modelo, ou seja, calcule o resultado esperado para a expressão e compare com o resultado obtido pelo seu programa.
2. Utilizando a função `plot` da bilbioteca `matplotlib,` .
3. No problema da lavanderia, crie uma situação de desistência, isto é: caso a fila de espera por lavadoras seja de 5 clientes, o próximo cliente a chegar no sistema desiste imediatamente de entrar na lavanderia.

Atributos e variáveis: diferenças em SimPy

Qual a diferença entre atributo e variável para um modelo de simulação? O atributo pertence à entidade, enquanto a variável pertence ao modelo. De outro modo, se um cliente chega a uma loja e compra 1, 2 ou 3 produtos, esse cliente possui um **atributo** imediato: o **número de produtos** comprados. Note que o atributo "número de produtos" é um valor diferente para cada cliente, ou seja: é um valor exclusivo do cliente.

Por outo lado, um parâmetro de saída importante seria o número total de produtos vendidos nesta loja ao longo da duração da simulação. O total de produtos é a soma dos atributos "número de produtos" de cada cliente que comprou algo na loja. Assim, o total vendido é uma **variável** do modelo, que se acumula a cada nova compra, independentemente de quem é o cliente.

Em SimPy a coisa é trivial: toda variável **local** funciona como atributo da entidade gerada e toda variável **global** é naturalmente uma variável do modelo. Não se trata de uma regra absoluta, nem tampouco foi imaginada pelos desenvolvedores da biblioteca, é decorrente da necessidade de se representar os processos do modelo de simulação por meio de **funções** que, por sua vez representam entidades executando alguma coisa.

Usuários de pacotes comerciais (Simul8, Anylogic, GPSS, Arena etc.) estão acostumados a informar explicitamente ao modelo o que é atributo e o que é variável. Em SimPy, basta lembrar que as variáveis globais serão variáveis de todo o modelo e que os atributos de interesse devem ser transferidos de um processo ao outro por transferência de argumentos no cabeçalho das funções.

Voltemos ao exemplo de chegadas de clientes numa loja. Queremos que cada cliente tenha como atributo o número de produtos desejados:

```
import random    # gerador de números aleatórios
import simpy     # biblioteca de simulação

contaVendas = 0  # variável global que manrca o número de vendas realizadas

def geraChegadas(env):
    # função que cria chegadas de entidades no sistema
    # variável local = atributo da entidade
    contaEntidade = 0
    while True:
        yield env.timeout(1)
        contaEntidade += 1
        # atributo do cliente: número de produtos desejados
        produtos = random.randint(1,3)
        print("%.1f Chegada do cliente %i\tProdutos desejados: %d"
                % (env.now, contaEntidade, produtos))

        # inicia o processo de atendimento do cliente de atributos contaEntidade
        # e do número de produtos
        env.process(compra(env, "cliente %d" % contaEntidade, produtos))

def compra(env, nome, produtos):
    # função que realiza a venda para as entidades
    # nome e produtos, são atributo da entidade

    global contaVendas # variável global = variável do modelo

    for i in range(0, produtos):
        yield env.timeout(2)
        contaVendas += produtos
        print("%.1f Compra do %s \tProdutos comprados: %d" % (env.now, nome, produtos))

random.seed(1000)               # semente do gerador de números aleatórios
env = simpy.Environment()       # cria o environment do modelo
env.process(geraChegadas(env))  # cria o processo de chegadas

env.run(until=5)                # roda a simulação por 10 unidades de tempo
print("\nTotal vendido: %d produtos" % contaVendas)
```

A execução do programa por apenas 5 minutos, apresenta como resposta:

```
1.0 Chegada do cliente 1        Produtos desejados: 2
2.0 Chegada do cliente 2        Produtos desejados: 3
3.0 Compra do cliente 1         Produtos comprados: 2
3.0 Chegada do cliente 3        Produtos desejados: 1
4.0 Compra do cliente 2         Produtos comprados: 3
4.0 Chegada do cliente 4        Produtos desejados: 2

Total vendido: 5 produtos
```

É importante destacar no exemplo, que o cliente (ou entidade) gerado(a) pela função `geraChegadas` é enviado(a) para a função `compra` com seu atributo `produtos,` como se nota na linha em que o cliente chama o processo de compra:

```
env.process(compra(env, "cliente %d" % contaEntidade, produtos))
```

Agora raciocine de modo inverso: seria possível representar o número total de produtos vendidos como uma variável local? Intuitivamente, somos levados a refletir na possibilidade de *transferir* o número total de produtos como uma parâmetro de chamada da função. Mas, reflita mais um tiquinho... É possível passar o total vendido como um parâmetro de chamada da função?

Do modo como o problema foi modelado, isso não é possível, pois cada chegada gera um novo processo `compra` independente para cada cliente e não há como transferir tal valor de uma chamada do processo para outra. A seção a seguir, apresenta uma alternativa interessante que evita o uso de variáveis globais num modelo de simulação.

Atributos em modelos orientados ao objeto

Para aqueles que programam com classes e objetos, o atributo é naturalmente o atributo da entidade (ou do processo). Uma facilidade que a programação voltada ao objeto possui é que podemos criar atributos para recursos também. Neste caso, basta que o recurso seja criado dentro de uma classe.

Por exemplo, a fila M\/M\/1 poderia ser modelada por uma classe `Servidor`, em que um dos seus atributos é o próprio `Resource` do SimPy, como mostra o código a seguir:

```
import random
import simpy

class Servidor(object):
    # cria a classe Servidor
    # note que um dos atributos é o próprio Recurso do simpy
    def __init__(self, env, capacidade, duracao):
        # atributos do recurso
        self.env = env
        self.res = simpy.Resource(env, capacity=capacidade)
        self.taxaExpo = 1.0/duracao
        self.clientesAtendidos = 0

    def atendimento(self, cliente):
        # executa o atendimento
        print("%.1f Início do atendimento do %s" % (env.now, cliente))
        yield self.env.timeout(random.expovariate(self.taxaExpo))
        print("%.1f Fim do atendimento do %s" % (env.now, cliente))

def processaCliente(env, cliente, servidor):
    # função que processa o cliente

    print('%.1f Chegada do %s' % (env.now, cliente))
    with servidor.res.request() as req: # note que o Resource é um atributo também
        yield req

        print('%.1f Servidor ocupado pelo %s' % (env.now, cliente))
        yield env.process(servidor.atendimento(cliente))
        self.clientesAtendidos += 1
        print('%.1f Servidor desocupado pelo %s' % (env.now, cliente))


def geraClientes(env, intervalo, servidor):
    # função que gera os clientes
    i = 0
    while True:
        yield env.timeout(random.expovariate(1.0/intervalo))
        i += 1
        env.process(processaCliente(env, 'cliente %d' % i, servidor))


random.seed(1000)

env = simpy.Environment()
# cria o objeto servidor (que é um recurso)
servidor = Servidor(env, 1, 1)
env.process(geraClientes(env, 3, servidor))

env.run(until=5)
```

Quando processado por apenas 5 minutos, o modelo anterior fornece:

```
4.5 Chegada do cliente 1
4.5 Servidor ocupado pelo cliente 1
4.5 Início do atendimento do cliente 1
4.6 Fim do atendimento do cliente 1
4.6 Servidor desocupado pelo cliente 1
```

No caso da programação voltada ao objeto, uma variável do modelo pode pertencer a uma classe, sem a necessidade de que a variável seja global. Por exemplo, o atributo `clientesAtendidos` da classe `Servidor` é uma variável que representa o total de cliente atendidos ao longo da simulação. Caso a representação utilizada não fosse voltada ao objeto, o número de clientes atendidos seria forçosamente uma variável global.

Conteúdos desta seção

Conteúdo	Descrição
representação de atributos	os atributos devem ser representados localmente e transferidos entre funções (ou processos) como parâmetros das funções (ou processos)
representação de variáveis	as variáveis do modelo são naturalmente representadas como variáveis globais ou, no caso da programação voltada ao objeto, como atributos de classes.

Desafios

> **Desafio 7**: retome o problema da lavanderia (Desafio 6). Estime o tempo médio que os clientes atendidos aguardaram pela lavadora. Dica: você precisará de uma variável global para o cálculo do tempo de espera e um atributo para marcar a hora de chegada no sistema.
>
> **Desafior 8**: no desafio anterior, caso você simule por 10 ou mais horas, deve notar como o tempo de espera pela lavadora fica muito alto. Para identificar o gargalo do sistema, acrescente a impressão do número de clientes que ficaram em fila ao final da simulação. Você consegue otimizar o sistema a partir do modelo construído?

Solução dos desafios 7 e 8

> **Desafio 7**: retome o problema da lavanderia (Desafio 6). Estime o tempo médio que os clientes atendidos aguardaram pela lavadora.
>
> Dica: você precisará de uma variável global para o cálculo do tempo de espera e um atributo para marcar a hora de chegada do cliente na lavadora.

O tempo médio de espera por fila de um recurso - no caso lavadoras - é estimado pelo soma do tempo que todos os clientes aguardaram pelo recurso, dividido pelo número de clientes que ocuparam o recurso ao longo da simulação.

Assim, vamos criar duas variáveis globais para armazenar a soma do tempo de espera por lavadora de todos os clientes que ocuparam as lavadoras, bem como o número de clientes que ocuparam as mesmas lavadoras:

```
import random
import simpy

contaClientes = 0           # conta clientes que chegaram no sistema
tempoEsperaLavadora = 0     # conta tempo de espera total por lavadora
contaLavadora = 0           # conta clientes que ocuparam uma lavadora
```

A seguir, precisamos alterar a função `lavaSeca` para calcular corretamente o tempo de espera por lavadora de cada cliente, somar este valor à variável global `tempoEsperaLavadora` e incrementar o número de clientes que ocuparam lavadoras na variável global `contaLavadora` (representação apenas da parte do código que é alterada):

```
def lavaSeca(env, cliente, lavadoras, cestos, secadoras):
    # função que processa a operação de cada cliente dentro da lavanderia
    global tempoEsperaLavadora, contaLavadora

    # marca atributo chegada com o tempo atual de chegada da entidade
    chegada = env.now
    # ocupa a lavadora
    req1 = lavadoras.request()
    yield req1
    # incrementa lavadoras ocupadas
    contaLavadora += 1
    # calcula o tempo de espera em fila por lavadora
    tempoFilaLavadora = env.now - chegada
    tempoEsperaLavadora += tempoFilaLavadora
    print("%4.1f %s ocupa lavadora" %(env.now, cliente))
    yield env.timeout(distributions('lavar'))
```

Ao final do programa, basta acrescentar uma linha para imprimir o tempo médio em fila de espera por lavadoras e o número de vezes que uma lavadora foi ocupada ao longo da simulação:

```
random.seed(10)
env = simpy.Environment()
lavadoras = simpy.Resource(env, capacity=3)
cestos = simpy.Resource(env, capacity=2)
secadoras = simpy.Resource(env, capacity=1)
env.process(chegadaClientes(env, lavadoras, cestos, secadoras))

env.run(until=40)

print("\nTempo médio de espera por lavadoras: %.2f min. Clientes atendidos: %i"
        %(tempoEsperaLavadora/contaLavadora, contaLavadora))
```

Quando executado por 40 minutos, o modelo completo com as alterações anteriores fornece como saída:

```
 4.2 Chegada do cliente 1
 4.2 Cliente 1 ocupa lavadora
12.6 Chegada do cliente 2
12.6 Cliente 2 ocupa lavadora
24.2 Cliente 1 ocupa cesto
27.2 Cliente 1 desocupa lavadora
27.2 Cliente 1 ocupa secadora
28.8 Cliente 1 desocupa cesto
32.6 Cliente 2 ocupa cesto
36.3 Cliente 2 desocupa lavadora
38.7 Cliente 1 desocupa secadora
38.7 Cliente 2 ocupa secadora

Tempo médio de espera por lavadoras: 0.00 min. Clientes atendidos: 2
```

> **Desafio 8**: no desafio anterior, caso você simule por 10 ou mais horas, deve notar como o tempo de espera pela lavadora fica muito alto. Para identificar o gargalo do sistema, acrescente a impressão do número de clientes que ficaram em fila ao final da simulação. Você consegue otimizar o sistema a partir do modelo construído?

Quando simulamos o sistema por 10 horas (=10*60 minutos), obtemos como resposta:

```
...
599.6 Cliente 75 ocupa cesto
600.0 Chegada do cliente 133

Tempo médio de espera por lavadoras: 138.63 min. Clientes atendidos: 77
```

Para a solução do desafio, basta acrescentar uma linha ao final do programa principal que imprime as filas de por recursos (lavadoras, cestos e secadoras) ao final da simulação:

```
env.run(until=600)

print("\nTempo médio de espera por lavadoras: %.2f min. Clientes atendidos: %i"
        %(tempoEsperaLavadora/contaLavadora, contaLavadora))
print("Fila de clientes ao final da simulação: lavadoras %i cestos %i secadoras %i"
        %(len(lavadoras.queue), len(cestos.queue), len(secadoras.queue)))
```

Quando simulado por 600 minutos (ou 10 horas), a saída do modelo fornece:

```
...
599.6 Cliente 75 ocupa cesto
600.0 Chegada do cliente 133

Tempo médio de espera por lavadoras: 138.63 min. Clientes atendidos: 77
Fila de clientes ao final da simulação: lavadoras 56 cestos 0 secadoras 0
```

Portanto, ao final da simulação, existem 56 clientes aguardando uma lavadora livre, enquanto nenhum cliente aguarda por cestos ou secadoras. Temos um caso clássico de fila **infinita**, isto é: a taxa de horária de atendimento das lavadoras é menor que a taxa horária com que os clientes chegam à lavanderia. Assim, se 1 cliente ocupa em média 20 minutos uma lavadora, a taxa de atendimento em cada lavadora é de $$\mu=$$ 0.05 clientes\/min (=1 cliente \/20 min), enquanto a taxa de chegadas de clientes na lavandeira é de $$\lambda=$$0.20 clientes\/min (= 1 cliente\/5 min).

Como a taxa de atendimento é menor que a taxa de chegadas, a fila cresce indefinidamente. Para termos um sistema equilibrado, precisaríamos de um número de lavadoras tal que se garanta que a taxa de atendimento da soma das lavadoras seja maior que a taxa de chegadas de clientes no sistema ou:

$$\rho=\frac{\lambda}{n*\mu} < 1 \to n > \frac{\lambda}{\mu} = \frac{0.20}{0.05}=4$$

Portanto, com 5 (ou mais) lavadoras eliminaríamos **o gargalo na lavagem**.

O bom da simulação é que podemos testar se a calculera anterior faz sentido. Quando simulado para 5 lavadoras, o modelo fornece como saída:

```
...
598.8 Cliente 107 ocupa cesto
599.0 Cliente 105 desocupa secadora

Tempo médio de espera por lavadoras: 4.40 min. Clientes atendidos: 108
Fila de clientes ao final da simulação: lavadoras 0 cestos 0 secadoras 0
```

Com 5 lavadoras, portanto, já não existe fila residual.

Teste seus conhecimentos

1. Elabore uma pequena rotina capaz de simule o sistema para números diferentes de recursos (será que o número de cestos e secadoras não está exagerado também?). Manipule o modelo para encontrar o número mínimo de recursos necessário, de modo a não haver gargalos no sistema.

Solução dos desafios 9 e 10

> **Desafio 9**: Considere que cada entidade gerada no primeiro exemplo desta seção tem um peso em gramas dado por uma distribuição normal de média 10 e desvio padrão igual a 3. Crie um critério de parada para quando a média dos pesos das entidades geradas esteja no intervalo entre 9,5 e 10,5.

Este primeiro desafio envolve poucas modificações no programa original. Acrescentamos três variáveis novas: `media,` `contador` e `pesoTotal;` o laço `while` foi substituído pelo critério de parada e algumas linhas foram acrescentadas para o cálculo da média de peso até a última entidade gerada. O peso de cada entidade é sorteado pela função `random.normalvariate(mu, sigma)` da biblioteca `random.`

```
import simpy
import random

def geraChegada(env, numEntidades):
    media, contador, pesoTotal = 0, 0, 0

    while media > 10.5 or media < 9.5:      # critério de parada
        yield env.timeout(1)
        contador += 1                       # conta entidades geradas
        peso = random.normalvariate(10, 3)  # sorteia o peso da entidade
        pesoTotal += peso                   # acumula o peso total até agora
        media = pesoTotal/contador          # calcula média dos pesos
        print("%4.1f nova chegada\tPeso: %4.1f kg\tMédia atual: %4.1f"
                %(env.now, peso, media))


random.seed(100)
env = simpy.Environment()
chegadas = env.process(geraChegada(env, 5)) # gere apenas 5 entidades
env.run()                                   # executa até o fim de todos os processos
```

Quando executado, o modelo anterior apresenta como resultado:

```
 1.0 nova chegada       Peso:  6.7 kg   Média atual:  6.7
 2.0 nova chegada       Peso: 14.7 kg   Média atual: 10.7
 3.0 nova chegada       Peso: 12.1 kg   Média atual: 11.2
 4.0 nova chegada       Peso: 13.3 kg   Média atual: 11.7
 5.0 nova chegada       Peso:  6.0 kg   Média atual: 10.6
 6.0 nova chegada       Peso: 13.5 kg   Média atual: 11.0
 7.0 nova chegada       Peso:  5.8 kg   Média atual: 10.3
```

> **Desafio 10**: Modifique o critério anterior para que a parada ocorra quando a média for 10 kg, com um intervalo de confiança de amplitude 0,5 e nível de significância igual a 95%. Dica: utilize a biblioteca `numpy` para isso (consulte o Stack Overflow.

Esta situação exige um pouco mais no processo de codificação, contudo é algo muito utilizado em modelos de simulação de eventos discretos.

Como agora queremos o Intervalo de Confiança de uma dada amostra, os valores dos pesos serão armazenados em uma lista (`pesosList` , no caso do desafio).

A biblioteca *numpy* fornece um meio fácil de se estimar a média e o desvio padrão de uma amostra de valores armazenada numa lista:

- `numpy.mean(pesosList)` : estima a média da lista `pesosList` ;
- `numpy.std(pesosList)` : estima o desvio-padrão da lista `pesosList`

Para o cálculo do intervalo de confiança, devemos lembrar que, para amostras pequenas, a sua expressão é dada por:

$$ [\bar{x}- t*{1-\alpha , n-1} \frac{s}{\sqrt{n}} , \bar{x} + t*{1-\alpha , n-1} \frac{s}{\sqrt{n}}]

$$

A biblioteca *scipy.stats* possui diversas funções estatísticas, dentre elas, a distribuição t de student, necessária para o cálculo do intervalo de confiança. Como está será uma operação rotineira nos nossos modelos de simulação, o ideal é encapsular o código em uma função específica:

```
def intervaloConfMedia(a, conf=0.95):
    # retorna a média e a amplitude do intervalo de confiança dos valores contidos em a
    media, sem = numpy.mean(a), scipy.stats.sem(a)
    m = scipy.stats.t.ppf((1+conf)/2., len(a)-1)
    h = m*sem
    return media, h
```

A função anterior calcula a média e amplitude de um intervalo de confiança, a partir da lista de valores e do nível de confiança desejado.

O novo programa então ficaria:

```
import simpy
import random
import numpy        # biblioteca numpy para computação científica http://www.numpy.org/
import scipy.stats  # bilbioteca scipy.stats de funções estatísticas

def intervaloConfMedia(a, conf=0.95):
    # retorna a média e a amplitude do intervalo de confiança dos valores contidos em a
    media, sem = numpy.mean(a), scipy.stats.sem(a)
    m = scipy.stats.t.ppf((1+conf)/2., len(a)-1)
    return media, h

def geraChegada(env):
    pesosList = []                          # lista para armazenar os valores de pesos gerados

    while True:
        yield env.timeout(1)
        # adiciona à lista o peso da entidade atual
        pesosList.append(random.normalvariate(10, 5))

        # calcula a amplitude do intervalo de confiança, com nível de significância = 95%
        if len(pesosList) > 1:
            media, amplitude = intervaloConfMedia(pesosList, 0.95)
            print("%4.1f Média atual: %.2f kg\tAmplitude atual: %.2f kg"
                    %(env.now, media, amplitude))

            # se a amplitude atende ao critério estabelecido, interronpe o processo
            if amplitude < 0.5:
                print("\n%4.1f Intervalo de confiança atingido após %s valores! [%.2f, %.2f]"
                    % (env.now, len(pesosList), media-amplitude, media+amplitude))
                #termina o laço while
                break


random.seed(100)
env = simpy.Environment()
chegadas = env.process(geraChegada(env))
env.run()                                   # executa até o fim de todos os processos
```

O programa anterior leva 411 amostras para atingir o intervalo desejado:

```
...
410.0 Média atual: 10.17 kg     Amplitude atual: 0.50 kg
411.0 Média atual: 10.18 kg     Amplitude atual: 0.50 kg

411.0 Intervalo de confiança atingido após 411 valores! [9.68, 10.68]
```

Existem diversas maneiras de se estimar o intervalo de confiança utilizando-se as bibliotecas do Python. A maneira aqui proposta se baseia no `numpy` e no `scipy.stats`. Eventualmente tais bibliotecas não estejam instaladas na seu ambiente Python e eu antecipo: isso pode ser um baita problema para você =(

A questão aqui é que os modelos de simulação usualmente têm grande demanda por processamento estatístico de valores durante ou mesmo ao final da simulação. A biblioteca `numpy` facilita bastante esta tarefa, principalmente quando se considera o suporte dado pelos usuários do Stack Overflow .

Como sugestão, habitue-se a construir funções padronizadas para monitoramento e cálculos estatísticos, de modo que você pode reaproveitá-las em novos programas. Em algum momento, inclusive, você pode criar sua própria biblioteca de funções para análise de saída de modelos de simulação e compartilhar com a comunidade de software livre.

Outros tipos de recursos: com prioridade e preemptivos

Além do recurso como definido nas seções anteriores, o SimPy possui dois tipos específicos de recursos: com prioridade e "peemptivos".

Recursos com prioridade: `PriorityResource`

Um recurso pode ter uma fila de entidades desejando ocupá-lo para executar determinado processo. Existindo a fila, o recurso será ocupado respeitando a ordem de chegada das entidades (ou a regra FIFO).

Contudo, existem situações em que algumas entidades possuem *prioridades* sobre as outras, de modo que elas desrespeitam a regra do primeiro a chegar é o primeiro a ser atendido.

Por exemplo, considere um consultório de pronto atendimento de um hospital em que 70% do pacientes são de prioridade baixa (pulseira verde), 20% de prioridade intermediária (pulseira amarela) e 10% de prioridade alta (pulseira vermelha). Existem 2 médicos que realizam o atendimento e que sempre verificam inicialmente a ordem de prioridade dos pacientes na fila. Os pacientes chegam entre si em intervalos exponencialmente distribuídos, com média de 5 minutos e o atendimento é também exponencialmente distribuído, com média de 9 minutos por paciente.

No exemplo, os médicos são recursos, mas também respeitam uma regra específica de prioridade. Um médico ou recurso deste tipo, é criado pelo comando:

```
medicos = simpy.PriorityResource(env, capacity=capacidade_desejada)
```

Para a solução do exemplo, o modelo aqui proposto terá 3 funções: uma para sorteio do tipo de pulseira, uma para geração de chegadas de pacientes e outra para atendimento dos pacientes.

Como uma máscara inicial do modelo, teríamos:

```
import simpy
import random

def sorteiaPulseira():
    # retorna a cor da pulseira e sua prioridade
    pass

def chegadaPacientes(env, medicos):
    # gera pacientes exponencialmente distribuídos
    # sorteia a pulseira
    # inicia processo de atendimento
    pass

def atendimento(env, paciente, pulseira, prio, medicos):
    # ocupa um médico e realiza o atendimento do paciente
    pass

random.seed(100)
env = simpy.Environment()
medicos = simpy.PriorityResource(env, capacity=2) # cria os 2 médicos
chegadas = env.process(chegadaPacientes(env, medicos))

env.run(until=20)
```

O preenchimento da máscara pode ser feito de diversas maneiras, um possibilidade seria:

```
import simpy
import random

def sorteiaPulseira():
    # retorna a cor da pulseira e sua prioridade
    r = random.random()                 # sorteia número aleatório ente 0 e 1
    if r <= .70:                        # 70% é pulseira verde
        return "pulseira verde", 3
    elif r <= .90:                      # 20% (=90-70) é pulseira amarela
        return "pulseira amarela", 2
    return "pulseira vermelha", 1       # 10% (=100-90) é pulseira vermelha

def chegadaPacientes(env, medicos):
    #gera pacientes exponencialmente distribuídos

    i = 0
    while True:
        yield env.timeout(random.expovariate(1/5))
        i += 1

        # sorteia a pulseira
        pulseira, prio = sorteiaPulseira()
        print("%4.1f Paciente %2i com %s chega" %(env.now, i, pulseira))

        # inicia processo de atendimento
        env.process(atendimento(env, "Paciente %2i" %i, pulseira, prio, medicos))

def atendimento(env, paciente, pulseira, prio, medicos):
    # ocupa um médico e realiza o atendimento do paciente

    with medicos.request(priority=prio) as req:
        yield req
        print("%4.1f %s com %s inicia o atendimento" %(env.now, paciente, pulseira))
        yield env.timeout(random.expovariate(1/9))
        print("%4.1f %s com %s termina o atendimento" %(env.now, paciente, pulseira))

random.seed(100)
env = simpy.Environment()
# cria os médicos
medicos = simpy.PriorityResource(env, capacity=2)
chegadas = env.process(chegadaPacientes(env, medicos))

env.run(until=20)
```

O importante a ser destacado é que a prioridade é informada ao `request` do recurso `medicos` pelo argumento `priority:`

```
with medicos.request(priority=prio) as req:
    yield req
```

Para o SimPy, **quando menor o valor fornecido** para o parâmetro `priority,` **maior a prioridade** daquela entidade na fila. Assim, a função `sorteiaPulseira` retorna 3 para a pulseira verde (de menor prioridade) e 1 para a vermelha (de maior prioridade).

Quando o modelo anterior é executado, fornece como saída:

```
 0.8 Paciente  1 com pulseira verde chega
 0.8 Paciente  1 com pulseira verde inicia o atendimento
 8.2 Paciente  2 com pulseira amarela chega
 8.2 Paciente  2 com pulseira amarela inicia o atendimento
11.0 Paciente  3 com pulseira verde chega
11.4 Paciente  4 com pulseira verde chega
11.7 Paciente  5 com pulseira vermelha chega
11.8 Paciente  1 com pulseira verde termina o atendimento
11.8 Paciente  5 com pulseira vermelha inicia o atendimento
15.5 Paciente  5 com pulseira vermelha termina o atendimento
15.5 Paciente  3 com pulseira verde inicia o atendimento
18.8 Paciente  3 com pulseira verde termina o atendimento
18.8 Paciente  4 com pulseira verde inicia o atendimento
```

Percebemos que o paciente 5 chegou no instante 11,7 minutos, depois do pacientes 3 e 4, mas iniciou seu atendimento assim que um médico ficou livre no instante 11,8 minutos (exatamente aquele que atendia ao Paciente 1).

Recursos que podem ser interrompidos: `PreemptiveResource`

Considere, no exemplo anterior, que o paciente de pulseira vermelha tem uma prioridade tal que ele interrompe o atendimento atual do médico e imediatamente é atendido. Os recursos com preemptividade são recursos que aceitam a interrupção da tarefa em execução para iniciar outra de maior prioridade.

Um recurso capaz de ser interrompido é criado pelo comando:

```
medicos = simpy.PreemptiveResource(env, capacity=capacidade)
```

Assim, o modelo anterior precisa ser modificado de modo a criar os médicos corretamente:

```
random.seed(100)
env = simpy.Environment()
# cria os médicos
medicos = simpy.PreemptiveResource(env, capacity=2)
chegadas = env.process(chegadaPacientes(env, medicos))

env.run(until=20)
```

Agora, devemos modificar a função `atendimento` para garantir que quando um recurso for requisitado por um processo de menor prioridade, ele causará uma interrupção no Python, o que obriga a utilização de bloco de controle de interrupção `try:...except`.

Quando um recurso deve ser interrompido, o SimPy retorna um interrupção do tipo `simpy.Interrupt,` como mostrado no código a seguir (noteo bloco `try...except` dentro da função atendimento):

```
def atendimento(env, paciente, pulseira, prio, medicos):
    # ocupa um médico e realiza o atendimento do paciente

    with medicos.request(priority=prio) as req:
        yield req
        print("%4.1f %s com %s inicia o atendimento" %(env.now, paciente, pulseira))
        try:
            yield env.timeout(random.expovariate(1/9))
            print("%4.1f %s com %s termina o atendimento" %(env.now, paciente, pulseira))
        except:
            print("%4.1f %s com %s tem atendimento interrompido" %(env.now, paciente, pulseira
))

random.seed(100)
env = simpy.Environment()
# cria os médicos
medicos = simpy.PreemptiveResource(env, capacity=2)
chegadas = env.process(chegadaPacientes(env, medicos))

env.run(until=20)
```

Quando simulado por apenas 20 minutos, o modelo acrescido das correções apresentadas fornece a seguinte saída:

```
 0.8 Paciente  1 com pulseira verde chega
 0.8 Paciente  1 com pulseira verde inicia o atendimento
 8.2 Paciente  2 com pulseira amarela chega
 8.2 Paciente  2 com pulseira amarela inicia o atendimento
11.0 Paciente  3 com pulseira verde chega
11.4 Paciente  4 com pulseira verde chega
11.7 Paciente  5 com pulseira vermelha chega
11.7 Paciente  1 com pulseira verde tem atendimento interrompido
11.7 Paciente  5 com pulseira vermelha inicia o atendimento
15.3 Paciente  5 com pulseira vermelha termina o atendimento
15.3 Paciente  3 com pulseira verde inicia o atendimento
18.7 Paciente  3 com pulseira verde termina o atendimento
18.7 Paciente  4 com pulseira verde inicia o atendimento
```

Note que o Paciente 5 interrompe o atendimento do Paciente 1, como desejado.

Contudo, a implementação anterior está cheia de limitações: pacientes com pulseira amarela não deveriam interromper o atendimento, mas na implementação proposta eles devem interromper o atendimento de pacientes de pulseira verde. Para estas situações, o `request` possui um argumento `preempt` que permite ligar ou desligar a opção de preemptividade:

```
with medicos.request(priority=prio, preempt=preempt) as req:
    yield req
```

O modelo alterado para interromper apenas no caso de pulseiras vermelhas, ficaria (note que o argumento `preempt` é agora fornecido diretamente a partir da função `sorteiaPulseira):`

```
import simpy
import random

def sorteiaPulseira():
    # retorna a cor da pulseira e sua prioridade
    r = random.random() # sorteia número aleatório ente 0 e 1
    if r <= .70: # 70% é pulseira verde
        return "pulseira verde", 3, False
    elif r <= .90: # 20% (=90-70) é pulseira amarela
        return "pulseira amarela", 2, False
    return "pulseira vermelha", 1, True # 10% (=100-90) é pulseira vermelha

def chegadaPacientes(env, medicos):
    #gera pacientes exponencialmente distribuídos

    i = 0
    while True:
        yield env.timeout(random.expovariate(1/5))
        i += 1

        # sorteia a pulseira
        pulseira, prio, preempt = sorteiaPulseira()
        print("%4.1f Paciente %2i com %s chega" %(env.now, i, pulseira))

        # inicia processo de atendimento
        env.process(atendimento(env, "Paciente %2i" %i, pulseira, prio, preempt, medicos))

def atendimento(env, paciente, pulseira, prio, preempt, medicos):
    # ocupa um médico e realiza o atendimento do paciente

    with medicos.request(priority=prio, preempt=preempt) as req:
        yield req
        print("%4.1f %s com %s inicia o atendimento" %(env.now, paciente, pulseira))
        try:
            yield env.timeout(random.expovariate(1/9))
            print("%4.1f %s com %s termina o atendimento" %(env.now, paciente, pulseira))
        except:
            print("%4.1f %s com %s tem atendimento interrompido" %(env.now, paciente, pulseira
))

random.seed(100)
env = simpy.Environment()
# cria os médicos
medicos = simpy.PreemptiveResource(env, capacity=2)
chegadas = env.process(chegadaPacientes(env, medicos))

env.run(until=20)
```

O modelo anterior, quando executado por apenas 20 minutos, fornece como saída:

```
 0.8 Paciente  1 com pulseira verde chega
 0.8 Paciente  1 com pulseira verde inicia o atendimento
 8.2 Paciente  2 com pulseira amarela chega
 8.2 Paciente  2 com pulseira amarela inicia o atendimento
11.0 Paciente  3 com pulseira verde chega
11.4 Paciente  4 com pulseira verde chega
11.7 Paciente  5 com pulseira vermelha chega
11.7 Paciente  1 com pulseira verde tem atendimento interrompido
11.7 Paciente  5 com pulseira vermelha inicia o atendimento
15.3 Paciente  5 com pulseira vermelha termina o atendimento
15.3 Paciente  3 com pulseira verde inicia o atendimento
18.7 Paciente  3 com pulseira verde termina o atendimento
18.7 Paciente  4 com pulseira verde inicia o atendimento
```

Conteúdos desta seção

Conteúdo	Descrição
`meuRecurso = simpy.PriorityResource(env, capacity=1)`	Cria um recurso com prioridade e capacidade = 1
`meuRequest = meuRecurso.request(env, priority=prio)`	Solicita o recurso meuRecurso (note que ele ainda não ocupa o recurso) respeitando a ordem de prioridade primeiro e a regra FIFO a seguir
`meuRecursoPreempt = simpy.PreemptiveResource(env, capacity=1)`	Cria um recurso em `env` que pode ser interrompido por entidades de prioridade maior
`meuRequest = meuRecursoPreempt.request(env, priority=prio, preempt=preempt)`	Solicita o recurso meuRecurso (note que ele ainda não ocupa o recurso) respeitando a ordem de prioridade primeiro e a regra FIFO a seguir. Caso preempt seja False o o recurso não é interrompido
`try:...except simpy.Interrupt:`	Chamada de interrupção utilizada na lógica try:...except:

Desafios

> **Desafio 11**: acrescente ao último programa proposto o cálculo do tempo de atendimento que ainda falta de atendimento para o paciente que foi interrompido por outro e imprima o resultado na tela.
>
> **Desafio 12**: quando um paciente é interrompido, ele deseja retornar ao atendimento de onde parou. Altere o programa para que um paciente de pulseira verde interrompido possa retornar para ser atendido no tempo restante do seu atendimento. Dica: altere a numeração de prioridades de modo que um paciente verde interrompido tenha prioridade superior ao de um paciente verde que acabou de chegar.

Solução dos desafios 11 e 12

> **Desafio 11**: acrescente ao último programa proposto o cálculo do tempo de atendimento que ainda falta para o paciente que foi interrompido por outro e imprima o resultado na tela.

Neste caso, precisamos acrescentar o cálculo do tempo faltante para o paciente na função `atendimento:`

```
def atendimento(env, paciente, pulseira, prio, preempt, medicos):
    # ocupa um médico e realiza o atendimento do paciente

    with medicos.request(priority=prio, preempt=preempt) as req:
        yield req
        inicioAtendimento = env.now
        print("%4.1f %s com %s inicia o atendimento" %(env.now, paciente, pulseira))
        try:
            # sorteia o tempo de atendimento
            tempoAtendimento = random.expovariate(1/9)
            yield env.timeout(tempoAtendimento)
            print("%4.1f %s com %s termina o atendimento"
                %(env.now, paciente, pulseira))
        except simpy.Interrupt:
            # recalcula o tempo de atendimento
            tempoAtendimento -= env.now-inicioAtendimento
            print("%4.1f %s com %s tem atendimento interrompido"
                    %(env.now, paciente, pulseira))
            print("%4.1f %s ainda precisa de %4.1f min de atendimento"
                    %(env.now, paciente, tempoAtendimento))
```

Quando o modelo é executado por apenas 20 minutos, com a alteração apresentada da função `atendimento,` temos como saída:

```
 0.8 Paciente  1 com pulseira verde chega
 0.8 Paciente  1 com pulseira verde inicia o atendimento
 8.2 Paciente  2 com pulseira amarela chega
 8.2 Paciente  2 com pulseira amarela inicia o atendimento
11.0 Paciente  3 com pulseira verde chega
11.4 Paciente  4 com pulseira verde chega
11.7 Paciente  5 com pulseira vermelha chega
11.7 Paciente  1 com pulseira verde tem atendimento interrompido
11.7 Paciente  1 ainda precisa de  0.1 min de atendimento
11.7 Paciente  5 com pulseira vermelha inicia o atendimento
15.3 Paciente  5 com pulseira vermelha termina o atendimento
15.3 Paciente  3 com pulseira verde inicia o atendimento
18.7 Paciente  3 com pulseira verde termina o atendimento
18.7 Paciente  4 com pulseira verde inicia o atendimento
```

> **Desafio 12**: quando um paciente é interrompido, ele deseja retornar ao atendimento de onde parou. Altere o programa para que um paciente de pulseira verde interrompido possa retornar para ser atendido no tempo restante do seu atendimento. Dica: altere a numeração de prioridades de modo que um paciente verde interrompido tenha prioridade superior ao de um paciente verde que acabou de chegar.

Novamente, as alterações no modelo anterior resumem-se à função `atendimento` : precisamos aumentar a prioridade de um paciente interrompido em relação aos pacientes que acabam de chegar com a mesma pulseira, afinal, ele tem prioridade em relação a um paciente recém chegado de gravidade equivalente. Além disso, tal paciente, deve ser atendido pelo tempo *restante* de atendimento, de modo que a função deve receber como parâmetro esse tempo.

O artifício utilizado neste segundo caso foi acrescentar um parâmetro opcional à função, `tempoAtendimento` , de modo que se ele não é fornecido (caso de um paciente novo), a função sorteia um tempo exponecialmente distribuído, com média de 9 minutos. De outro modo, se o parâmetro é fornecido, isso significa que ele é um parceiro interrompido e, portanto, já tem um tempo restante de atendimento calculado.

O código a seguir, representa uma possível solução para a *nova* função `atendimento` do desafio:

```
def atendimento(env, paciente, pulseira, prio, preempt, medicos, tempoAtendimento=None):
    # ocupa um médico e realiza o atendimento do paciente

    with medicos.request(priority=prio, preempt=preempt) as req:
        yield req
        inicioAtendimento = env.now
        print("%4.1f %s com %s inicia o atendimento" %(env.now, paciente, pulseira))
        try:
            # sorteia o tempo de atendimento
            if not tempoAtendimento:
                tempoAtendimento = random.expovariate(1/9)
            yield env.timeout(tempoAtendimento)
            print("%4.1f %s com %s termina o atendimento"
                    %(env.now, paciente, pulseira))
        except simpy.Interrupt:
            # recalcula o tempo de atendimento
            tempoAtendimento -= env.now-inicioAtendimento
            print("%4.1f %s com %s tem atendimento interrompido"
                    %(env.now, paciente, pulseira))
            print("%4.1f %s ainda precisa de %4.1f min de atendimento"
            %(env.now, paciente, tempoAtendimento))

            # aumenta a prioridade reduzindo o valor
            prio -= 0.01
            env.process(atendimento(env, paciente, pulseira, prio, preempt, medicos, tempoAten
dimento))

random.seed(100)
env = simpy.Environment()
# cria os médicos
medicos = simpy.PreemptiveResource(env, capacity=2)
chegadas = env.process(chegadaPacientes(env, medicos))

env.run(until=20)
```

Quando executado por apenas 20 minutos, o modelo completo - acrescido da nova função `atendimento`, fornece como saída:

```
 0.8 Paciente  1 com pulseira verde chega
 0.8 Paciente  1 com pulseira verde inicia o atendimento
 8.2 Paciente  2 com pulseira amarela chega
 8.2 Paciente  2 com pulseira amarela inicia o atendimento
11.0 Paciente  3 com pulseira verde chega
11.4 Paciente  4 com pulseira verde chega
11.7 Paciente  5 com pulseira vermelha chega
11.7 Paciente  1 com pulseira verde tem atendimento interrompido
11.7 Paciente  1 ainda precisa de  0.1 min de atendimento
11.7 Paciente  5 com pulseira vermelha inicia o atendimento
15.3 Paciente  5 com pulseira vermelha termina o atendimento
15.3 Paciente  1 com pulseira verde inicia o atendimento
15.5 Paciente  1 com pulseira verde termina o atendimento
15.5 Paciente  3 com pulseira verde inicia o atendimento
18.8 Paciente  3 com pulseira verde termina o atendimento
18.8 Paciente  4 com pulseira verde inicia o atendimento
```

Note que agora, o Paciente 1, diferentemente do que ocorre na saída do desafio 11, é atendido antes do Paciente 3, representando o fato de que, mesmo interrompido, ele voltou para o início da fila.

Interrupções de processos: `simpy.Interrupt`

Você está todo feliz e contente atravessando a galáxia no seu X-Wing quando... PIMBA! Seu dróide astromecânico pifa e só lhe resta interromper a viagem para consertá-lo, antes que apareça um maldito caça TIE das forças imperiais.

Nesta seção iremos interromper processos já em execução e depois retomar a operação inicial. A aplicação mais óbvia é para a quebra de equipamentos durante a operação, como no caso do R2D2.

A interrupção de um processo em SimPy é realizada por meio de um comando `Interrupt` aplicado ao processo já iniciado. O cuidado aqui é que quando um recurso é interrompido por outro processo ele causa uma interrupção, de fato, no Python, o que nos permite utilizar o bloco de controle de interrupção `try:...except`, o que não deixa de ser uma boa coisa, dada a sua facilidade.

Criando quebras de equipamento

Voltando ao exemplo do X-Wing, considere que a cada 10 horas o R2D2 interrompe a viagem para uma manutenção de 5 horas e que a viagem toda levaria (sem não houvessem paralisações) 50 horas.

Inicialmente, vamos criar duas variáveis globais: uma para representar se o X-Wing está operando - afinal, não queremos interrompê-lo quando ele já estiver em manutenção - e outra para armazenar o tempo ainda restante para a viagem.

```
import simpy

viajando = False    # variável global que avisa se o x-wing está operando
duracaoViagem = 30  # variável global que marca a duração atual da viagem
```

O próximo passo, é criar uma função que represente a viagem do x-wing, garantindo não só que ela dure o tempo correto, mas também que lide com o processo de interrupção:

```
import simpy

viajando = False        # variável global que avisa se o x-wing está operando
duracaoViagem = 30      # variável global que marca a duração atual da viagem

def viagem(env, tempoParada):
    #processo de viagem do x-wing
    global viajando, duracaoViagem

    partida = env.now         # início da viagem
    while duracaoViagem > 0:  # enquanto ainda durar a viagem, execute:
        try:
            viajando = True
            # (re)inicio da viagem
            inicioViagem = env.now
            print("%5.1f Viagem iniciada" %(env.now))
            # tempo de viagem restante
            yield env.timeout(duracaoViagem)
            duracaoViagem -= env.now - inicioViagem

        except simpy.Interrupt:
            # se o processo de viagem foi interrompido execute
            # atualiza o tempo restante de viagem
            duracaoViagem -= env.now - inicioViagem
            print("%5.1f Falha do R2D2\tTempo de viagem restante: %4.1f horas"
                    %(env.now, duracaoViagem))
            # tempo de manutenção do R2D2
            yield env.timeout(tempoParada)

    # ao final avisa o término da viagem e sua duração
    print("%5.1f Viagem concluida\tDuração total da viagem: %4.1f horas"
            %(env.now, env.now-partida))


env = simpy.Environment()
viagem = env.process(viagem(env, 15))

env.run()
```

O importante no programa anterior é notar o bloco `try:...except:,` interno ao laço `while duracaoViagem > 0,` que mantém o nosso X-Wing em processo enquanto a variável `duracaoViagem` for maior que zero. O `except` aguarda um novo comando, o `simpy.Interrupt` , que nada mais é do que uma interrupção causada por algum outro processo dentro do `Environment.`

Quando executado, o programa fornece uma viagem tranquila:

```
  0.0 Viagem iniciada
 30.0 Viagem concluida  Duração total da viagem: 30.0 horas
```

A viagem é tranquila, pois não criamos ainda um "gerador de interrupções", que nada mais é do que um processo em SimPy que cria a interrupção da viagem.

Note, na penúltima linha do código anterior, que o processo em execução foi armazenado na variável `viagem` e oque devemos fazer é interrompê-lo de 10 em 10 horas. Para tanto, a função `paradaTecnica` a seguir, verifica se o processo de viagem está em andamento e paralisa a operação depois de 10 horas:

```
def paradaTecnica(env, intervaloQuebra, viagem):
    # processo de paradas entre intervalos de quebra
    global viajando, duracaoViagem

    while duracaoViagem > 0:                # este processo só ocorre durante a viagem
        yield env.timeout(intervaloQuebra)  # aguarda a próxima quebra do R2D2
        if viajando:                        # R2D2 somente quebra durante a viagem
            viagem.interrupt()              # interrompe o processo viagem
            viajando = False                # desliga a viagem

env = simpy.Environment()
viagem = env.process(viagem(env, 15))
env.process(paradaTecnica(env, 10, viagem))

env.run()
```

A função `paradaTecnica,` portanto, recebe como parâmetro o próprio *objeto* que representa o processo `viagem` e, por meio do comando:

```
viagem.interrupt()
```

Provoca uma interrupção no processo, a ser reconhecida pela função `viagem` na linha:

```
except simpy.Interrupt:
```

Adicionalmente, o processo parada técnica também deve ser inciado ao início da simulação, de modo que a parte final do modelo fica:

```
env = simpy.Environment()
viagem = env.process(viagem(env, 15))
env.process(paradaTecnica(env, 10, viagem))

env.run()
```

Quando executado, o modelo completo fornece como saída:

```
  0.0 Viagem iniciada
 10.0 Falha do R2D2     Tempo de viagem restante: 20.0 horas
 25.0 Viagem iniciada
 30.0 Falha do R2D2     Tempo de viagem restante: 15.0 horas
 45.0 Viagem iniciada
 50.0 Falha do R2D2     Tempo de viagem restante: 10.0 horas
 65.0 Viagem iniciada
 70.0 Falha do R2D2     Tempo de viagem restante:  5.0 horas
 85.0 Viagem iniciada
 90.0 Falha do R2D2     Tempo de viagem restante:  0.0 horas
105.0 Viagem concluida  Duração total da viagem: 105.0 horas
```

Alguns aspectos importantes do código anterior:

1. A utilização de variáveis globais foi fundamental para informar ao processo de parada o status do processo de viagem. É por meio de variáveis globais que um processo "sabe" o que está ocorrendo no outro;
2. Como a execução `env.run()` não tem um tempo final pré-estabelecido, a execução dos processos é terminada quando o laço:

   ```
   while duracaoViagem > 0
   ```

   Torna-se falso. Note que esse `while` deve existir nos dois processos em execução, caso contrário, o programa seria executado indefinidamente;

3. Dentro da função `paradaTecnica` a variável global `viajando` impede que ocorram duas quebras ao mesmo tempo. Naturalmente o leitor atento sabe que isso jamais ocorreria, afinal, o tempo de duração da quebra é inferior ao intervalo entre quebras. Mas fica o exercício: execute o mesmo programa, agora para uma duração de quebra de 15 horas e veja o que acontece.

4. Se um modelo possui uma lógica de interrupção `.interrupt()` e não possui um comando `except simpy.Interrupt` para lidar com a paralização do processo, o SimPy finalizará a simulação retornando o erro:

   ```
   Interrupt: Interrupt(None)
   ```

   Na próxima seção é apresentada uma alternativa para manter o processamento da simulação.

Interrompendo um proccesso sem captura por `try...except`

Caso o modelo possua um comando de interrupção de processo, mas não exista nenhuma lógica de captura com o bloco `try... except,` a simulação será finalizada com a mensagem de erro:

```
Interrupt: Interrupt(None)
```

O SimPy cria, para cada processo, uma propriedade chamada `defused` que permite contornar a paralisação. Assim, pode-se interromper um processo, sem que essa interrupção provoque estrago algum ao processamento do modelo. Para *desativar* a paralisação do modelo (e manter apenas a paralisação do processo), basta tornar a propriedade `defused = True` :

```
processVar.interrupt()
processVar.defused = True
```

Um exemplo bem prático: você está na sua rotina de exercícios matinais, quando... PIMBA:

```
import simpy

def forca(env):
    # processo a ser interrompido
    while True:
        yield env.timeout(1)
        print('%d Eu estou com a Força e a Força está comigo.' % env.now)

def ladoNegro(env, proc):
    yield env.timeout(3)
    print('%d Venha para o lado negro da força, nós temos CHURROS!' % env.now)
    # interrompe o processo proc
    proc.interrupt()
    # defused no processo para evitar a interrupção da simulação
    proc.defused = True
    print('%d Ponto para o Império!' % env.now)

env = simpy.Environment()

forcaProc = env.process(forca(env))
ladoNegroProc = env.process(ladoNegro(env, forcaProc))

env.run()
```

O modelo anterior deve ser autoexplicativo: um processo `forca` é interrompido pelo
Quando executado, o modelo fornece como resultado:

```
1 Eu estou com a Força e a Força está comigo.
2 Eu estou com a Força e a Força está comigo.
3 Venha para o lado negro da força, nós temos CHURROS!
3 Ponto para o Império!
```

Conceitos desta seção

Conteúdo	Descrição
`processVar = env.process(função_processo(env))`	armazena o processo da função_processo na variável `processVar`
`processVar.interrupt()`	interrompe o processo armazenado na variável `processVar`
`try:...except simpy.Interrupt`	lógica `try...except` necessária para interrupção do processo

Desafios

> **Desafio 13** Considere que existam dois tipos de paradas: uma do R2D2 e outra do canhão de combate. A parada do canhão de combate ocorre sempre depois de 25 horas de viagem (em quebra ou não) e seu reparo dura 2 horas. Contudo, para não perder tempo, a manutenção do canhão só é realizada quando o R2D2 quebra.
>
> **Desafio 14** Você não acha que pode viajar pelo espaço infinito sem encontrar alguns TIEs das forças imperiais, não é mesmo? Considere que a cada 25 horas, você se depara com um TIE imperial. O ataque dura 30 minutos e, se nesse tempo você não estiver com o canhão funcionando, a sua próxima viagem é para o encontro do mestre Yoda.
>
> Dica: construa uma função `executaCombate` que controla todo o processo de combate. Você vai precisar também de uma variável global que informa se o X-Wing está ou não em combate.

Solução dos desafios 13 e 14

> **Desafio 13** Considere que existam dois tipos de paradas: uma do R2D2 e outra do canhão de combate. A parada do canhão de combate ocorre sempre depois de 25 horas de viagem (em quebra ou não) e seu reparo dura 2 horas. Contudo, para não perder tempo, a manutenção do canhão só é realizada quando o R2D2 quebra.

Inicialmente, precisamos de uma variável global para verificar a situação do canhão:

```
import simpy

canhao = True           # variável global que avisa se o canhão está funcionando
viajando = False        # variável global que avisa se o x-wing está operando
duracaoViagem = 30      # variável global que marca a duração atual da viagem
```

A função viagem, agora deve lidar com um tempo de parada do canhão. Contudo, não existe uma interrupção para o canhão, pois nosso indomável piloto jedi apenas verifica a situação do canhão ao término do concerto do R2D2:

```
def viagem(env, tempoParada, tempoParadaCanhao):
    # processo de viagem do x-wing
    global viajando, duracaoViagem, canhao

    partida = env.now         # início da viagem
    while duracaoViagem > 0:  # enquanto ainda durar a viagem, execute:
        try:
            viajando = True
            # (re)inicio da viagem
            inicioViagem = env.now
            print("%5.1f Viagem iniciada" %(env.now))
            # tempo de viagem restante
            yield env.timeout(duracaoViagem)
            duracaoViagem -= env.now - inicioViagem

        except simpy.Interrupt:
            # se o processo de viagem foi interrompido execute
            # atualiza o tempo restante de viagem
            duracaoViagem -= env.now - inicioViagem
            print("%5.1f Falha do R2D2\tTempo de viagem restante: %4.1f horas"
                    %(env.now, duracaoViagem))
            # tempo de manutenção do R2D2
            yield env.timeout(tempoParada)
            print("%5.1f R2D2 operante" %env.now)
            # se o canhão não estiver funcionando
            if not canhao:
                print("%5.1f R2, ligue o canhão!" % env.now)
                # tempo de manutenção do canhão
                yield env.timeout(tempoParadaCanhao)
                canhao = True
                print("%5.1f Pi..bi...bi\tTradução: canhão operante!" % env.now)

    # ao final avisa o término da viagem e sua duração
    print("%5.1f Viagem concluida\tDuração total da viagem: %4.1f horas"
            %(env.now, env.now-partida))
```

Além da função de parada técnica (já pertencente ao nosso modelo desde a seção anterior), precisamos de uma função que tire de operação o canhão, ou seja, apenas desligue a variável global `canhao:`

```
def quebraCanhao(env, intervaloCanhao):
    # processo de quebra do canhao entre intervalos definidos
    global canhao, duracaoViagem

    while duracaoViagem > 0:                # este processo só ocorre durante a viagem
        yield env.timeout(intervaloCanhao)  # aguarda a próxima quebra do canhao
        if canhao:
            canhao = False
            print("%5.1f Pi uiui...bi\tTradução: canhão inoperante!" % (env.now))
```

Por fim, o processo de quebra do canhão deve ser inciado juntamente com o resto do modelo de simulação:

```
env = simpy.Environment()
viagem = env.process(viagem(env, 15, 2))
env.process(paradaTecnica(env, 10, viagem))
env.process(quebraCanhao(env, 25))

env.run()
```

Quando o modelo completo é executado, ele fornece como saída:

```
  0.0 Viagem iniciada
 10.0 Falha do R2D2     Tempo de viagem restante: 20.0 horas
 25.0 Pi uiui...bi      Tradução: canhão inoperante!
 25.0 R2D2 operante
 25.0 R2, ligue o canhão!
 27.0 Pi..bi...bi       Tradução: canhão operante!
 27.0 Viagem iniciada
 30.0 Falha do R2D2     Tempo de viagem restante: 17.0 horas
 45.0 R2D2 operante
 45.0 Viagem iniciada
 50.0 Pi uiui...bi      Tradução: canhão inoperante!
 50.0 Falha do R2D2     Tempo de viagem restante: 12.0 horas
 65.0 R2D2 operante
 65.0 R2, ligue o canhão!
 67.0 Pi..bi...bi       Tradução: canhão operante!
 67.0 Viagem iniciada
 70.0 Falha do R2D2     Tempo de viagem restante:  9.0 horas
 75.0 Pi uiui...bi      Tradução: canhão inoperante!
 85.0 R2D2 operante
 85.0 R2, ligue o canhão!
 87.0 Pi..bi...bi       Tradução: canhão operante!
 87.0 Viagem iniciada
 90.0 Falha do R2D2     Tempo de viagem restante:  6.0 horas
100.0 Pi uiui...bi      Tradução: canhão inoperante!
105.0 R2D2 operante
105.0 R2, ligue o canhão!
107.0 Pi..bi...bi       Tradução: canhão operante!
107.0 Viagem iniciada
110.0 Falha do R2D2     Tempo de viagem restante:  3.0 horas
125.0 Pi uiui...bi      Tradução: canhão inoperante!
125.0 R2D2 operante
125.0 R2, ligue o canhão!
127.0 Pi..bi...bi       Tradução: canhão operante!
127.0 Viagem iniciada
130.0 Falha do R2D2     Tempo de viagem restante:  0.0 horas
145.0 R2D2 operante
145.0 Viagem concluida  Duração total da viagem: 145.0 horas
150.0 Pi uiui...bi      Tradução: canhão inoperante!
```

> **Desafio 14** Você não acha que pode viajar pelo espaço infinito sem encontrar alguns TIEs das forças imperiais, não é mesmo? Considere que a cada 25 horas, você se depara com um TIE imperial. O ataque dura 30 minutos e, se nesse tempo você não estiver com o canhão funcionando, a sua próxima viagem é para o encontro do mestre Yoda.
>
> Dica: construa uma função executaCombate que controla todo o processo de combate. Você vai precisar também de uma variável global que informa se o X-Wing está ou não em combate.

Incialmente, precisamos de uma variável global para verificar a situação do combate:

```
import simpy

emCombate = False       # variável global que avisa se o x-wing está em combate
canhao = True           # variável global que avisa se o canhão está funcionando
viajando = False        # variável global que avisa se o x-wing está operando
duracaoViagem = 30      # variável global que marca a duração atual da viagem
```

A função `executaCombate` a seguir, incia o processo de combate e verifica quem foi o vitorioso:

```
def executaCombate(env, intervaloCombate, duracaoCombate):
    # processo de execução do combate

    global emCombate, canhao, duracaoViagem

    while duracaoViagem > 0: # este processo só ocorre durante a viagem
        # aguarda o próximo encontro com as forças imperirais
        yield env.timeout(intervaloCombate)
        # inicio do combate
        emCombate = True
        if not canhao:
            print("%5.1f Fim da viagem\tO lado negro venceu" %env.now)
            break
        else:
            print("%5.1f Combate iniciado\tReze para que o canhão não quebre!"
                    %env.now)
            try:
                yield env.timeout(duracaoCombate)
                emCombate = False
                print("%5.1f Fim do combate\tR2D2 diz: tá tudo tranquilo, tá tudo normalizado."

                        %env.now)
            except simpy.Interrupt:
                print("%5.1f OPS... O canhão quebrou durante o combate." %env.now)
                print("%5.1f Fim da viagem\tO lado negro venceu" %env.now)
                break
```

A quebra de canhão agora deve verificar se a nave está em combate, pois, neste caso, o X-Wing será esmagado pelo TIE:

```
def quebraCanhao(env, intervaloCanhao, combate):
    # processo de quebra do canhao entre intervalos definidos
    global canhao, duracaoViagem, emCombate

    while duracaoViagem > 0:                # este processo só ocorre durante a viagem
        yield env.timeout(intervaloCanhao)  # aguarda a próxima quebra do canhao
        if canhao:
            canhao = False
            print("%5.1f Pi uiui...bi\tTradução: canhão inoperante!" % (env.now))
            if emCombate:
                combate.interrupt()
```

Finalmente, a incialização do modelo deve contar com uma chamada para o processo `executaCombate:`

```
env = simpy.Environment()
viagem = env.process(viagem(env, 15, 2))
combate = env.process(executaCombate(env, 20, 0.5))
env.process(paradaTecnica(env, 10, viagem))
env.process(quebraCanhao(env, 25, combate))

env.run(until=combate)
```

A última linha do código anterior tem algo importante: optei por executar a simulação enquanto durar o processo de combate, afinal, ele mesmo só é executado caso a X-Wing esteja ainda em viagem.

Por fim, quando executado, o modelo completo fornece como saída:

```
 0.0 Viagem iniciada
10.0 Falha do R2D2     Tempo de viagem restante: 20.0 horas
20.0 Combate iniciado  Reze para que o canhão não quebre!
20.5 Fim do combate    R2D2 diz: tá tudo tranquilo, tá tudo normalizado.
25.0 Pi uiui...bi      Tradução: canhão inoperante!
25.0 R2D2 operante
25.0 R2, ligue o canhão!
27.0 Pi..bi...bi       Tradução: canhão operante!
27.0 Viagem iniciada
30.0 Falha do R2D2     Tempo de viagem restante: 17.0 horas
40.5 Combate iniciado  Reze para que o canhão não quebre!
41.0 Fim do combate    R2D2 diz: tá tudo tranquilo, tá tudo normalizado.
45.0 R2D2 operante
45.0 Viagem iniciada
50.0 Pi uiui...bi      Tradução: canhão inoperante!
50.0 Falha do R2D2     Tempo de viagem restante: 12.0 horas
61.0 Fim da viagem     O lado negro venceu
```

Teste seus conhecimentos

1. Não colocamos distribuições aleatórias nos processos. Acrescente distribuições nos diversos

processos e verifique se o modelo precisa de alterações;

2. Acrescente a tentativa da destruição da Estrela da Morte ao final da viagem: com 50% de chances, nosso intrépido jedi consegue acertar um tiro na entrada do reator e explodir a Estrela da Morte (se ele erra, volta ao combate). Replique algumas vezes o modelo e estime a probabilidade de sucesso da operação.

Armazenagem e seleção de objetos específicos com `Store, FilterStore e PriorityStore`

O SimPy possui uma ferramenta para armazenamento de objetos - como valores, recursos etc. - chamada `Store;` um comando de acesso a objetos específicos dentro do `Store` por meio de filtro, o `FilterStore` e um comando de acesso de objetos por ordem de prioridade, o `PriorityStore.` O programador experiente vai notar certa similaridade da família `Store` com o dicionário do Python.

Vamos desvendar o funcionamento do `Store` a partir de um exemplo bem simples: simulando o processo de atendimento em uma barbearia com três barbeiros. Quando você chega a uma barbearia e tem uma ordem de preferência entre os barbeiros, isto é: barbeiro 1 vem antes do 2, que vem antes do 3, precisará selecionar o *recurso* barbeiro na ordem certa de sua preferência, mas lembre-se: cada cliente tem seu gosto e gosto não se discute, simula-se!

Construindo um conjunto de objetos com `Store`

Inicialmente, considere que os clientes são atribuídos ao primeiro barbeiro que encontrar-se livre, indistintamente. Se todos os barbeiros estiverem ocupados, o cliente aguarda em fila.

O comando que constrói um armazém de objetos é o `simpy.Store():`

```
meuStore = simpy.Store(env, capacity=capacidade)
```

Para manipular o `Store` criado, temos três comandos à disposição:

- `meuStore.items:` adiciona objetos ao `meuStore;`
- `yield meuStore.get():` retira o primeiro objeto disponível de `meuStore` ou, caso o `meuStore` esteja vazio, aguarda até que algum objeto sela colocado no `Store;`
- `yield meuStore.put(umObjeto):` coloca um objeto no `meuStore` ou, caso o `meuStore` esteja cheio, aguarda até que surja um espaço vazio para colocar o objeto.

> Observação: se a capacidade não for fornecida, o SimPy assumirá que a capacidade do Store é ilimitada.

Para a nossa barbearia, criaremos um `Store` que armazenará o nome dos barbeiros, aqui denominados de 0, 1 e 2:

```
env = simpy.Environment()

# cria 3 barbeiros diferentes
barbeirosList = [simpy.Resource(env, capacity=1) for i in range(3)]

# cria um Store para armazenar os barbeiros
barbeariaStore = simpy.Store(env, capacity=3)
barbeariaStore.items = [0, 1, 2]
```

No código anterior, criamos uma lista com três recursos que representarão os barbeiros. A seguir, criamos uma `Store,` chamada `barbeariaStore,` de capacidade 3 e adicionamos, na linha seguinte, um lista com os três números que representam os próprios barbeiros.

Em resumo, nosso `Store` contém apenas os números 0, 1 e 2.

Considere que o intervalo entre chegadas sucessivas de clientes é exponencialmente distribuído com média de 5 minutos e que cada barbeiro leva um tempo normalmente distribuído com média 10 e desvio padrão de 2 minutos para cortar o cabelo.

Uma possível máscara para o modelo de simulação seria:

```
import simpy
import random

TEMPO_CHEGADAS = 5      # intervalo entre chegadas de clientes
TEMPO_CORTE = [10, 2]   # tempo médio de corte

def chegadaClientes(env, barbeariaStore):
    # gera clientes exponencialmente distribuídos
    # encaminha para o processo de atendimento

def atendimento(env, cliente, barbeariaStore):
    # ocupa um barbeiro específico e realiza o corte

random.seed(100)
env = simpy.Environment()

# cria 3 barbeiros diferentes
barbeirosList = [simpy.Resource(env, capacity=1) for i in range(3)]

# cria um Store para armazenar os barbeiros
barbeariaStore = simpy.Store(env, capacity=3)
barbeariaStore.items = [0, 1, 2]

# inicia processo de chegadas de clientes
env.process(chegadaClientes(env, barbeariaStore))
env.run(until = 20)
```

A função para gerar clientes é semelhante a tantas outras que já fizemos neste livro:

```
def chegadaClientes(env, barbeariaStore):
    # gera clientes exponencialmente distribuídos
    # encaminha para o processo de atendimento
    i = 0
    while True:
        yield env.timeout(random.expovariate(1/TEMPO_CHEGADAS))
        i += 1
        print("%5.1f Cliente %i chega." %(env.now, i))
        env.process(atendimento(env, i, barbeariaStore))
```

A função de atendimento, traz a novidade de que primeiro devemos *retirar* um objeto/barbeiro do `Store` e, ao final do atendimento, devolvê-lo ao `Store:`

```
def atendimento(env, cliente, barbeariaStore):
    # ocupa um barbeiro específico e realiza o corte
    chegada = env.now
    # retira o barbeiro do Store
    barbeiroNum = yield barbeariaStore.get()
    espera = env.now - chegada
    print("%5.1f Cliente %i inicia.\t\tBarbeiro %i ocupado.\tTempo de fila: %2.1f"
            %(env.now, cliente, barbeiroNum, espera))
    with barbeirosList[barbeiroNum].request() as req:
        yield req
        yield env.timeout(random.normalvariate(*TEMPO_CORTE))
        print("%5.1f Cliente %i termina.\tBarbeiro %i liberado."
                %(env.now, cliente, barbeiroNum))
    # devolve o barbeiro ao Store
    barbeariaStore.put(barbeiroNum)
```

Quando estamos retirando um objeto do `barbeariaStore,` estamos apenas retirando o nome (ou identificador) do barbeiro disponível. Algo como retirar um cartão com o nome do barbeiro de uma pilha de barbeiros disponíveis. Se o cartão é retirado, significa que, enquanto ele não for devolvido à pilha, nenhum cliente poderá ocupá-lo, pois ele não se encontra na pilha de barbeiros disponíveis.

Para ocuparmos o recurso (ou barbeiro) selecionado corretamente, utilizamos o identificador como índice da lista `barbeiroList` . Assim, temporariamente, o barbeiro retirado do Store fica indisponível para outros clientes, pois a linha:

```
barbeiroNum = yield barbeariaStore.get()
```

Só retornará um barbeiro dentre aqueles que ainda estão à disposição.
Ao ser executado por apenas 20 minutos, o modelo de simulação completo da barbeira fornece como saída:

```
11.8 Cliente 1 chega.
11.8 Cliente 1 inicia.          Barbeiro 0 ocupado.     Tempo de fila: 0.0
17.6 Cliente 2 chega.
17.6 Cliente 2 inicia.          Barbeiro 1 ocupado.     Tempo de fila: 0.0
19.5 Cliente 1 termina.         Barbeiro 0 liberado.
```

No caso do exemplo, o `Store` armazenou basicamente uma lista de números [0,1,2], que representam os nomes dos barbeiros. Poderíamos sofisticar um pouco mais o exemplo e criar um dicionário (em Python) para manipular os nomes reais dos barbeiros. Por exemplo, se os nomes dos barbeiros são: João, José e Mário, poderíamos montar o `barberirosStore` com seus respectivos nomes e evitar o uso de números:

```
random.seed(100)
env = simpy.Environment()

#cria 3 barbeiros diferentes e armazena em um dicionário
barbeirosList = [simpy.Resource(env, capacity=1) for i in range(3)]
barbeirosNomes = ['João', 'José', 'Mario']
barbeirosDict = dict(zip(barbeirosNomes, barbeirosList))

#cria um Store para armazenar os barbeiros
barbeariaStore = simpy.Store(env, capacity=3)
barbeariaStore.items = barbeirosNomes

# inicia processo de chegadas de clientes
env.process(chegadaClientes(env, barbeariaStore))
env.run(until = 20)
```

O exemplo anterior apenas reforça que `Store` é um local para se armazenar objetos de qualquer tipo (semelhante ao `dict` do Python).

> Observação: `Store` opera segundo uma regra FIFO *(Firt-in-First-out*), ou seja: o primeiro objeto a entrar no Store por meio de um `.put()` será o primeiro objeto a sair dele com um `.get()`.

Selecionando um objeto específico com `FilterStore()`

Considere agora o caso bastante comum em que precisamos selecionar um recurso específico (segundo alguma regra) dentro de um conjunto de recursos disponíveis. Na barbearia, por exemplo, cada cliente agora tem um barbeiro preferido e, se ele não está disponível, o cliente prefere aguardar sua liberação.
Neste caso, vamos assumir que a preferência de um cliente é uniformemente distribuída entre os três barbeiros.
Precisamos, portanto, de um modo de selecionar um objeto específico dentro do `Store`. O SimPy tem um comando para construir um conjunto de objetos filtrável, o `FilterStore`:

```
meuFilterStore = simpy.FilterStore(env, capacity=capacidade)
```

A grande diferença para o `Store` é que podemos utilizar uma função anônima do Python dentro do comando `.get()` e assim puxar um objeto de dentro do `FilterStore` segundo alguma regra codificada por nós.
Inicialmente, criaremos um `FilterStore` de barbeiros, que armazenará apenas os números 1, 2 e 3:

```
random.seed(150)
env = simpy.Environment()

#cria 3 barbeiros diferentes
barbeirosList = [simpy.Resource(env, capacity=1) for i in range(3)]

#cria um Store para armazenar os barbeiros
barbeariaStore = simpy.FilterStore(env, capacity=3)
barbeariaStore.items = [0, 1, 2]

# inicia processo de chegadas de clientes
env.process(chegadaClientes(env, barbeariaStore))
env.run(until = 20)
```

A função geradora de clientes terá uma ligeira modificação, pois temos de atribuir a cada cliente um barbeiro específico, respeitando o sorteio de uma distribuição uniforme:

```
import simpy
import random

TEMPO_CHEGADAS = 5          # intervalo entre chegadas de clientes
TEMPO_CORTE = [10, 5]       # tempo médio de corte
PREF_BARBEIRO = [0, 2]      # preferência de barbeiros

def chegadaClientes(env, barbeariaStore):
    # gera clientes exponencialmente distribuídos
    # sorteia o barbeiro
    # inicia processo de atendimento
    i = 0
    while True:
        yield env.timeout(random.expovariate(1/TEMPO_CHEGADAS))
        i += 1
        barbeiroEscolhido = random.randint(*PREF_BARBEIRO)
        print("%5.1f Cliente %i chega.\t\tBarbeiro %i escolhido."
                %(env.now, i, barbeiroEscolhido))
        env.process(atendimento(env, i, barbeiroEscolhido, barbeariaStore))
```

Na função anterior, o *atributo* `barbeiroEscolhido` armazena o número do barbeiro sorteado e envia a informação para a função que representa o processo de atendimento.
A função `atendimento` utilizará uma função anônima para buscar o barbeiro certo dentro do `FilterStore` criado:

```
def atendimento(env, cliente, barbeiroEscolhido, barbeariaStore):
    #ocupa um barbeiro específico e realiza o corte
    chegada = env.now
    barbeiroNum = yield barbeariaStore.get(lambda barbeiro: barbeiro==barbeiroEscolhido)
    espera = env.now - chegada
    print("%5.1f Cliente %i incia.\t\tBarbeiro %i ocupado.\tTempo de fila: %2.1f"
            %(env.now, cliente, barbeiroEscolhido, espera))
    with barbeirosList[barbeiroNum].request() as req:
        yield req
        yield env.timeout(random.normalvariate(*TEMPO_CORTE))
        print("%5.1f Cliente %i termina.\tBarbeiro %i liberado."
                %(env.now, cliente, barbeiroEscolhido))
    barbeariaStore.put(barbeiroNum)
```

Para selecionar o número certo do barbeiro, existe uma função `lambda` inserida dentro do `.get()`:

```
barbeiroNum = yield barbeariaStore.get(lambda barbeiro: barbeiro==barbeiroEscolhido)
```

Esta função percorre os objetos dentro da `barbeariaStore` até encontrar um que tenha o número do respectivo barbeiro desejado pelo cliente. Note que também poderíamos ter optado por uma construção alternativa utilizando o nome dos barbeiros e não os números - neste caso, uma alternativa seria seguir o exemplo da seção anterior e utilizar um dicionário para associar o nome dos barbeiros aos respectivos recursos.
Quando executado, o modelo anterior fornece:

```
  8.7 Cliente 1 chega.          Barbeiro 1 escolhido.
  8.7 Cliente 1 inicia.         Barbeiro 1 ocupado.     Tempo de fila: 0.0
  9.7 Cliente 2 chega.          Barbeiro 0 escolhido.
  9.7 Cliente 2 inicia.         Barbeiro 0 ocupado.     Tempo de fila: 0.0
 12.4 Cliente 3 chega.          Barbeiro 1 escolhido.
 15.5 Cliente 1 termina.        Barbeiro 1 liberado.
 15.5 Cliente 3 inicia.         Barbeiro 1 ocupado.     Tempo de fila: 3.0
```

Repare que o cliente 3 chegou num instante em que o barbeiro 1 estava ocupado atendendo o cliente 1, assim ele foi obrigado a esperar em fila por 3 minutos, até que o cliente 1 liberasse o Barbeiro 1.

Criando um `Store` com prioridade: `PriorityStore`

Como sabemos, um `Store` segue a regra FIFO, de modo que o primeiro objeto a entrar no `Store` será o primeiro a sair do `Store`. É possível quebrar essa regra por meio do `PriorityStore`:

```
meuPriorityStore = simpy.PriorityStore(env, capacity=inf)
```

Para acrescentar um objeto qualquer ao `meuPriorityStore` já criado, a sequência de passos é primeiro criar um objeto `PriorityItem` - que representará o objeto a ser armazendo - para depois inseri-lo com um comando `put,` como representado no exemplo a seguir:

```
# criar o PriorityItem
meuObjetoPriority = simpy.PriorityItem(priority=priority, item=meuObjeto)
# adicionar o PriorityItem ao PriorityStore
meuPriorityStore.put(meuObjetoPriority)
```

> Observação: como no caso do `PriorityResource`, quanto menor o valor de `priority`, maior a preferência pelo objeto.

No caso dos barbeiros: "João, José e Mário", considere que a ordem de prioridades é a própria ordem alfabética dos nomes. Assim, inicialmente, construiremos dois dicionários para armazenar essas informações sobre os barbeiros:

```
random.seed(100)
env = simpy.Environment()

# cria 3 barbeiros diferentes e armazena em um dicionário
barbeirosList = [simpy.Resource(env, capacity=1) for i in range(3)]
barbeirosNomes = ['João', 'José', 'Mario']
# lista com a ordem de prioridades dos barbeiros
barbeirosPrioList = ['0', '1', '2']

# dicionário de recursos e prioridades
barbeirosDict = dict(zip(barbeirosNomes, barbeirosList))
barbeirosPrioDict = dict(zip(barbeirosNomes, barbeirosPrioList))
```

A partir dos dicionários anteriores, podemos construir um `PriorityStore` que armazena os nomes dos barbeiros e suas prioridades:

```
# cria um Store para armazenar os barbeiros
barbeariaStore = simpy.PriorityStore(env, capacity=3)
for nome in barbeirosNomes:
    barbeiro = simpy.PriorityItem(priority=barbeirosPrioDict[nome], item=nome)
    barbeariaStore.put(barbeiro)

# inicia processo de chegadas de clientes
env.process(chegadaClientes(env, barbeariaStore))
env.run(until = 20)
```

A função `atendimento` é muito semelhante às anteriores, basta notar que o comando `get` vai buscar não o nome, ma um *objeto* `PriorityItem` dentro do `PriorityStore`. Este objeto possui dois atributos: `item` - no nosso caso, o nome do barbeiro - e `priority` - a prioridade do objeto.

A seguir, apresenta-se uma possível implementação da função `atendimento.` Note que o objeto `barbeiro` é um `PriorityItem` e que, para sabermos o seu nome, precisamos do comando `barbeiro.item:`

```
def atendimento(env, cliente, barbeariaStore):
    #ocupa um barbeiro específico e realiza o corte
    chegada = env.now
    barbeiro = yield barbeariaStore.get()
    # extraí o nome do barbeiro
    barbeiroNome = barbeiro.item
    espera = env.now - chegada
    print("%5.1f Cliente %i inicia.\t\tBarbeiro %s ocupado.\tTempo de fila: %2.1f"
            %(env.now, cliente, barbeiroNome, espera))
    with barbeirosDict[barbeiroNome].request() as req:
        yield req
        yield env.timeout(random.normalvariate(*TEMPO_CORTE))
        print("%5.1f Cliente %i termina.\tBarbeiro %s liberado."
                %(env.now, cliente, barbeiroNome))
    barbeariaStore.put(barbeiro)
```

O modelo de simulação completo, quando simulado por apenas 20 minutos, fornece como saída:

```
 0.8 Cliente 1 chega.
 0.8 Cliente 1 inicia.         Barbeiro João ocupado.  Tempo de fila: 0.0
 3.8 Cliente 2 chega.
 3.8 Cliente 2 inicia.         Barbeiro José ocupado.  Tempo de fila: 0.0
10.4 Cliente 3 chega.
10.4 Cliente 3 inicia.         Barbeiro Mario ocupado. Tempo de fila: 0.0
12.7 Cliente 2 termina.        Barbeiro José liberado.
13.9 Cliente 1 termina.        Barbeiro João liberado.
14.2 Cliente 4 chega.
14.2 Cliente 4 inicia.         Barbeiro João ocupado.  Tempo de fila: 0.0
14.5 Cliente 5 chega.
14.5 Cliente 5 inicia.         Barbeiro José ocupado.  Tempo de fila: 0.0
17.8 Cliente 3 termina.        Barbeiro Mario liberado.
```

> Observação 1: internamente, o SimPy trata a "família" `Store` como recursos com capacidade ilimitada de armazenamento de objetos.
> Observação 2: uma implementação alternativa para o problema anterior, seria armazenar no `PriorityStore` não o nome do barbeiro, mas o próprio recurso criado. (Veja o tópico: "Teste seus conhecimentos" na próxima seção.

Conceitos desta seção

Conteúdo	Descrição
`meuStore = simpy.Store(env, capacity=capacity`	cria um *Store* `meuStore` : um armazém de objetos com capacidade `capacity` . Caso o parâmetro `capacity` não seja fornecido, o SimPy considera `capacity=inf` .
`yield meuStore.get()`	retira o primeiro objeto disponível de `meuStore` ou, caso o `meuStore` esteja vazio, aguarda até que algum objeto esteja disponível.
`yield meuStore.put(umObjeto)`	coloca um objeto no `meuStore` ou, caso o `meuStore` esteja cheio, aguarda até que surja um espaço vazio para colocar o objeto.
`meuFilterStore = simpy.FilterStore(env, capacity=capacity)`	cria um *Store* `meuStore` : um armazém de objetos filtráveis com capacidade `capacity` . Caso o parâmetro `capacity` não seja fornecido, o SimPy considera `capacity=inf` .
`yield meuFilterStore.get(filter=<function <lambda>>)`	retira o 1° objeto do `meuFilterStore` que retorne True para a função anônima fornecida por filter.
`meuPriorityStore = simpy.PriorityStore(env, capacity=inf)`	cria um *PriorityStore* `meuPriorityStore` - uma armazém de objetos com ordem de prioridade e capacidade `capacity` . Caso o parâmetro `capacity` não seja fornecido, o SimPy considera `capacity=inf` .
`meuObjetoPriority = simpy.PriorityItem(priority=priority, item=meuObjeto)`	cria um objeto `meuObjetoPriority` a partir de um objeto `meuObjeto` existente, com prioridade para ser armazenado em um `PriorityStore` . A `priority` deve ser um objeto ordenável.
`meuObjetoPriority.item`	retorna o atributo `item` do objeto `meuObjetoPriority` , definido no momento de criação do `PriorityItem.`
`meuObjetoPriority.priority`	retorna o atributo `priority` do objeto `meuObjetoPriority` , definido no momento de criação do `PriorityItem.`
`meuPriorityStore.put(meuObjetoPriority)`	coloca o objeto `meuObjetoPriority` no `PriorityStore` `meuPriorityStore` ou, caso o `meuPriorityStore` esteja cheio, aguarda até que surja um espaço vazio para colocar o objeto.
`yield meuPriorityStore.get()`	retorna o primeiro objeto disponível em `meuPriorityStore` respeitando a ordem de prioridade atribuída ao `PriorityItem` (objetos com valor menor de prioridade são escolhidos primeiro). Caso o `meuPriorityStore` esteja vazio, aguarda até que surja um espaço vazio para colocar o objeto.

Desafios

> **Desafio 15**: considere que na barbearia, 40% dos clientes escolhem seu barbeiro favorito, sendo que, 30% preferem o barbeiro A, 10% preferem o barbeiro B e nenhum prefere o barbeiro C (o proprietário do salão). Construa um modelo de simulação representativo deste sistema.
>
> **Desafio 16:** acrescente ao modelo da barbearia, a possibilidade de desistência e falta do barbeiro. Neste caso, existe 5% de chance de um barbeiro faltar em determinado dia. Neste caso, considere 3 novas situações:
>
> - Se o barbeiro favorito faltar, o respectivo cliente vai embora;
> - O cliente que não possuir um barbeiro favorito olha a fila de clientes: se houver mais de 6 clientes em fila, ele desiste e vai embora;
> - O cliente que possui um barbeiro favorito, não esperará se houver mais de 3 clientes esperando seu barbeiro favorito.

Solução dos desafios 15 e 16

> **Desafio 15**: considere que na barbearia, 40% dos clientes escolhem seu barbeiro favorito, sendo que, 30% preferem o barbeiro A, 10% preferem o barbeiro B e nenhum prefere o barbeiro C (o proprietário do salão). Construa um modelo de simulação representativo deste sistema.

Como existe preferência pelo barbeiro, naturalmente a escolha mais simples é trabalharmos com o `FilterStore` . O código a seguir, cria uma lista de barbeiros com os nomes, outra com os respectivos `Resources,` um dicionário para localizarmos o barbeiro por seu nome e, por fim, um `FilterStore` com os nomes dos barbeiros:

```
random.seed(50)
env = simpy.Environment()

# cria 3 barbeiros diferentes e armazena em um dicionário
barbeirosNomes = ['Barbeiro A', 'Barbeiro B', 'Barbeiro C']
barbeirosList = [simpy.Resource(env, capacity=1) for i in range(3)]
barbeirosDict = dict(zip(barbeirosNomes, barbeirosList))

# cria um FilterStore para armazenar os barbeiros
barbeariaStore = simpy.FilterStore(env, capacity=3)
barbeariaStore.items = barbeirosNomes

# inicia processo de chegadas de clientes
env.process(chegadaClientes(env, barbeariaStore))
env.run(until = 20)
```

Quando um cliente chega, existe 40% de chance dele preferir o barbeiro A e 10% de preferir o barbeiro B. O código a seguir atribui o barbeiro utilizando-se da função `random:`

```
def chegadaClientes(env, barbeariaStore):
    # gera clientes exponencialmente distribuídos
    i = 0
    while True:
        yield env.timeout(random.expovariate(1/TEMPO_CHEGADAS))
        i += 1
        # tem preferência por barbeiro?
        r = random.random()
        if r <= 0.30:
            barbeiroEscolhido ='Barbeiro A'
        elif r <= 0.40:
            barbeiroEscolhido = 'Barbeiro B'
        else:
            barbeiroEscolhido = 'Sem preferência'
        print("%5.1f Cliente %i chega.\t\t%s." %(env.now, i, barbeiroEscolhido))
        # inicia processo de atendimento
        env.process(atendimento(env, i, barbeiroEscolhido, barbeariaStore))
```

Por fim, o processo de atendimento deve diferenciar os clientes que possuem um barbeiro favorito. Assim, devemos criar uma função anônima `lambda` para resgatar o barbeiro correto do `FilterStore:`

```
def atendimento(env, cliente, barbeiroEscolhido, barbeariaStore):
    # ocupa um barbeiro específico e realiza o corte
    chegada = env.now
    if barbeiroEscolhido != 'Sem preferência':
        # retira do FilterStore o barbeiro escolhido no processo anterior
        barbeiro = yield barbeariaStore.get(lambda barbeiro: barbeiro==barbeiroEscolhido)
    else:
        # cliente sem preferência, retira o primeiro barbeiro do FilterStore
        barbeiro = yield barbeariaStore.get()
    espera = env.now - chegada
    print("%5.1f Cliente %i inicia.\t\t%s ocupado.\tTempo de fila: %2.1f"
          %(env.now, cliente, barbeiro, espera))
    # ocupa o recurso barbeiro
    with barbeirosDict[barbeiro].request() as req:
        yield req
        tempoCorte = random.normalvariate(*TEMPO_CORTE)
        yield env.timeout(tempoCorte)
        print("%5.1f Cliente %i termina.\t%s liberado." %(env.now, cliente, barbeiro))
    # devolve o barbeiro para o FilterStore
    barbeariaStore.put(barbeiro)
```

Note, no código anterior, que, caso o cliente não tenha barbeiro preferido, o `get` do `FilterStore` é utilizado sem nenhuma função `lambda` dentro do parêntesis.

Por fim, o código completo do modelo:

```
import simpy
import random

TEMPO_CHEGADAS = 5          # intervalo entre chegadas de clientes
TEMPO_CORTE = [10, 2]       # tempo médio de corte

def chegadaClientes(env, barbeariaStore):
    # gera clientes exponencialmente distribuídos
    i = 0
    while True:
        yield env.timeout(random.expovariate(1/TEMPO_CHEGADAS))
        i += 1
        # tem preferência por barbeiro?
        r = random.random()
        if r <= 0.30:
            barbeiroEscolhido ='Barbeiro A'
        elif r <= 0.40:
            barbeiroEscolhido = 'Barbeiro B'
        else:
            barbeiroEscolhido = 'Sem preferência'
        print("%5.1f Cliente %i chega.\t\t%s." %(env.now, i, barbeiroEscolhido))
        # inicia processo de atendimento
        env.process(atendimento(env, i, barbeiroEscolhido, barbeariaStore))

def atendimento(env, cliente, barbeiroEscolhido, barbeariaStore):
    #ocupa um barbeiro específico e realiza o corte
```

```
    chegada = env.now
    if barbeiroEscolhido != 'Sem preferência':
        # retira do FilterStore o barbeiro escolhido no processo anterior
        barbeiro = yield barbeariaStore.get(lambda barbeiro: barbeiro==barbeiroEscolhido)
    else:
        # cliente sem preferência, retira o primeiro barbeiro do FilterStore
        barbeiro = yield barbeariaStore.get()
    espera = env.now - chegada
    print("%5.1f Cliente %i inicia.\t\t%s ocupado.\tTempo de fila: %2.1f"
          %(env.now, cliente, barbeiro, espera))
    # ocupa o recurso barbeiro
    with barbeirosDict[barbeiro].request() as req:
        yield req
        tempoCorte = random.normalvariate(*TEMPO_CORTE)
        yield env.timeout(tempoCorte)
        print("%5.1f Cliente %i termina.\t%s liberado." %(env.now, cliente, barbeiro))
    # devolve o barbeiro para o FilterStore
    barbeariaStore.put(barbeiro)


random.seed(50)
env = simpy.Environment()

# cria 3 barbeiros diferentes e armazena em um dicionário
barbeirosNomes = ['Barbeiro A', 'Barbeiro B', 'Barbeiro C']
barbeirosList = [simpy.Resource(env, capacity=1) for i in range(3)]
barbeirosDict = dict(zip(barbeirosNomes, barbeirosList))

# cria um FilterStore para armazenar os barbeiros
barbeariaStore = simpy.FilterStore(env, capacity=3)
barbeariaStore.items = barbeirosNomes

# inicia processo de chegadas de clientes
env.process(chegadaClientes(env, barbeariaStore))
env.run(until = 20)
```

Quando executado por apenas 20 minutos, o modelo anterior fornece:

```
  3.4 Cliente 1 chega.          Barbeiro A.
  3.4 Cliente 1 inicia.         Barbeiro A ocupado.     Tempo de fila: 0.0
  8.5 Cliente 2 chega.          Sem preferência.
  8.5 Cliente 2 inicia.         Barbeiro B ocupado.     Tempo de fila: 0.0
  9.0 Cliente 3 chega.          Barbeiro A.
  9.8 Cliente 4 chega.          Sem preferência.
  9.8 Cliente 4 inicia.         Barbeiro C ocupado.     Tempo de fila: 0.0
 11.8 Cliente 1 termina.        Barbeiro A liberado.
 11.8 Cliente 3 inicia.         Barbeiro A ocupado.     Tempo de fila: 2.8
 16.6 Cliente 2 termina.        Barbeiro B liberado.
 19.0 Cliente 4 termina.        Barbeiro C liberado.
```

> **Desafio 16:** acrescente ao modelo da barbearia, a possibilidade de desistência e falta do barbeiro. Neste caso, existe 5% de chance de um barbeiro faltar em determinado dia. Além disso, considere 3 novas situações:
>
> - Se o barbeiro favorito faltar, o respectivo cliente vai embora;
> - O cliente que não possuir um barbeiro favorito olha a fila de clientes: se houver mais de 6 clientes em fila, ele desiste e vai embora;
> - O cliente que possui um barbeiro favorito, não esperará se houver mais de 3 clientes esperando seu barbeiro favorito.

Como teremos de identificar quantos clientes estão aguardando o respectivo barbeiro favorito, uma saída seria utilizar um dicionário para armazenar o número de clientes em fila (outra possibilidade seria um `Store` específico para cada fila):

```
random.seed(25)
env = simpy.Environment()

# cria 3 barbeiros diferentes e armazena em um dicionário
barbeirosNomes = ['Barbeiro A', 'Barbeiro B', 'Barbeiro C']

# falta de um barbeiro
if random.random() <= 0.05:
    barbeirosNomes.remove(random.choice((barbeirosNomes)))

barbeirosList = [simpy.Resource(env, capacity=1) for i in range(len(barbeirosNomes))]
barbeirosDict = dict(zip(barbeirosNomes, barbeirosList))

# dicionário para armazenar o número de clientes em fila de favoritos
filaDict = {k:0 for k in barbeirosNomes}

# cria um FilterStore para armazenar os barbeiros
barbeariaStore = simpy.FilterStore(env, capacity=3)
barbeariaStore.items = barbeirosNomes

# inicia processo de chegadas de clientes
env.process(chegadaClientes(env, barbeariaStore))
env.run(until = 30)
```

Para garantir a falta de um barbeiro em 5% das simulações, foi novamente utilizado o comando `random.random` e o comando `[random.choice](https://docs.python.org/dev/library/random.html#random.choice),` que seleciona uniformemente um elemento da lista `barbeirosNomes` :

```
if random.random() <= 0.05:
    barbeirosNomes.remove(random.choice((barbeirosNomes)))
```

Na linha anterior, além de sortearmos um dos barbeiros, ele é removido da lista de barbeiros, o que facilita o processo de desistência do cliente.

O processo de chegadas de clientes não precisa ser modificado em relação ao desafio anterior, contudo, o processo de atendimento precisa armazenar o número de clientes em fila por barbeiro - que pode ser feito por meio de um dicionário - e o número de clientes em fila total - que pode ser feito por meio de uma variável global que armazena o número total de clientes em fila.

Uma possível codificação para a nova função `atendimento` seria:

```
def atendimento(env, cliente, barbeiroEscolhido, barbeariaStore):
    # ocupa um barbeiro específico e realiza o corte
    global filaAtual     # número de clientes em fila

    chegada = env.now
    if barbeiroEscolhido != 'Sem preferência':
        if barbeiroEscolhido not in barbeirosDict:
            # caso o barbeiro tenha faltado, desiste do atendimento
            print("%5.1f Cliente %i desiste.\t%s ausente."
                %(env.now, cliente, barbeiroEscolhido))
            env.exit()
        if filaDict[barbeiroEscolhido] > 3:
            # caso a fila seja maior do que 6, desiste do atendimento
            print("%5.1f Cliente %i desiste.\t%s com mais de 3 clientes em fila."
                %(env.now, cliente, barbeiroEscolhido))
            env.exit()
        # cliente atual entra em fila e incrementa a fila do barbeiro favorito
        filaAtual += 1
        filaDict[barbeiroEscolhido] = filaDict[barbeiroEscolhido] + 1
        barbeiro = yield barbeariaStore.get(lambda barbeiro: barbeiro==barbeiroEscolhido)
        filaDict[barbeiroEscolhido] = filaDict[barbeiroEscolhido] - 1
    else:
        # cliente sem preferência, verifica o tamanho total da fila
        if filaAtual > 6:
            # caso a fila seja maior do que 6, desiste do atendimento
            print("%5.1f Cliente %i desiste.\tFila com mais de 6 clientes em fila."
                %(env.now, cliente))
            env.exit()
        else:
            # cliente entra em fila e pega o primeiro barbeiro livre
            filaAtual += 1
            barbeiro = yield barbeariaStore.get()

    # cliente já tem barbeiro, então sai da fila
    filaAtual -= 1

    espera = env.now - chegada
    print("%5.1f Cliente %i inicia.\t\t%s ocupado.\tTempo de fila: %2.1f"
            %(env.now, cliente, barbeiro, espera))
    # ocupa o recurso barbeiro
    with barbeirosDict[barbeiro].request() as req:
        yield req
        tempoCorte = random.normalvariate(*TEMPO_CORTE)
        yield env.timeout(tempoCorte)
        print("%5.1f Cliente %i termina.\t%s liberado." %(env.now, cliente, barbeiro))
    # devolve o barbeiro para o FilterStore
    barbeariaStore.put(barbeiro)
```

O código completo do modelo do desafio, ficaria:

```
import simpy
import random

TEMPO_CHEGADAS = 5          # intervalo entre chegadas de clientes
TEMPO_CORTE = [10, 2]       # tempo médio de corte
filaAtual = 0               # armazena o tamanho atual da fila de clientes

def chegadaClientes(env, barbeariaStore):
    # gera clientes exponencialmente distribuídos
    i = 0
    while True:
        yield env.timeout(random.expovariate(1/TEMPO_CHEGADAS))
        i += 1
        # tem preferência por barbeiro?
        r = random.random()
        if r <= 0.30:
            barbeiroEscolhido = 'Barbeiro A'
        elif r <= 0.40:
            barbeiroEscolhido = 'Barbeiro B'
        else:
            barbeiroEscolhido = 'Sem preferência'
        print("%5.1f Cliente %i chega.\t\t%s." %(env.now, i, barbeiroEscolhido))
        # inicia processo de atendimento
        env.process(atendimento(env, i, barbeiroEscolhido, barbeariaStore))

def atendimento(env, cliente, barbeiroEscolhido, barbeariaStore):
    #ocupa um barbeiro específico e realiza o corte
    global filaAtual

    chegada = env.now
    if barbeiroEscolhido != 'Sem preferência':
        if barbeiroEscolhido not in barbeirosDict:
            # caso o barbeiro tenha faltado, desiste do atendimento
            print("%5.1f Cliente %i desiste.\t%s ausente."
                %(env.now, cliente, barbeiroEscolhido))
            env.exit()
        if filaDict[barbeiroEscolhido] > 3:
            # caso a fila seja maior do que 6, desiste do atendimento
            print("%5.1f Cliente %i desiste.\t%s com mais de 3 clientes em fila."
                %(env.now, cliente, barbeiroEscolhido))
            env.exit()
        # cliente atual entra em fila e incrementa a fila do barbeiro favorito
        filaAtual += 1
        filaDict[barbeiroEscolhido] = filaDict[barbeiroEscolhido] + 1
        barbeiro = yield barbeariaStore.get(lambda barbeiro: barbeiro==barbeiroEscolhido)
        filaDict[barbeiroEscolhido] = filaDict[barbeiroEscolhido] - 1
    else:
        # cliente sem preferência, verifica o tamanho total da fila
        if filaAtual > 6:
            # caso a fila seja maior do que 6, desiste do atendimento
            print("%5.1f Cliente %i desiste.\tFila com mais de 6 clientes em fila."
                %(env.now, cliente))
            env.exit()
        else:
            # cliente entra em fila e pega o primeiro barbeiro livre
            filaAtual += 1
            barbeiro = yield barbeariaStore.get()
```

```
    # cliente já tem barbeiro, então sai da fila
    filaAtual -= 1

    espera = env.now - chegada
    print("%5.1f Cliente %i inicia.\t\t%s ocupado.\tTempo de fila: %2.1f"
            %(env.now, cliente, barbeiro, espera))
    # ocupa o recurso barbeiro
    with barbeirosDict[barbeiro].request() as req:
        yield req
        tempoCorte = random.normalvariate(*TEMPO_CORTE)
        yield env.timeout(tempoCorte)
        print("%5.1f Cliente %i termina.\t%s liberado." %(env.now, cliente, barbeiro))
    # devolve o barbeiro para o FilterStore
    barbeariaStore.put(barbeiro)

random.seed(25)
env = simpy.Environment()

# cria 3 barbeiros diferentes e armazena em um dicionário
barbeirosNomes = ['Barbeiro A', 'Barbeiro B', 'Barbeiro C']

# falta de um barbeiro
if random.random() <= 0.05:
    barbeirosNomes.remove(random.choice((barbeirosNomes)))

barbeirosList = [simpy.Resource(env, capacity=1) for i in range(len(barbeirosNomes))]
barbeirosDict = dict(zip(barbeirosNomes, barbeirosList))

# dicionário para armazenar o número de clientes em fila de favoritos
filaDict = {k:0 for k in barbeirosNomes}

# cria um FilterStore para armazenar os barbeiros
barbeariaStore = simpy.FilterStore(env, capacity=3)
barbeariaStore.items = barbeirosNomes

# inicia processo de chegadas de clientes
env.process(chegadaClientes(env, barbeariaStore))
env.run(until = 30)
```

Quando executado por apenas 30 minutos, o modelo anterior fornece:

```
 13.1 Cliente 1 chega.          Sem preferência.
 13.1 Cliente 1 inicia.         Barbeiro A ocupado.     Tempo de fila: 0.0
 14.3 Cliente 2 chega.          Barbeiro A.
 26.6 Cliente 1 termina.        Barbeiro A liberado.
 26.6 Cliente 2 inicia.         Barbeiro A ocupado.     Tempo de fila: 12.3
 29.6 Cliente 3 chega.          Sem preferência.
 29.6 Cliente 3 inicia.         Barbeiro B ocupado.     Tempo de fila: 0.0
```

Teste seus conhecimentos

1. Considere que a barbearia opera 6 horas por dia. Acrescente ao seu modelo às estatísticas de clientes atendidos, clientes que desistiram (e por qual razão), ocupação dos barbeiros e

tempo médio de espera em fila por barbeiro.

2. Dada a presente demanda da barbearia, quantos barbeiros devem estar trabalhando, caso o proprietário pretenda que o tempo médio de espera em fila seja inferior a 15 minutos?
3. Refaça o exemplo do `PriorityStore` da seção anterior, armazenando não mais o nome do barbeiro, mas o seu respectivo `resource.`

Enchendo ou esvaziando caixas, tanques ou objetos com `Container()`

Um tipo especial de recurso no SimPy é o `container`. Intuitivamente, um `container` seria um taque ou caixa em que se armazenam coisas. Você pode encher ou esvaziar em quantidade, como se fosse um tanque de água ou uma caixa de laranjas.

A sua utilização é bastante simples, por exemplo, podemos modelar um tanque de 100 unidades de capacidade ($$m^3$$, por exemplo), com um estoque inicial de 50 unidades, por meio do seguinte código:

```
import simpy

env = simpy.Environment()
# cria um tanque de 100 m3 de capacidade, com 50 m3 no início da simulação
tanque = simpy.Container(env, capacity=100, init=50)
```

O `container` possui três comandos importantes:

- Para encher: `tanque.put(quantidade)`
- Para esvaziar: `tanque.get(quantidade)`
- Para obter o nível atual: `tanque.level`

Enchendo o meu container `yield meuContainer.put(quantidade)`

Considere que um posto de gasolina possui um tanque com capacidade de 100 $$m^3$$ (ou 100.000 litros) de combustível e que o tanque já contém 50 $$m^3$$ armazenado.

Criaremos uma função, `enchimentoTanque`, que enche o tanque com 50 $$m^3$$ sempre que um novo caminhão de reabastecimento de combustível chega ao posto:

```
import simpy
import random

TANQUE_CAMINHAO = 50       # capacidade de abastecimento do caminhão

def enchimentoTanque(env, qtd, tanque):
    # enche o tanque
    print("%d Novo caminhão com %4.1f m3.\t Nível atual: %5.1f m3"
            % (env.now, qtd, tanque.level))
    yield tanque.put(qtd)
    print("%d Tanque enchido com %4.1f m3.\t Nível atual: %5.1f m3"
            % (env.now, qtd, tanque.level))

random.seed(150)
env = simpy.Environment()
#cria um tanque de 100 m3, com 50 m3 no início da simulação
tanque = simpy.Container(env, capacity=100, init=50)
env.process(enchimentoTanque(env, TANQUE_CAMINHAO, tanque))

env.run(until = 500)
```

A saída do programa é bastante simples, afinal o processo de enchimento do tanque é executado apenas uma vez:

```
0 Novo caminhão de combustível com 50.0 m3. Nível atual:  50.0 m3
0 Tanque enchido com 50.0 m3. Nível atual: 100.0 m3
```

Se você iniciar o tanque do posto a sua plena capacidade (100 $$m^3$$), o caminhão tentará abastecer, mas não conseguirá por falta de espaço, virtualmente aguardando espaço no tanque na linha:

```
yield tanque.put(qtd)
```

Dentro da função `enchimentoTanque`.

Esvaziando o meu container: `yield meuContainer.get(quantidade)`

Considere que o posto atende automóveis que chegam em intervalos constantes de 5 minutos entre si e que cada veículo abastece 100 litros ou 0,10 $$m^3$$.

Partindo do modelo anterior, vamos criar duas funções: uma para gerar os veículos e outra para transferir o combustível do tanque para o veículo.
Uma possível máscara para o modelo seria:

```
import simpy
import random

TANQUE_CAMINHAO = 50        # capacidade de abastecimento do caminhão
TANQUE_VEICULO = 0.10       # capacidade do veículo
TEMPO_CHEGADAS = 5          # tempo entre chegadas sucessivas de veículos

def chegadasVeiculos(env, tanque):
    # gera chegadas de veículos por produto

def esvaziamentoTanque(env, qtd, tanque):
    # esvazia o tanque

def enchimentoTanque(env, qtd, tanque):
    # enche o tanque
    print("%d Novo caminhão com %4.1f m3.\t Nível atual: %5.1f m3"
            % (env.now, qtd, tanque.level))
    yield tanque.put(qtd)
    print("%d Tanque enchido com %4.1f m3.\t Nível atual: %5.1f m3"
            % (env.now, qtd, tanque.level))

random.seed(150)
env = simpy.Environment()
# cria um tanque de 100 m3, com 50 m3 no início da simulação
tanque = simpy.Container(env, capacity=100, init=50)
env.process(chegadasVeiculos(env, tanque))

env.run(until = 20)
```

A função `chegadasVeiculos` gera os veículos que buscam abastecimento no posto e que, a seguir, chamam a função `esvaziamentoTanque` responsável por provocar o esvaziamento do tanque do posto na quantidade desejada pelo veículo:

```
def chegadasVeiculos(env, tanque):
    # gera chegadas de veículos por produto
    while True:
        yield env.timeout(TEMPO_CHEGADAS)
        # carrega veículo
        env.process(esvaziamentoTanque(env, TANQUE_VEICULO, tanque))
```

A função que representa o processo de esvaziamento do tanque é semelhante a de enchimento da seção anterior, a menos da opção `get(qtd),` que retira a quantidade `qtd` do `container tanque:`

```
def esvaziamentoTanque(env, qtd, tanque):
    # esvazia o tanque
    print("%d Novo veículo de %3.2f m3.\t Nível atual: %5.1f m3"
            % (env.now, qtd, tanque.level))
    yield tanque.get(qtd)
    print("%d Veículo atendido de %3.2f.\t Nível atual: %5.1f m3"
            % (env.now, qtd, tanque.level))
```

O modelo de simulação completo do posto de gasolina fica:

```
import simpy
import random

TANQUE_CAMINHAO = 50        # capacidade de abastecimento do caminhão
TANQUE_VEICULO = 0.10       # capacidade do veículo
TEMPO_CHEGADAS = 5          # tempo entre chegadas sucessivas de veículos

def chegadasVeiculos(env, tanque):
    # gera chegadas de veículos por produto
    while True:
        yield env.timeout(TEMPO_CHEGADAS)
        # carrega veículo
        env.process(esvaziamentoTanque(env, TANQUE_VEICULO, tanque))

def esvaziamentoTanque(env, qtd, tanque):
    # esvazia o tanque
    print("%d Novo veículo de %3.2f m3.\t Nível atual: %5.1f m3"
            % (env.now, qtd, tanque.level))
    yield tanque.get(qtd)
    print("%d Veículo atendido de %3.2f m3.\t Nível atual: %5.1f m3"
            % (env.now, qtd, tanque.level))

def enchimentoTanque(env, qtd, tanque):
    # enche o tanque
    print("%d Novo caminhão com %4.1f m3.\t Nível atual: %5.1f m3"
            % (env.now, qtd, tanque.level))
    yield tanque.put(qtd)
    print("%d Tanque enchido com %4.1f m3.\t Nível atual: %5.1f m3"
            % (env.now, qtd, tanque.level))

random.seed(150)
env = simpy.Environment()
#cria um tanque de 100 m3, com 50 m3 no início da simulação
tanque = simpy.Container(env, capacity=100, init=50)
env.process(chegadasVeiculos(env, tanque))

env.run(until = 200)
```

Quando por 200 minutos, o modelo anterior fornece como saída:

```
5 Novo veículo de 0.10 m3.       Nível atual:  50.0 m3
5 Veículo atendido de 0.10 m3.   Nível atual:  49.9 m3
10 Novo veículo de 0.10 m3.      Nível atual:  49.9 m3
10 Veículo atendido de 0.10.     Nível atual:  49.8 m3
15 Novo veículo de 0.10 m3.      Nível atual:  49.8 m3
15 Veículo atendido de 0.10 m3.  Nível atual:  49.7 m3
```

Criando um sensor para o nível atual do `container`

Ainda no exemplo do posto, vamos chamar um caminhão de reabastecimento sempre que o tanque atingir o nível de 50 $$m^3$$. Para isso, criaremos uma função `sensorTanque` capaz de reconhecer o instante exato em que o nível do tanque abaixou do valor desejado e, portanto, deve ser enviado um caminhão de reabastecimento.

Inicialmente, para identificar se o nível do tanque abaixou além no nível mínimo, precisamos verificar qual o nível atual. Contudo, esse processo de verificação não é contínuo no tempo e deve ter o seu intervalo entre verificações pré-definido no modelo.

Assim, são necessários dois parâmetros: um para o nível mínimo e outro para o intervalo entre verificações do nível do tanque. Uma possível codificação para a função `sensorTanque` seria:

```
NIVEL_MINIMO = 50           # nível mínimo de reabastecimento do tanque
TEMPO_CONTROLE = 1          # tempo entre verificações do nível do tanque

def sensorTanque(env, tanque):
    # quando o tanque baixar se certo nível, dispara o enchimento
    while True:
        if tanque.level <= NIVEL_MINIMO:
            # dispara pedido de enchimento
            yield env.process(enchimentoTanque(env, TANQUE_CAMINHAO, tanque))
        # aguarda um tempo para fazer a nova chegagem do nível do tanque
        yield env.timeout(TEMPO_CONTROLE)
```

A função `sensorTanque` é um laço infinito `(while True)` que a cada 1 minuto (configurável na constante `TEMPO_CONTROLE)` verifica se o nível atual do tanque está abaixo ou igual ao nível mínimo (configurável na constante `NIVEL_MINIMO).
O modelo completo com a implentação do sensor fica:

```
import simpy
import random

TANQUE_CAMINHAO = 50        # capacidade de abastecimento do caminhão
TANQUE_VEICULO = 0.10       # capacidade do veículo
TEMPO_CHEGADAS = 5          # tempo entre chegadas sucessivas de veículos
NIVEL_MINIMO = 50           # nível mínimo de reabastecimento do tanque
TEMPO_CONTROLE = 1          # tempo entre verificações do nível do tanque

def sensorTanque(env, tanque):
    # quando o tanque baixar se certo nível, dispara o enchimento
    while True:
        if tanque.level <= NIVEL_MINIMO:
            # dispara pedido de enchimento
            yield env.process(enchimentoTanque(env, TANQUE_CAMINHAO, tanque))
        # aguarda um tempo para fazer a nova chegagem do nível do tanque
        yield env.timeout(TEMPO_CONTROLE)

def chegadasVeiculos(env, tanque):
    # gera chegadas de veículos por produto
    while True:
        yield env.timeout(TEMPO_CHEGADAS)
        # carrega veículo
        env.process(esvaziamentoTanque(env, TANQUE_VEICULO, tanque))

def esvaziamentoTanque(env, qtd, tanque):
    # esvazia o tanque
    print("%d Novo veículo de %3.2f m3.\t Nível atual: %5.1f m3"
            % (env.now, qtd, tanque.level))
    yield tanque.get(qtd)
    print("%d Veículo atendido de %3.2f m3.\t Nível atual: %5.1f m3"
            % (env.now, qtd, tanque.level))

def enchimentoTanque(env, qtd, tanque):
    # enche o tanque
    print("%d Novo caminhão com %4.1f m3.\t Nível atual: %5.1f m3"
            % (env.now, qtd, tanque.level))
    yield tanque.put(qtd)
    print("%d Tanque enchido com %4.1f m3.\t Nível atual: %5.1f m3"
            % (env.now, qtd, tanque.level))

random.seed(150)
env = simpy.Environment()
#cria um tanque de 100 m3, com 50 m3 no início da simulação
tanque = simpy.Container(env, capacity=100, init=50)
env.process(chegadasVeiculos(env, tanque))
env.process(sensorTanque(env, tanque))

env.run(until = 20)
```

Note a criação do processo do `sensorTanque` na penúltima linha do programa:

```
env.process(sensorTanque(env, tanque))
```

Este processo garante que o sensor estará operante ao longo de toda a simulação. Quando executado, o programa anterior retorna:

```
0 Novo caminhão com 50.0 m3.     Nível atual:  50.0 m3
0 Tanque enchido com 50.0 m3.    Nível atual: 100.0 m3
5 Novo veículo de 0.10 m3.       Nível atual: 100.0 m3
5 Veículo atendido de 0.10 m3.   Nível atual:  99.9 m3
10 Novo veículo de 0.10 m3.      Nível atual:  99.9 m3
10 Veículo atendido de 0.10 m3.  Nível atual:  99.8 m3
15 Novo veículo de 0.10 m3.      Nível atual:  99.8 m3
15 Veículo atendido de 0.10 m3.  Nível atual:  99.7 m3
```

> Observação 1: Note que o enchimento ou esvaziamento dos tanques é instantâneo, isto é: não existe nenhuma taxa de enchimento ou esvaziamento associada aos processos. Cabe ao programador modelar situações em que a taxa de transferência é relevante (veja o Desafio 17, a seguir).
>
> Observação 2: O tanque pode ser esvaziado ou enchido simultaneamente. Novamente cabe ao programador modelar a situação em que isto não se verifica (veja o Desafio 18, a seguir).

Conceitos desta seção

Conteúdo	Descrição
`meuContainer = simpy.Container(env, capacity=capacity, init=init`	cria um *container* `meuContainer` com capacidade `capacity` e quantidade inicial de `init`
`yield meuContainer.put(quantidade)`	adiciona uma dada `quantidade` ao `meuContainer` , se houver espaço suficiente, caso contrário aguarda até que o espaço esteja disponível
`yield meuContainer.get(quantidade)`	retira uma dada `quantidade` ao `meuContainer` , se houver quantidade suficiente, caso contrário aguarda até que a quantidade esteja disponível
`meuContainer.level`	retorna a quantidade disponível atualmente em `meuContainer`

Desafios

> **Desafio 17:** considere, no exemplo do posto, que a taxa de enchimento do tanque é de 1 litro/min e a de esvaziamento é de 2 litros/min. Altere o modelo para que ele incorpore os tempos de enchimento e esvaziamento, bem como forneça o tempo que o veículo aguardou na fila por atendimento.
>
> **Desafio 18:** continuando o exemplo, modifique o modelo de modo que ele represente a situação em que o tanque não pode ser enchido e esvaziado simultaneamente.

Criando lotes (ou agrupando) entidades durante a simulação

Uma situação bastante comum em modelos de simulação é o agrupamento de entidades em lotes ou o seu oposto: o desmembramento de um lote em diversas entidades separadas. É usual em softwares de simulação proprietários existir um comando (ou bloco) específico para isso. Por exemplo, o Arena possui o "Batch/Separate", o Simul8 o "Batching" etc.

Vamos partir de um exemplo simples, em que uma célula de produção deve realizar a tarefa de montagem de um certo componente a partir do encaixe de uma peça A com duas peças B. O operador da célula leva em média 5 minutos para montar o componente, segundo uma distribuição normal com desvio padrão de 1 minuto. Os processos de chegadas dos lotes A e B são distintos entre si, com tempos entre chegadas sucessivas uniformemente distribuídos no intervalo entre 40 a 60 minutos.

Uma tática para agrupamento de lotes utilizando o `Container`

Uma maneira de resolver o problema é criar um `Container` de estoque temporário para cada peça. Assim, criamos dois estoques, respectivamente para as peças A e B, de modo que o componente só poderá iniciar sua montagem se cada estoque contiver ao menos o número de peças necessárias para sua montagem.

Comecemos criando uma possível máscara para o problema:

```
import simpy
import random

TEMPO_CHEGADAS = [40, 50]       # intervalo entre chegadas de peças
TEMPO_MONTAGEM = [5, 1]         # tempo de montagem do componente
componentesProntos = 0          # variável para o total de componentes produzidos

def chegadaPecas(env, pecasContainerDict, tipo, tamLote):
    # gera lotes de pecas em intervalos uniformemente distribuídos
    # encaminha para o estoque
    pass

def montagem(env, pecasContainerDict, numA, numB):
    # montagem do componente
    global componentesProntos
    pass

random.seed(100)
env = simpy.Environment()

# cria estoques de peças
pecasContainerDict = {}
pecasContainerDict['A'] = simpy.Container(env)
pecasContainerDict['B'] = simpy.Container(env)

# inicia processos de chegadas de pecas
env.process(chegadaPecas(env, pecasContainerDict, 'A', 10))
env.process(chegadaPecas(env, pecasContainerDict, 'B', 10))

# inicia processo de montagem
env.process(montagem(env, pecasContainerDict, 1, 2))
env.run(until=80)
```

Na máscara anterior, foram criadas duas funções: `chegaPecas`, que gera os lotes de peças A e B e armazena nos respectivos estoques e `montagem`, que retira as peças do estoque e montam o componente.

Note que criei um dicionário no Python: `pecasContainerDict`, para armazenar o `Container` de cada peça:

```
# cria estoques de peças
pecasContainerDict = {}
pecasContainerDict['A'] = simpy.Container(env)
pecasContainerDict['B'] = simpy.Container(env)
```

A função de geração de peças de fato, gera lotes e armazena dentro do Container o número de peças do lote:

```
def chegadaPecas(env, pecasContainerDict, tipo, tamLote):
    # gera lotes de pecas em intervalos uniformemente distribuídos
    # encaminha para o estoque
    while True:
        pecasContainerDict[tipo].put(tamLote)
        print("%5.1f Chegada de lote\t%s\tPeças: %i"
                %(env.now, tipo, tamLote))
        yield env.timeout(random.uniform(*TEMPO_CHEGADAS))
```

Note que, diferentemente das funções de geração de entidades criadas nas seções anteriores deste livro, a função `chegadaPecas` não encaminha a entidade criada para uma nova função, iniciando um novo processo (de atendimento, por exemplo). A função apenas armazena uma certa quantidade de peças, `tamLote,` dentro do respectivo `Container` na linha:

```
pecasContainerDict[tipo].put(tamLote)
```

O processo de montagem também recorre ao artifício de um laço infinito, pois, basicamente, representa uma operação que está sempre pronta para executar a montagem, desde que existam o número de peças mínimas à disposição nos respectivos estoques:

```
def montagem(env, pecasContainerDict, numA, numB):
    # montagem do componente
    global componentesProntos
    while True:
        # marca o instante em que a célula esta livre para a montagem
        chegada = env.now
        yield pecasContainerDict['A'].get(numA)
        yield pecasContainerDict['B'].get(numB)
        # armazena o tempo de espera por peças e inicia a montagem
        espera = env.now - chegada
        print("%5.1f Inicia montagem\tEstoque A: %i\tEstoque B: %i\tEspera: %4.1f"
                %(env.now, pecasContainerDict['A'].level, pecasContainerDict['B'].level, esper
a))
        yield env.timeout(random.normalvariate(*TEMPO_MONTAGEM))
        # acumula componente montado
        componentesProntos += 1
        print("%5.1f Fim da montagem\tEstoque A: %i\tEstoque B: %i\tComponentes: %i\t"
            %(env.now, pecasContainerDict['A'].level, pecasContainerDict['B'].level,
              componentesProntos))
```

A parte central da função anterior é garantir que o processo só possa se iniciar caso existam peças suficientes para o componente final. Isto é garantido pelo comando `get` aplicado a cada `Container` de peças necessárias:

```
        yield pecasContainerDict['A'].get(numA)
        yield pecasContainerDict['B'].get(numB)
```

Quando executado, o modelo completo fornece como saída:

```
  0.0 Chegada de lote   tipo A: +10 peças
  0.0 Chegada de lote   tipo B: +10 peças
  0.0 Inicia montagem   Estoque A: 9    Estoque B: 8    Espera:  0.0
  6.6 Fim da montagem   Estoque A: 9    Estoque B: 8    Componentes: 1
  6.6 Inicia montagem   Estoque A: 8    Estoque B: 6    Espera:  0.0
 12.3 Fim da montagem   Estoque A: 8    Estoque B: 6    Componentes: 2
 12.3 Inicia montagem   Estoque A: 7    Estoque B: 4    Espera:  0.0
 18.4 Fim da montagem   Estoque A: 7    Estoque B: 4    Componentes: 3
 18.4 Inicia montagem   Estoque A: 6    Estoque B: 2    Espera:  0.0
 22.1 Fim da montagem   Estoque A: 6    Estoque B: 2    Componentes: 4
 22.1 Inicia montagem   Estoque A: 5    Estoque B: 0    Espera:  0.0
 28.2 Fim da montagem   Estoque A: 5    Estoque B: 0    Componentes: 5
 41.5 Chegada de lote   tipo A: +10 peças
 44.5 Chegada de lote   tipo B: +10 peças
 44.5 Inicia montagem   Estoque A: 14   Estoque B: 8    Espera: 16.3
 48.9 Fim da montagem   Estoque A: 14   Estoque B: 8    Componentes: 6
 48.9 Inicia montagem   Estoque A: 13   Estoque B: 6    Espera:  0.0
 53.1 Fim da montagem   Estoque A: 13   Estoque B: 6    Componentes: 7
 53.1 Inicia montagem   Estoque A: 12   Estoque B: 4    Espera:  0.0
 59.1 Fim da montagem   Estoque A: 12   Estoque B: 4    Componentes: 8
 59.1 Inicia montagem   Estoque A: 11   Estoque B: 2    Espera:  0.0
 64.7 Fim da montagem   Estoque A: 11   Estoque B: 2    Componentes: 9
 64.7 Inicia montagem   Estoque A: 10   Estoque B: 0    Espera:  0.0
 70.0 Fim da montagem   Estoque A: 10   Estoque B: 0    Componentes: 10
```

O que o leitor deve ter achado interessante é o modo passivo da função `montagem` que, por meio de um laço infinito `while True` aguarda o aparecimento de peças suficientes nos estoques para iniciar a montagem. Interessante também é notar que não alocamos recursos para a operação e isso significa que o modelo de simulação atual não permite a montagem simultânea de componentes (veja o tópico "Teste seus conhecimentos" na próxima seção).

Agrupando lotes por atributo da entidade utilizando o `FilterStore`

Outra situação bastante comum em modelos de simulação é quando precisamos agrupar entidades por atributo. Por exemplo, os componentes anteriores são de duas cores: brancos ou verdes, de modo que a célula de montagem agora deve pegar peças A e B com as cores corretas.

Como agora existe um atributo (no caso, cor) que diferencia uma peça da outra, precisaremos de um `FilterStore`, para garantir a escolha certa da peça no estoque. Contudo, devemos lembrar que o `FilterStore`, diferentemente do `Container`, não permite que se armazene ou retire múltiplos objetos ao mesmo tempo. O comando `put` (ou mesmo o `get),` é limitado a um objeto por vez. Por fim, a montagem do componente agora é pelo atributo "cor", o que significa que a função `montagem` deve ser chamada uma vez para cada valor do atributo (no caso duas vezes: "branco" ou "verde").

De modo semelhante ao exemplo anterior, uma máscara para o problema seria:

```
import simpy
import random

TEMPO_CHEGADAS = [40, 50]       # intervalo entre chegadas de peças
TEMPO_MONTAGEM = [5, 1]         # tempo de montagem do componente
componentesProntos = 0          # variável para o total de componentes produzidos

def chegadaPecas(env, pecasFilterStoreDict, tipo, tamLote):
    # gera lotes de pecas em intervalos uniformemente distribuídos
    # sorteia a cor das peças
    # coloca um número tamLote de peças dentro do FilterStore
    pass

def montagem(env, pecasFilterStoreDict, numA, numB, cor):
    # montagem do componente
    global componentesProntos
    pass

random.seed(100)
env = simpy.Environment()

# cria um dicionário para armazenar os FilterStore
pecasFilterStoreDict = {}
pecasFilterStoreDict['A'] = simpy.FilterStore(env)
pecasFilterStoreDict['B'] = simpy.FilterStore(env)

# inicia processos de chegadas de pecas
env.process(chegadaPecas(env, pecasFilterStoreDict, 'A', 10))
env.process(chegadaPecas(env, pecasFilterStoreDict, 'B', 10))

# inicia processos de montagem de pecas
env.process(montagem(env, pecasFilterStoreDict, 1, 2, 'branco'))
env.process(montagem(env, pecasFilterStoreDict, 1, 2, 'verde'))

env.run(until=80)
```

Note que foi criado um dicionário `pecasFilterStore` armazena um `FilterStore` para cada tipo de peça.
Vamos agora construir a função `chegadaPecas`, considerando que ela deve sortear a cor do lote de peças e enviar todas as peças do lote (uma por vez) para o respectivo `FilterStore.`

Para sortear a cor do lote, uma opção é utilizar o comando random.choice, enquanto o envio de múltiplas peças para o `FilterStore` pode ser feito por um laço `for,` como mostra o código a seguir:

```
def chegadaPecas(env, pecasFilterStoreDict, tipo, tamLote):
    # gera lotes de pecas em intervalos uniformemente distribuídos
    while True:
        # sorteia a cor das peças
        cor = random.choice(("branco", "verde"))
        # coloca um número tamLote de peças dentro do FilterStore
        for i in range(tamLote):
            yield pecasFilterStoreDict[tipo].put(cor)
        print("%5.1f Chegada de lote\ttipo: %s\t\tCor: %s"
                %(env.now, tipo, cor))
        yield env.timeout(random.uniform(*TEMPO_CHEGADAS))
```

A função `montagem,` de modo semelhante a função anterior, é formada por um laço infinito do tipo `while... True`, mas deve retirar múltiplas peças de cada estoque, respeitando o atributo "cor". O código a seguir, soluciona este problema novamente com laços do tipo `for` e uma função anônima para buscar a cor correta da peça dentro do `FilterStore:`

```
def montagem(env, pecasFilterStoreDict, numA, numB, cor):
    # montagem do componente
    global componentesProntos

    while True:
        # marca o instante em que a célula está livre para a montagem
        chegada = env.now
        for i in range(numA):
            yield pecasFilterStoreDict['A'].get(lambda c: c==cor)
        for i in range(numB):
            yield pecasFilterStoreDict['B'].get(lambda c: c==cor)
        # armazena o tempo de espera por peças e inicia a montagem
        espera = env.now - chegada
        print("%5.1f Inicia montagem\tCor: %s\tEspera: %4.1f"
                %(env.now, cor, espera))
        yield env.timeout(random.normalvariate(*TEMPO_MONTAGEM))
        # acumula componente montado
        componentesProntos += 1
        print("%5.1f Fim da montagem\tCor: %s\tComponentes: %i\tEstoque A: %i\tEstoque B: %i"
            %(env.now, cor, componentesProntos,  len(pecasFilterStoreDict['A'].items),
              len(pecasFilterStoreDict['B'].items)))
```

Dois pontos merecem destaque na função anterior:

1. A função `montagem`, deve ser chamada duas vezes na inicialização da simulação, uma para cada cor. Isto significa que nossa implementação permite a montagem simultânea de peças de cores diferentes. Caso seja necessário contornar este problema, basta a criação de um recurso "montador" (veja o tópico "Teste seus conhecimentos" na próxima seção";
2. Na última linha, para contabilizar o total de peças ainda em estoque, como o `FilterStore` não possui um método `.level,` utilizou-se a função `len()` aplicada a todos os `items` do `FilterStore.`

Quando executado por apenas 80 minutos, o programa anterior fornece como saída:

```
 0.0 Chegada de lote   tipo: A         Cor: branco
 0.0 Chegada de lote   tipo: B         Cor: verde
44.5 Inicia montagem   Cor: verde      Espera: 44.5
44.5 Chegada de lote   tipo: A         Cor: verde
47.7 Inicia montagem   Cor: branco     Espera: 47.7
47.7 Chegada de lote   tipo: B         Cor: branco
50.3 Fim da montagem   Cor: verde      Componentes: 1  Estoque A: 18   Estoque B: 16
50.3 Inicia montagem   Cor: verde      Espera:  0.0
51.4 Fim da montagem   Cor: branco     Componentes: 2  Estoque A: 17   Estoque B: 14
51.4 Inicia montagem   Cor: branco     Espera:  0.0
54.8 Fim da montagem   Cor: verde      Componentes: 3  Estoque A: 16   Estoque B: 12
54.8 Inicia montagem   Cor: verde      Espera:  0.0
57.0 Fim da montagem   Cor: branco     Componentes: 4  Estoque A: 15   Estoque B: 10
57.0 Inicia montagem   Cor: branco     Espera:  0.0
59.2 Fim da montagem   Cor: verde      Componentes: 5  Estoque A: 14   Estoque B: 8
59.2 Inicia montagem   Cor: verde      Espera:  0.0
61.3 Fim da montagem   Cor: branco     Componentes: 6  Estoque A: 13   Estoque B: 6
61.3 Inicia montagem   Cor: branco     Espera:  0.0
64.6 Fim da montagem   Cor: branco     Componentes: 7  Estoque A: 12   Estoque B: 4
64.6 Inicia montagem   Cor: branco     Espera:  0.0
65.9 Fim da montagem   Cor: verde      Componentes: 8  Estoque A: 11   Estoque B: 2
65.9 Inicia montagem   Cor: verde      Espera:  0.0
68.8 Fim da montagem   Cor: branco     Componentes: 9  Estoque A: 10   Estoque B: 0
71.1 Fim da montagem   Cor: verde      Componentes: 10 Estoque A: 9    Estoque B: 0
```

Naturalmente, existem outras soluções, mas optei por um caminho que mostrasse algumas limitações para um problema bastante comum em modelos de simulação.

> **Desafio 19**: Considere, no primeiro exemplo, que o componente possui mais duas partes, C e D que devem ser previamente montadas entre si para, a seguir, serem encaixadas nas peças A e B. Os tempos de montagem são todos semelhantes. (Dica: generalize a função `montagem` apresentada no exemplo).
>
> **Desafio 20**: Nos exemplos anteriores, os processos de montagem são paralelos. Considere que existe apenas um montador compartilhado para todos processos. Generalize a função montagem do desafio anterior, de modo que ela receba como parâmetro o respectivo recurso utilizado no processo.

Solução dos desafios 19 e 20

> **Desafio 19**: Considere, no primeiro exemplo, que o componente possui mais duas partes, C e D que devem ser previamente montadas entre si para, a seguir, serem encaixadas nas peças A e B. Os tempos de montagem são todos semlhantes. (Dica: generalize a função `montagem` apresentada no exemplo).

Para este desafio, não precisamos alterar o processo de chegadas, apenas vamos chamá-lo mais vezes, para as peças tipo A, B, C e D.
Inicialmente, portanto, precisamos criar um `Container` para cada etapa da montagem, bem como chamar a função de geração de lotes para as 4 partes iniciais do componente:

```
random.seed(100)
env = simpy.Environment()

# cria estoques de peças
tipoList = ['A', 'B', 'C', 'D', 'AB', 'CD', 'ABCD']
pecasContainerDict = {}
for tipo in tipoList:
    pecasContainerDict[tipo] = simpy.Container(env)

# inicia processos de chegadas de pecas
for i in "ABCD":
    env.process(chegadaPecas(env, pecasContainerDict, i, 10))
```

O bacana desse desafio é explorar o potencial de *generalidade* do SimPy, afinal, os processos de montagem são semelhantes, o que muda apenas é o quais as peças estão sendo unidas.
Imagine por um momento, que podemos querer unir 3 ou mais peças. A lógica em si não é muito diferente daquela feita na seção anterior, muda apenas a necessidade de se lidar com um número de peças diferentes. Assim, para *generalizar* a função `montagem`, proponho utilizar o operador `**kwargs` que envia um conjunto de parâmetros para uma função como um dicionário.
A ideia aqui é chamar o processo de montagem de maneira flexível, por exemplo:

```
# inicia processos de montagem
env.process(montagem(env, pecasContainerDict, 'AB', A=1, B=2))
env.process(montagem(env, pecasContainerDict, 'CD', C=1, D=2))
env.process(montagem(env, pecasContainerDict, 'ABCD', AB=1, CD=1))
```

Estamos chamando a função montagem para diferentes configurações de peças a serem montadas. Por exemplo, a primeira linha representa uma montagem de uma peça A com duas B; a segunda uma peça C e duas D e a terceira linha representa uma peça do tipo AB com uma do tipo CB, formando o componente ABCD.

O código a seguir, apresenta uma solução que dá razoável generalidade para a função `montagem`:

```
def montagem(env, pecasContainerDict, keyOut, **kwargs):
    # montagem do componente

    while True:
        # marca o instante em que a célula esta livre para a montagem
        chegada = env.now

        # pega uma peça de cada um dos items do dicionário kwargs
        for key, value in kwargs.items():
            yield pecasContainerDict[key].get(value)

        # armazena o tempo de espera por peças e inicia a montagem
        espera = env.now - chegada
        print("%5.1f Inicia montagem\t%s\tEspera: %4.1f" %(env.now, keyOut, espera))
        yield env.timeout(random.normalvariate(*TEMPO_MONTAGEM))
        # acumula componente montado no Container de saída keyOut
        yield pecasContainerDict[keyOut].put(1)
        print("%5.1f Fim da montagem\t%s\tEstoque: %i"
                %(env.now, keyOut, pecasContainerDict[keyOut].level))
```

No código anterior, os componentes montados são colocados no `Container` definido no parâmetro `keyOut` e, note como a parte relativa ao parâmetro `**kwargs` é tratada como um mero dicionário pelo Python.

O modelo de simulação completo do desafio 19 fica:

```
import simpy
import random

TEMPO_CHEGADAS = [40, 50]       # intervalo entre chegadas de peças
TEMPO_MONTAGEM = [5, 1]         # tempo de montagem do componente

def chegadaPecas(env, pecasContainerDict, tipo, tamLote):
    # gera lotes de pecas em intervalos uniformemente distribuídos
    # encaminha para o estoque
    while True:
        pecasContainerDict[tipo].put(tamLote)
        print("%5.1f Chegada de lote\t%s\tPeças: %i"
                %(env.now, tipo, tamLote))
        yield env.timeout(random.uniform(*TEMPO_CHEGADAS))


def montagem(env, pecasContainerDict, keyOut, **kwargs):
    # montagem do componente

    while True:
        # marca o instante em que a célula esta livre para a montagem
        chegada = env.now

        # pega uma peça de cada um dos items do dicionário kwargs
        for key, value in kwargs.items():
            yield pecasContainerDict[key].get(value)

        # armazena o tempo de espera por peças e inicia a montagem
        espera = env.now - chegada
        print("%5.1f Inicia montagem\t%s\tEspera: %4.1f" %(env.now, keyOut, espera))
        yield env.timeout(random.normalvariate(*TEMPO_MONTAGEM))
        # acumula componente montado no Container de saída keyOut
        yield pecasContainerDict[keyOut].put(1)
        print("%5.1f Fim da montagem\t%s\tEstoque: %i"
                %(env.now, keyOut, pecasContainerDict[keyOut].level))

random.seed(100)
env = simpy.Environment()

# cria estoques de peças
tipoList = ['A', 'B', 'C', 'D', 'AB', 'CD', 'ABCD']
pecasContainerDict = {}
for tipo in tipoList:
    pecasContainerDict[tipo] = simpy.Container(env)

# inicia processos de chegadas de pecas
for i in "ABCD":
    env.process(chegadaPecas(env, pecasContainerDict, i, 10))

# inicia processos de montagem
env.process(montagem(env, pecasContainerDict, 'AB', A=1, B=2))
env.process(montagem(env, pecasContainerDict, 'CD', C=1, D=2))
env.process(montagem(env, pecasContainerDict, 'ABCD', AB=1, CD=1))

env.run(until=40)
```

Quando executado por apenas 40 minutos, o modelo anterior fornece como saída:

```
  0.0 Chegada de lote   A       Peças: 10
  0.0 Chegada de lote   B       Peças: 10
  0.0 Chegada de lote   C       Peças: 10
  0.0 Chegada de lote   D       Peças: 10
  0.0 Inicia montagem   AB      Espera:  0.0
  0.0 Inicia montagem   CD      Espera:  0.0
  5.7 Fim da montagem   AB      Estoque: 1
  5.7 Inicia montagem   AB      Espera:  0.0
  6.1 Fim da montagem   CD      Estoque: 0
  6.1 Inicia montagem   ABCD    Espera:  6.1
  6.1 Inicia montagem   CD      Espera:  0.0
  9.4 Fim da montagem   AB      Estoque: 1
  9.4 Inicia montagem   AB      Espera:  0.0
  9.7 Fim da montagem   CD      Estoque: 1
  9.7 Inicia montagem   CD      Espera:  0.0
 12.3 Fim da montagem   ABCD    Estoque: 1
 12.3 Inicia montagem   ABCD    Espera:  0.0
 13.8 Fim da montagem   AB      Estoque: 1
 13.8 Inicia montagem   AB      Espera:  0.0
 13.9 Fim da montagem   CD      Estoque: 1
 13.9 Inicia montagem   CD      Espera:  0.0
 18.2 Fim da montagem   ABCD    Estoque: 2
 18.2 Inicia montagem   ABCD    Espera:  0.0
 19.2 Fim da montagem   CD      Estoque: 1
 19.2 Inicia montagem   CD      Espera:  0.0
 19.4 Fim da montagem   AB      Estoque: 1
 19.4 Inicia montagem   AB      Espera:  0.0
 21.8 Fim da montagem   ABCD    Estoque: 3
 21.8 Inicia montagem   ABCD    Espera:  0.0
 25.2 Fim da montagem   CD      Estoque: 1
 25.3 Fim da montagem   AB      Estoque: 1
 26.6 Fim da montagem   ABCD    Estoque: 4
 26.6 Inicia montagem   ABCD    Espera:  0.0
 34.5 Fim da montagem   ABCD    Estoque: 5
```

> **Desafio 20**: Nos exemplos anteriores, os processos de montagem são paralelos. Considere que existe apenas um montador compartilhado para todos processos. Generalize a função montagem do desafio anterior, de modo que ela receba como parâmetro o respectivo recurso utilizado no processo.
> Neste desafio precisamos apenas garantir que a função `montagem` receba como parâmetro o respectivo recurso a ser utilizado no processo:

```
def montagem(env, pecasContainerDict, montador, keyOut, **kwargs):
    # montagem do componente

    while True:
        # marca o instante em que a célula esta livre para a montagem
        with montador.request() as req:
            # ocupa o recurso montador
            yield req
            chegada = env.now
            # pega uma peça de cada um dos items do dicionário kwargs
            for key, value in kwargs.items():
                yield pecasContainerDict[key].get(value)

            # armazena o tempo de espera por peças e inicia a montagem
            espera = env.now - chegada
            print("%5.1f Inicia montagem\t%s\tEspera: %4.1f"
                    %(env.now, keyOut, espera))
            yield env.timeout(random.normalvariate(*TEMPO_MONTAGEM))
            # acumula componente montado no Container de saída keyOut
            yield pecasContainerDict[keyOut].put(1)
            print("%5.1f Fim da montagem\t%s\tEstoque: %i"
                    %(env.now, keyOut, pecasContainerDict[keyOut].level))
```

Repare que a saída de tempo de espera que a função fornece é, na verdade, o tempo de espera do *montador* e não o tempo de espera em fila pelo recurso (veja o tópico "Teste seus conhecimentos" a seguir).

A chamada de execução do modelo não é muito diferente do desafio anterior, apenas precisamos criar um recurso único que realiza todos os processos:

```
random.seed(100)
env = simpy.Environment()

# cria estoques de peças
tipoList = ['A', 'B', 'C', 'D', 'AB', 'CD', 'ABCD']
pecasContainerDict = {}
for tipo in tipoList:
    pecasContainerDict[tipo] = simpy.Container(env)


# inicia processos de chegadas de pecas
for i in "ABCD":
    env.process(chegadaPecas(env, pecasContainerDict, i, 10))

# cria os recursos de montagem
montador = simpy.Resource(env, capacity=1)

# inicia processos de montagem
env.process(montagem(env, pecasContainerDict, montador, 'AB', A=1, B=2))
env.process(montagem(env, pecasContainerDict, montador,'CD', C=1, D=2))
env.process(montagem(env, pecasContainerDict, montador,'ABCD', AB=1, CD=1))

env.run(until=40)
```

Quando o modelo anterior é executado por apenas 40 minutos, temos como saída:

```
  0.0 Chegada de lote   A       Peças: 10
  0.0 Chegada de lote   B       Peças: 10
  0.0 Chegada de lote   C       Peças: 10
  0.0 Chegada de lote   D       Peças: 10
  0.0 Inicia montagem   AB      Espera:  0.0
  5.7 Fim da montagem   AB      Estoque: 1
  5.7 Inicia montagem   CD      Espera:  0.0
 11.8 Fim da montagem   CD      Estoque: 1
 11.8 Inicia montagem   ABCD    Espera:  0.0
 15.5 Fim da montagem   ABCD    Estoque: 1
 15.5 Inicia montagem   AB      Espera:  0.0
 21.6 Fim da montagem   AB      Estoque: 1
 21.6 Inicia montagem   CD      Espera:  0.0
 25.2 Fim da montagem   CD      Estoque: 1
 25.2 Inicia montagem   ABCD    Espera:  0.0
 29.6 Fim da montagem   ABCD    Estoque: 2
 29.6 Inicia montagem   AB      Espera:  0.0
 33.8 Fim da montagem   AB      Estoque: 1
 33.8 Inicia montagem   CD      Espera:  0.0
 39.7 Fim da montagem   CD      Estoque: 1
 39.7 Inicia montagem   ABCD    Espera:  0.0
```

Note como a produção de componentes ABCD caiu de 5 peças no peças no desafio 19, para apenas 2 neste desafio. Isso é naturalmente explicado, pois agora os processos não são mais paralelos e devem ser executados por um único montador.

Teste seus conhecimentos

1. Acrescente o cálculo do tempo médio em espera por fila de montador ao modelo do desafio 20;
2. Acrescente o cálculo do WIP - *Work In Progress* do processo ou o *trabalho em andamento*, isto é, quantas peças estão em produção ao longo do tempo de simulação;
3. Utilize a biblioteca matplotlib e construa um gráfico para evolução do estoque de cada peça ao longo da simulação, bem como do WIP.

Criando, manipulando e disparando eventos com `event()`

Nesta seção discutiremos comandos que lhe darão o poder de criar, manipular ou disparar seus próprios eventos, de modo independente dos processos já discutidos nas seções anteriores.

Mas com todo o poder, vem também a responsabilidade!

Atente-se para o fato de que, sem o devido cuidado, o seu modelo pode ficar um pouco confuso. Isto porque um evento pode ser criado a qualquer momento e fora do contexto original do processo em execução, naturalmente aumentando a complexidade do código.

Criando um evento isolado com `event()`

Considere um problema simples de controle de turno de abertura e fechamento de uma ponte elevatória. A ponte abre para automóveis, opera por 5 minutos, fecha e permite a passagem de embarcações no cruzamento por mais 5 minutos.

Naturalmente, esse modelo poderia ser implementado com os comandos já discutidos neste livro, contudo, a ideia desta seção é demonstrar como criar um evento específico que informe à ponte que ela deve fechar, algo semelhante a um sinal semafórico.

Em SimPy, um evento é criado pelo comando `env.event():`

```
abrePonte = env.event()           # cria o evento abrePonte
```

Criar um evento, não significa que executá-lo. Criar um evento significa apenas criá-lo na memória. Para processar um evento, isto é, marcá-lo como executado, utilizamos o método `.succeed():`

```
yield abrePonte.succeed()         # marca o evento abrePonte como executado
```

Podemos utilizar o evento criado de diversas formas em um modelo. Por exemplo, com o comando `yield` podemos fazer um processo aguardar até que o evento criado seja processado, com a linha:

```
yield abrePonte                   # aguarda até que o evento abrePonte seja processado
```

Retornando ao exemplo da ponte, criaremos um processo que representará o funcionamento da ponte. Inicialmente a ponte estará fechada e aguardará até que o evento `abrePonte` seja processado:

```
def ponteElevatoria(env):
    # opera a ponte elevatória
    global abrePonte

    print('%2.0f A ponte está fechada =(' %(env.now))
    # aguarda o evento para abertura da ponte
    yield abrePonte
    print('%2.0f A ponte está aberta  =)' %(env.now))
```

Note que `abrePonte` é tratado como uma variável global e isso significa que alguma outra função deve criá-lo e processá-lo, de modo que nossa função `ponteElevatória` abra a ponte no instante correto da simulação.

Assim, criaremos uma função geradora `turno` que representará o processo de controle do turno de abertura/fechamento da ponte e que será responsável por criar e processar o evento de abertura da mesma:

```
def turno(env):
    # abre e fecha a ponte
    global abrePonte

    while True:
        # cria evento para abertura da ponte
        abrePonte = env.event()
        # inicia o processo da ponte elvatória
        env.process(ponteElevatoria(env))
        # mantém a ponte fechada por 5 minutos
        yield env.timeout(5)
        # processa o evento de abertura da ponte
        yield abrePonte.succeed()
        # mantém a ponte aberta por 5 minutos
        yield env.timeout(5)
```

Note, na função anterior, que o evento é criado, mas **não é processado** imediatamente. De fato, ele só é processado quanto o método `abrePonte.succeed()` é executado, após o tempo de espera de 5 minutos.

Como queremos que a ponte funcione continuamente, um novo evento deve ser criado para representar o novo ciclo de abertura e fechamento. Isso está representado no início do laço com a linha:

```
# cria evento para abertura da ponte
abrePonte = env.event()
```

Precisamos deixar isso bem claro, paciente leitor: uma vez processado com o método `.succeed(),` o evento é extinto e caso seja necessário executá-lo novamente, teremos de recriá-lo com `env.event().`

Juntando tudo num único modelo de abre/fecha da ponte elevatória, temos:

```
import simpy

def turno(env):
    # abre e fecha a ponte
    global abrePonte

    while True:
        # cria evento para abertura da ponte
        abrePonte = env.event()
        # inicia o processo da ponte elvatória
        env.process(ponteElevatoria(env))
        # mantém a ponte fechada por 5 minutos
        yield env.timeout(5)
        # processa o evento de abertura da ponte
        yield abrePonte.succeed()
        # mantém a ponte aberta por 5 minutos
        yield env.timeout(5)

def ponteElevatoria(env):
    # opera a ponte elevatória
    global abrePonte

    print('%2.0f A ponte está fechada =(' %(env.now))
    # aguarda o evento para abertura da ponte
    yield abrePonte
    print('%2.0f A ponte está aberta  =)' %(env.now))

env = simpy.Environment()

# inicia o processo de controle do turno
env.process(turno(env))

env.run(until=20)
```

Quando executado por 20 minutos, o modelo anterior fornece:

```
 0 A ponte está fechada =(
 5 A ponte está aberta  =)
10 A ponte está fechada =(
15 A ponte está aberta  =)
```

No exemplo anterior, fizemos uso de uma variável global, `abrePonte`, para enviar a informação de que o evento de abertura da ponte foi disparado.

Isso é bom, mas também pode ser ruim =).

Note que o evento de abertura da ponte é criado e processado dentro da função `turno(env),` portanto, **fora** da função que controla o processo de abertura e fechamento da ponte, `ponteElevatoria(env).` As coisas podem realmente ficar confusas no seu modelo, caso você não tome o devido cuidado.

O método `.succeed()` ainda pode enviar um valor como parâmetro:

```
meuEvento.succeed(value=valor)
```

Poderíamos, por exemplo, enviar à função `ponteElevatoria` o tempo previsto para que a ponte fique aberta. O modelo a seguir, por exemplo, transfere o tempo de abertura (5 minutos) à função `ponteElevatoria` que o armazena na variável `tempoAberta:`

```
import simpy

def turno(env):
    # abre e fecha a ponte
    global abrePonte

    while True:
        # cria evento para abertura da ponte
        abrePonte = env.event()
        # inicia o processo da ponte elvatória
        env.process(ponteElevatoria(env))
        # mantém a ponte fechada por 5 minutos
        yield env.timeout(5)
        # dispara o evento de abertura da ponte
        yield abrePonte.succeed(value=5)
        # mantém a ponte aberta por 5 minutos
        yield env.timeout(5)

def ponteElevatoria(env):
    # opera a ponte elevatória
    global abrePonte

    print('%2.0f A ponte está fechada =(' %(env.now))
    # aguarda o evento para abertura da ponte
    tempoAberta = yield abrePonte
    print('%2.0f A ponte está  aberta =) e fecha em %2.0f minutos'
            %(env.now, tempoAberta))

env = simpy.Environment()
# inicia o processo de controle do turno
env.process(turno(env))

env.run(until=20)
```

Dentro da função `ponteElevatoria` a linha:

```
# aguarda o evento para abertura da ponte
tempoAberta = yield abrePonte
```

Aguarda até que o evento `abrePonte` seja executado e resgata seu valor (o tempo que a ponte deve permanecer aberta) na variável `tempoAberta.`

Concluindo, o potencial de uso do comando `event()` é extraordinário, mas, por experiência própria, garanto que seu uso descuidado pode tornar qualquer código ininteligível, candidato ao Campeonato Mundial de Código C Ofuscado (sim, isso existe!) ou mesmo algo semelhante a

utilizar desvios de laço `go... to` em um programa (des)estruturado).

Conceitos desta seção

Conteúdo	Descrição
`meuEvento = env.event()`	cria um novo *evento* `meuEvento` durante a simulação, mas não o processa.
`yield meuEvento`	aguarda até que o evento `meuEvento` seja processado
`yield meuEvento.succeed(value=valor)`	processa o evento `meuEvento`, isto é, engatilha o evento no tempo atual e inicia o seu processamento, retornando o parâmetro opcional `valor.`

Desafios

> **Desafio 21:** crie um processo de geração de automóveis que desejam cruzar a ponte, durante o horário de pico que dura 4 horas. Os intervalos entre chegadas sucessivas de veículos para travessia são exponencialmente distribuídos com média de 10 segundos (ou 6 veículos/min), e a ponte permite a travessia de 10 veículos por minuto. Após 4 horas de operação, quantos veículos estão em espera por travessia da ponte?
>
> **Desafio 22:** para o sistema anterior, construa um gráfico para o número de veículos em fila em função do tempo de abertura da ponte para travessia de automóveis. Qual o tempo mínimo que você recomendaria para abertura da ponte.

Solução dos desafios 21 e 22

> **Desafio 21:** crie um processo de geração de automóveis que desejam cruzar a ponte, durante o horário de pico que dura 4 horas. Os intervalos entre chegadas sucessivas de veículos para travessia são exponencialmente distribuídos com média de 10 segundos (ou 6 veículos/min), e a ponte permite a travessia de 10 veículos por minuto. Após 4 horas de operação, quantos veículos estão em espera por travessia da ponte?

Em relação ao modelo anterior, este novo sistema possui um processo de geração de chegadas de veículos e espera por abertura da ponte. Uma alternativa de implementação é utilizar um `Container` para armazenar os veículos em espera pela abertura da ponte, como no código a seguir:

```
import simpy
import random

CAPACIDADE_TRAVESSIA = 10   # capacidade da ponte em por min

def chegadaVeiculos(env, filaTravessia):
    while True:
        # aguarda a chegada do próximo veículo
        yield env.timeout(random.expovariate(6.0))
        # acrescenta o veículo na fila de travessia
        yield filaTravessia.put(1)
```

No código anterior, `filaTravessia` é um `Container` que representa o conjunto de veículos em espera por travessia da ponte. Cada veículo gerado é imediatamente transferido para o `Container`.

A função `turno` é semelhante à anterior, apenas com a inclusão do tempo de abertura da ponte, como o parâmetro `tempo_ponte:`

```
def turno(env, filaTravessia, tempo_ponte):
    # abre e fecha a ponte
    global abrePonte

    while True:
        # cria evento para abertura da ponte
        abrePonte = env.event()
        # inicia o processo da ponte elvatória
        env.process(ponteElevatoria(env, filaTravessia))
        # mantém a ponte fechada por 5 minutos
        yield env.timeout(5)
        # dispara o evento de abertura da ponte
        yield abrePonte.succeed(value=tempo_ponte)
        # mantém a ponte aberta por 5 minutos
        yield env.timeout(tempo_ponte)
```

Note que o parâmetro `tempo_ponte` é enviado como um valor para a função `ponteElevatória`, que agora deve lidar com a travessia dos veículos quando aberta. Neste caso, basta um comando `get` no `Container` que representa a fila de travessia dos veículos:

```
def ponteElevatoria(env, filaTravessia, tempo_ponte):
    # opera a ponte elevatória
    global abrePonte, naoAtendidos

    print('%2.0f A ponte está fechada =(' %(env.now))
    # aguarda o evento para abertura da ponte
    tempoAberta = yield abrePonte
    print('%2.0f A ponte está  aberta =) e fecha em %2.0f minutos'
            %(env.now, tempoAberta))

    # aguarda a chegada de mais veículos na fila de espera
    yield env.timeout(tempoAberta)

    # calcula quantos veículos podem atravessar a ponte
    numVeiculos = min(int(tempoAberta*CAPACIDADE_TRAVESSIA), filaTravessia.level)

    # retira os veículos da fila
    filaTravessia.get(numVeiculos)
    print('%2.0f Travessia de %i veículos\tFila atual: %i'
            %(env.now, numVeiculos, filaTravessia.level))
```

Analisando o código anterior, assim que a ponte abre, a linha:

```
# calcula quantos veículos podem atravessar a ponte
numVeiculos = min(int(tempoAberta*CAPACIDADE_TRAVESSIA), filaTravessia.level)
```

Estima quantos veículos, dentre aqueles que estão no `Container,` podem realizar a travessia. A seguir, os veículos são retirados do `Container` por meio de um comando `get` com parâmetro no número de veículos a serem retirados.

Ao final, deve-se criar o `Container`, realizar as chamadas dos processos e executar o modelo por 4 horas (ou 240 minutos):

```
random.seed(100)
env = simpy.Environment()

# container para representar a fila de automóveis aguardando a travessia
filaTravessia = simpy.Container(env)

# inicia o processo de controle do turno
env.process(turno(env,filaTravessia, 5))
env.process(chegadaVeiculos(env,filaTravessia))

env.run(until=240)
```

Quando executado, o modelo completo fornece como saída (resultados compactados):

```
  0 A ponte está fechada =(
  5 A ponte está  aberta =) e fecha em  5 minutos
 10 A ponte está fechada =(
...
225 A ponte está  aberta =) e fecha em  5 minutos
230 A ponte está fechada =(
230 Travessia de 50 veículos    Fila atual: 170
235 A ponte está  aberta =) e fecha em  5 minutos
```

Portanto, considerando-se as condições simuladas, o modelo indica que 170 veículos ainda estão em espera em fila ao final da última abertura da ponte dentro do horário de pico. Portanto, o tempo de abertura da ponte parece ser insuficiente durante o horário de pico.

> **Desafio 22:** para o sistema anterior, construa um gráfico para o número de veículos em fila em função do tempo de abertura da ponte para travessia de automóveis. Qual o tempo mínimo que você recomendaria de abertura da ponte.

Como o desafio deseja uma avaliação da fila ao final do horário de pico para diferentes valores de abertura da ponte, o primeiro passo é construir um laço para que o modelo possa ser executado para diferentes valores do tempo de abertura da ponte:

```
# lista para armazenar o resultado do cenário simulado
resultado = []

for tempo_ponte in range(5, 10):
    random.seed(100)
    env = simpy.Environment()

    # número de veículos não atendidos ao final da simulação
    naoAtendidos = 0

    # container para representar a fila de automóveis aguardando a travessia
    filaTravessia = simpy.Container(env)

    # inicia o processo de controle do turno
    env.process(turno(env,filaTravessia, tempo_ponte))
    env.process(chegadaVeiculos(env,filaTravessia))

    env.run(until=240)

    # arnazena o número de veículos do cenário atual em uma lista
    resultado.append((tempo_ponte, naoAtendidos))
```

Note, no código anterior, que foi criada uma lista, `resultado,` para armazenar o `tuple` com o tempo de abertura da ponte simulado e o resultado do número de veículos não atendidos ao final da simulação.

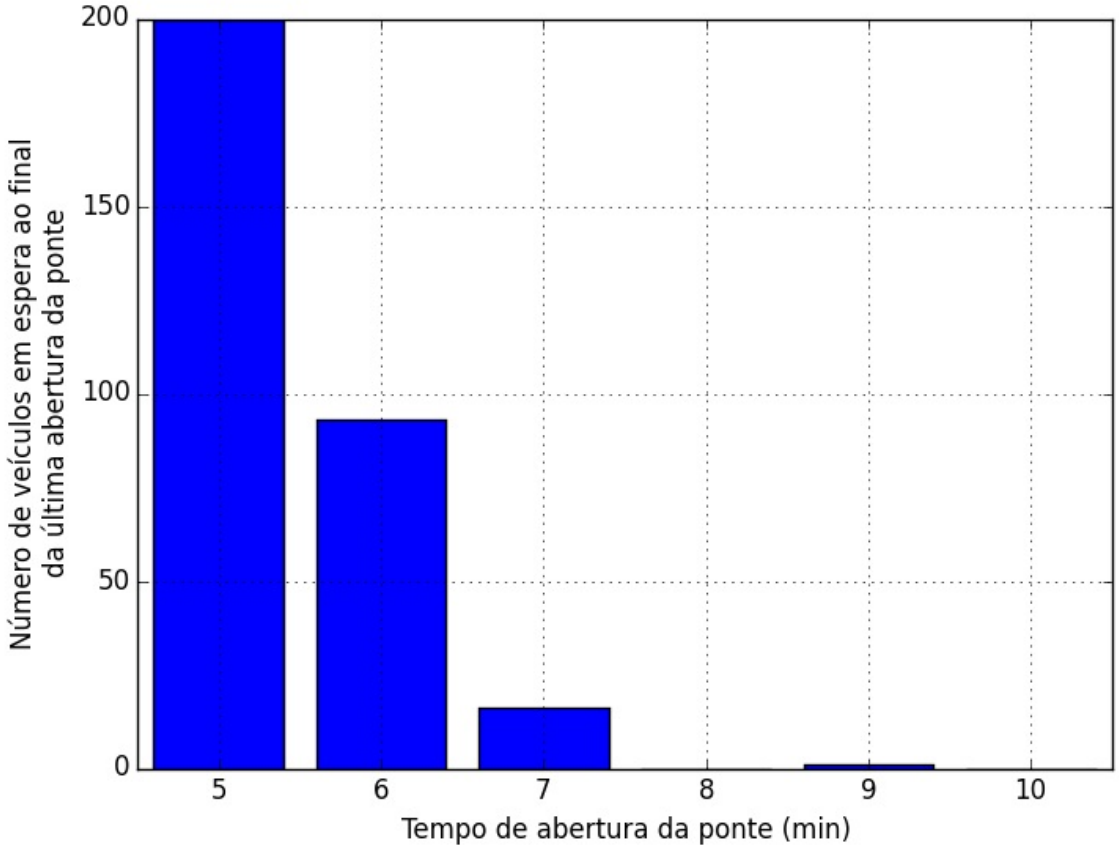
A função `ponteElevatoria` precisa de apenas de uma pequena modificação para assegurar que a variável global `naoAtendidos` receba corretamente o número de veículos não atendidos imediatamente após ao fechamento da ponte:

```
def ponteElevatoria(env, filaTravessia):
    # opera a ponte elevatória
    global abrePonte, naoAtendidos

    print('%2.0f A ponte está fechada =(' %(env.now))
    # aguarda o evento para abertura da ponte
    tempoAberta = yield abrePonte
    print('%2.0f A ponte está  aberta =) e fecha em %2.0f minutos'
            %(env.now, tempoAberta))

    # aguarda a chegada de mais veículos na fila de espera
    yield env.timeout(tempoAberta)
    # calcula quantos veículos podem atravessar a ponte
    numVeiculos = min(int(tempoAberta*CAPACIDADE_TRAVESSIA), filaTravessia.level)
    # retira os veículos da fila
    filaTravessia.get(numVeiculos)
    # armazena o número de veículos não atendidos ao final da abertura da ponte
    naoAtendidos = filaTravessia.level
    print('%2.0f Travessia de %i veículos\tFila atual: %i'
                %(env.now, numVeiculos, filaTravessia.level))
```

O próximo passo é acrescentar, ao final da simulação, um gráfico do número de veículos em função do tempo de abertura da ponte. Essa operação é facilitada pelo uso da biblioteca **matplotlib** no conjunto de dados armazenado na lista `resultado:`

```
# lista para armazenar o resultado do cenário simulado
resultado = []

for tempo_ponte in range(5, 11):
    random.seed(100)

    env = simpy.Environment()

    # número de veículos não atendidos ao final da simulação
    naoAtendidos = 0

    # container para representar a fila de automóveis aguardando a travessia
    filaTravessia = simpy.Container(env)

    # inicia o processo de controle do turno
    env.process(turno(env,filaTravessia, tempo_ponte))
    env.process(chegadaVeiculos(env,filaTravessia))

    env.run(until=240)

    # arnazena o número de veículos do cenário atual em uma lista
    resultado.append((tempo_ponte, naoAtendidos))

import matplotlib.pyplot as plt

# descompacta os valores armazenados na lista resultado e plota em um gráfico de barras
plt.bar(*zip(*resultado), align='center')
plt.xlabel('Tempo de abertura da ponte (min)')
plt.ylabel('Número de veículos em espera ao final\nda última abertura da ponte')
plt.xlim(4.5,10.5)
plt.grid(True)
plt.show()
```

Quando executado, o modelo anterior fornece como resultado o seguinte gráfico:



Em uma primeira análise, portanto, o tempo de abertura de 7 minutos seria suficiente para atender aos veículos durante o horário de pico. Contudo, nossa análise está limitada a uma replicação apenas, o que torna a conclusão, eventualmente, precipitada (veja o item 2 do tópico "Teste seus conhecimentos" a seguir).

Teste seus conhecimentos

1. Por que utilizamos na função `ponteElevatoria` a variável global `naoAtendidos?` Não seria suficiente armazenar na fila `resultados` diretamente o número de veículos no `Container` `filaTravessia,` pelo comando `filaTravessia.level?`
2. Como nos lembram Chiwf e Medina (2014)[1]: "nunca se deve tomar decisões baseadas em apenas uma replicação de um modelo de simulação". Afinal, basta modificar a semente geradora de número aleatórios, para que o resultado do gráfico seja outro (teste no seu modelo!). Modifique o modelo para que ele simule um número de replicações configurável para cada tempo de abertura da ponte. Adicionalmente, garanta que tempos de abertura diferentes utilizem a mesma sequencia de números aleatórios. (Dica: para esta segunda parte, armazene as sementes geradoras em uma lista).

[1] Chwif, L., and A. C. Medina. 2014. Modelagem e Simulacão de Eventos Discretos: Teoria e Aplicacões. 4ª ed. São Paulo: Elsevier Brasil.

Aguardando múltiplos eventos ao mesmo tempo com `AnyOf` e `AllOf`

Uma funcionalidade importante do SimPy é permitir que uma entidade aguarde até que dois ou mais eventos ocorram para então prosseguir com o processo. O SimPy possui duas opções muito interessantes para isso:

- `simpy.nyOf(env, eventos):` aguarda até que um dos eventos tenham ocorrido - `AnyOf` é equivalente ao símbolo de "|" (ou `or`);
- `simpy.AllOf(env, eventos):` aguarda até que todos os eventos tenham ocorrido - `AllOf` é equivalente ao símbolo de "&" (ou `and`). s Para compreender o funcionamento dos comandos anteriores, partiremos de um exemplo baseado numa obscura fábula infantil: a Lebre e a Tartaruga.

The Tortoise and the Hare", from an edition of Aesop's Fables illustrated by Arthur Rackham, 1912.

Neste exemplo, sortearemos um tempo de corrida para cada bicho e identificaremos quem foi o vencedor. Para tanto, além do sorteio, criaremos dois eventos que representam a corrida de cada bicho:

```
def corrida(env):
    # a lebre x tartaruga!
    # sorteia aleatoriamente os tempos dos animais
    lebreTempo = random.normalvariate(5,2)
    tartarugaTempo = random.normalvariate(5,2)
    # cria os eventos de corrida de cada animal
    lebreEvent = env.timeout(lebreTempo, value='lebre')
    tartarugaEvent = env.timeout(tartarugaTempo, value='tartaruga')
```

Na função `corrida,` criamos os eventos `lebreEvent` e `tartarugaEvent` , que simulam, respectivamente, as corridas da lebre e da tartaruga. Mas, lembre-se: apesar de criados, os eventos ainda não foram *necessariamente* executados. Como não existe um `yield` aplicado aos eventos, eles foram criados na memória do Python, mas só serão considerados *processados* após o tempo de simulação avançar o suficiente para que os tempos de `timeout` tenham passado.

De outra forma, os eventos `lebreEvent` e `tartarugaEvent` foram *disparados*, mas não processados, pois a função `corrida` ainda não tem um comando que aguarda o término do processamento desses eventos.

Aguardando até que, ao menos, um evento termine com `AnyOf`

Para garantir que a função `corrida` aguarde até que, ao menos, um dos corredores termine a prova, uma opção é acrescentar um `yield AnyOf()` (que pode ser substituído por "|") após a criação dos eventos:

```
    # começou!
    start = env.now
    print('%3.1f Iniciada a corrida!' %(env.now))
    # simule até que o ao menos um dos eventos termine
    resultado = yield lebreEvent | tartarugaEvent
    tempo = env.now - start
```

O `yield` garante que a função aguardará até que um dos dois eventos - `lebreEvent` ou `tartarugaEvent` - termine e a variável `resultado` armazenará qual desses eventos terminou primeiro (ou mesmo os dois, em caso de empate). Assim, para sabermos quem venceu, basta explorarmos o valor contido na variável `resultado` .
O código a seguir completa o modelo, testando qual dos dois eventos está na variável `resultado` :

```
import simpy
import random

def corrida(env):
    # a lebre x tartaruga!
    # sorteia aleatoriamente os tempos dos animais
    lebreTempo = random.normalvariate(5,2)
    tartarugaTempo = random.normalvariate(5,2)
    # cria os eventos de corrida de cada animal
    lebreEvent = env.timeout(lebreTempo, value='lebre')
    tartarugaEvent = env.timeout(tartarugaTempo, value='tartaruga')

    # começou!
    start = env.now
    print('%3.1f Iniciada a corrida!' %(env.now))
    # simule até que alguém chegue primeiro
    resultado = yield lebreEvent | tartarugaEvent
    tempo = env.now - start

    # quem venceu?
    if lebreEvent not in resultado:
        print('%3.1f A tartaruga venceu em %3.1f minutos' %(env.now, tempo))
    elif tartarugaEvent not in resultado:
        print('%3.1f A lebre venceu em %3.1f minutos' %(env.now, tempo))
    else:
        print('%3.1f Houve um empate em %3.1f minutos' %(env.now, tempo))

random.seed(10)
env = simpy.Environment()
proc = env.process(corrida(env))
env.run(until=10)
```

Quando o modelo anterior é executado, o bicho pega:

```
0.0 Iniciada a corrida!
5.3 A tartaruga venceu em 5.3 minutos
```

> **Observação:** a linha:
>
> ```
> resultado = yield lebreEvent | tartarugaEvent
> ```
>
> Poderia ter sido substituída, pela linha:
>
> ```
> resultado = yield simpy.AnyOf(env, lebreEvent, tartarugaEvent)
> ```

Aguardando todos os eventos com `AllOf`

Para não deixar ninguém triste, poderíamos forçar um empante na corrida, aguardando que a função `corrida` aguarde até que os dois eventos sejam concluídos para decretar o vencedor. Para isso, basta substituir a linha:

```
resultado = yield lebreEvent | tartarugaEvent
```

por:

```
resultado = yield lebreEvent & tartarugaEvent
```

Quando simulado, o novo modelo forncece como saída:

```
0.0 Iniciada a corrida!
5.4 Houve um empate em 5.4 minutos
```

O que ocorreu? Neste caso, o comando `AllOf` (ou "&") aguardou até que os dois eventos terminassem para liberar o processamento da linha seguinte de código e nosso desvio de condição `if` identificou que a variável `resultado` possuia os dois eventos armazenados.

Comprendendo melhor as saídas dos comandos `AllOf` e `AnyOf`

Agora que já sabemos o que fazem os comandos `AllOf` e `AnyOf`, vamos explorar nessa seção um pouco mais sobre o que exatamente esses comandos retornam.

Inicialmente, imprima os valores dos eventos e da variável `resultado`, para descobrir seus conteúdos:

```
import simpy
import random

def corrida(env):
    # a lebre x tartaruga!
    # sorteia aleatoriamente os tempos dos animais
    lebreTempo = random.normalvariate(5,2)
    tartarugaTempo = random.normalvariate(5,2)
    # cria os eventos de corrida de cada animal
    lebreEvent = env.timeout(lebreTempo, value='lebre')
    tartarugaEvent = env.timeout(tartarugaTempo, value='tartaruga')
    print('lebreEvent= ', lebreEvent)
    print('tartarugaEvent= ', tartarugaEvent)
    # começou!
    start = env.now
    print('%3.1f Iniciada a corrida!' %(env.now))
    # simule até que alguém chegue primeiro
    resultado = yield lebreEvent & tartarugaEvent
    print('resultado = ', resultado)
    tempo = env.now - start

    # quem venceu?
    if lebreEvent not in resultado:
        print('%3.1f A tartaruga venceu em %3.1f minutos' %(env.now, tempo))
    elif tartarugaEvent not in resultado:
        print('%3.1f A lebre venceu em %3.1f minutos' %(env.now, tempo))
    else:
        print('%3.1f Houve um empate em %3.1f minutos' %(env.now, tempo))

random.seed(10)
env = simpy.Environment()
proc = env.process(corrida(env))
env.run(until=10)
```

Quando executado, o programa fornece:

```
lebreEvent=  <Timeout(5.428964407135667, value=lebre) object at 0xa592470>
tartarugaEvent=  <Timeout(5.33749212083634, value=tartaruga) object at 0xa5920f0>
0.0 Iniciada a corrida!
resultado =  <ConditionValue {<Timeout(5.33749212083634, value=tartaruga) object at 0xa5920f0>
: 'tartaruga',
<Timeout(5.428964407135667, value=lebre) object at 0xa592470>: 'lebre'}>
5.4 Houve um empate em 5.4 minutos
```

Pela saída anterior, descobrimos, inicialmente, que os eventos são *objetos* do tipo `Timeout` e que armazenam tanto o tempo de espera, quanto o valor (ou `value`) fornecido na chamada da função `env.timeout`.

Um pouco mais abaixo, a saída revela que a variável `resultado` é um objeto da classe `ConditionValue` que, aparentemente, contém um dicionário de eventos em seu interior. Para acessar esse dicionário, o SimPy fornece o método `.todict():`

```
 resutado.todict()
```

Que retorna:

```
{<Timeout(5.428964407135667, value=lebre) object at 0xa18e30>: 'lebre',
<Timeout(5.33749212083634, value=tartaruga) object at 0xa18eb0>: 'tartaruga'}
```

Que nada mais é do que um dicionário padrão do Python, onde as `keys` são os eventos e os `items` são os valores dos eventos.

Conceitos desta seção

Conteúdo	Descrição	
`AnyOf(env, eventos)`	aguarda até que um dos eventos tenham ocorrido - `AnyOf` é equivalente ao símbolo de "\	" (ou `or`).
`AllOf(env, eventos)`	aguarda até que todos os eventos tenham ocorrido - `AllOf` é equivalente ao símbolo de "&" (ou `and`).	

Desafios

> **Desafio 23:** Considere que existe uma probabilidade de que a Lebre, por alguma razão mal explicada (eu realizaria um teste antidoping nos competidores), resolva que é uma boa ideia tirar uma soneca de 5 minutos em algum instante entre 2 e 10 minutos do início da corrida. Modele esta nova situação (dica: crie um função `soneca` que gera um evento que pode ocasionar a parada da Lebre ainda durante a corrida).
>
> **Desafio 24:** É interessante notar, que mesmo quando um dos competidores *perde* a corrida, de fato, o respectivo evento **não é** cancelado. Altere o modelo anterior para marcar o horário de chegada dos dois competidores, garantindo que os eventos `lebreEvent` e `tartarugaEvent` sejam executados até o fim.

Desafios

> **Desafio 23:** Considere que existe uma probabilidade de que a Lebre, por alguma razão mal explicada (eu realizaria um teste antidoping nos competidores), resolva que é uma boa ideia tirar uma soneca de 5 minutos em algum instante entre 2 e 10 minutos do início da corrida. Modele esta nova situação (dica: crie um função `soneca` que gera um evento que pode ocasionar a parada da Lebre ainda durante a corrida).

Para este desafio será criada uma função para gerar a soneca e uma função adicional para identificar o vencedor, pois agora a tartaruga pode (ou não) vencer enquanto a lebre estiver dormindo.

Começando pela função que determina o vencedor:

```
def imprimeVencedor(start, resultado, lebreEvent, tartarugaEvent):
    # determina o vencedor da corrida
    tempo = env.now - start
    # quem venceu?
    if lebreEvent not in resultado:
        print('%3.1f A tartaruga venceu em %3.1f minutos' %(env.now, tempo))
    elif tartarugaEvent not in resultado:
        print('%3.1f A lebre venceu em %3.1f minutos' %(env.now, tempo))
    else:
        print('%3.1f Houve um empate em %3.1f minutos' %(env.now, tempo))
```

A função `soneca` a seguir, cria um evento `sonecaEvent,` sorteia o instante da soneca e processa o evento no instante correto:

```
def soneca(env):
    global sonecaEvent

    # cria o evento da soneca
    sonecaEvent = env.event()
    # sorteia o instante da soneca
    inicioSoneca = random.uniform(2, 10)
    # aguarda instante da soneca
    yield env.timeout(inicioSoneca)
    # durma!
    sonecaEvent.succeed()
```

A função corrida, agora deve lidar com as diversas situações possíveis: uma soneca da lebre durante a corrida, a lebre acordando, a tartaruga ou a lebre vencendo. Essas diversas situações podem ser facilmente modeladas por um conjunto de lógicas de desvio condicional `if` concatenadas com operações do tipo `AnyOf` :

```
def corrida(env):
    # a lebre x tartaruga!
    global sonecaEvent

    # sorteia aleatoriamente os tempos dos animais
    lebreTempo = random.normalvariate(5,2)
    tartarugaTempo = random.normalvariate(5,2)
    # cria os eventos de corrida de cada animal
    lebreEvent = env.timeout(lebreTempo, value='lebre')
    tartarugaEvent = env.timeout(tartarugaTempo, value='tartaruga')

    # começou!
    start = env.now
    print('%3.1f Iniciada a corrida!' %(env.now))
    # simule até que alguém chegue primeiro ou alguém pegue no sono...
    resultado = yield lebreEvent | tartarugaEvent | sonecaEvent

    if sonecaEvent in resultado:
        # a lebre dormiu!
        lebreTempo -= env.now - start
        print('%3.1f A lebre capotou de sono?!?!' %(env.now))
        lebreAcorda = env.timeout(5, value='lebre')
        resultado = yield lebreAcorda | tartarugaEvent

        if tartarugaEvent in resultado:
            # a tartaruga venceu durante o sono da lebre
            imprimeVencedor(start, resultado, None, tartarugaEvent)
            env.exit()
        else:
            # a lebre acordou antes da corrida terminar
            print('%3.1f A lebre acordou, amigo!' %(env.now))
            lebreEvent2 = env.timeout(lebreTempo, value='lebre')
            resultado = yield lebreEvent2 | tartarugaEvent
            # alguém venceu!
            imprimeVencedor(start, resultado, lebreEvent2, tartarugaEvent)
    else:
        # alguém venceu, antes da lebre pegar no sono!
        imprimeVencedor(start, resultado, lebreEvent, tartarugaEvent)
```

Não devemos esquecer de inciar o processo da soneca:

```
random.seed(100)
env = simpy.Environment()
# inicia processo da soneca
env.process(soneca(env))
# inicia a corrida
env.process(corrida(env))

env.run(until=10)
```

Note, na função `corrida,` que quando a tartaruga vence enquanto a lebre está dormindo, a função `imprimeVencedor` é chamada com o parâmetro `lebreEvent=None`, já informando à função que a lebre perdeu.

Quando o modelo anterior é simulado, fornece como resultado:

```
0.0 Iniciada a corrida!
3.2 A lebre capotou de sono?!?!
7.6 A tartaruga venceu em 7.6 minutos
```

Você pode alterar o valor da semente geradora de números aleatórios - na linha `random.seed(...)` - e apostar com os amigos quem vencerá a próxima corrida!

> **Desafio 24:** É interessante notar, que mesmo quando um dos competidores *perde* a corrida, de fato, o respectivo evento **não é** cancelado. Altere o modelo anterior para marcar o horário de chegada dos dois competidores, garantindo que os eventos `lebreEvent` e `tartarugaEvent` sejam executados até o fim.

Uma possível solução para o desafio é transformar o a função `imprimeVencedor` em um processo que, após informar o vencedor, continua até que o outro competidor passe pela linha de chegada. Por exemplo, podemos uma alternativa seria:

```
def imprimeVencedor(env, start, resultado, lebreEvent, tartarugaEvent):
    # determina o vencedor da corrida
    tempo = env.now - start
    # quem venceu?
    if lebreEvent not in resultado:
        print('%3.1f A tartaruga venceu em %3.1f minutos' %(env.now, tempo))
        if lebreEvent:
            yield lebreEvent
            print('%3.1f A lebre chega em 2º lugar' %(env.now))
    elif tartarugaEvent not in resultado:
        print('%3.1f A lebre venceu em %3.1f minutos' %(env.now, tempo))
        yield tartarugaEvent
        print('%3.1f A tartaruga chega em 2º lugar' %(env.now))
    else:
        print('%3.1f Houve um empate em %3.1f minutos' %(env.now, tempo))
```

A função `corrida` deve agora utilizar comandos do tipo `env.process(imprimeVencedor)` para inciar o processo que determina quem venceu e continua a corrida até que o outro competidor chegue. Note, no código a seguir, que apenas no caso da lebre ser pega ainda dormindo, a modificação do código é um pouco mais trabalhosa:

```
def corrida(env):
    # a lebre x tartaruga!
    global sonecaEvent

    # sorteia aleatoriamente os tempos dos animais
    lebreTempo = random.normalvariate(5,2)
    tartarugaTempo = random.normalvariate(5,2)
    # cria os eventos de corrida de cada animal
    lebreEvent = env.timeout(lebreTempo, value='lebre')
    tartarugaEvent = env.timeout(tartarugaTempo, value='tartaruga')

    # começou!
    start = env.now
    print('%3.1f Iniciada a corrida!' %(env.now))
    # simule até que alguém chegue primeiro ou alguém pegue no sono...
    resultado = yield lebreEvent | tartarugaEvent | sonecaEvent

    if sonecaEvent in resultado:
        # a lebre dormiu!
        lebreTempo -= env.now - start
        print('%3.1f A lebre capotou de sono?!?!' %(env.now))
        lebreAcorda = env.timeout(lebreTempo, value='acorda')
        resultado = yield lebreAcorda | tartarugaEvent

        if tartarugaEvent in resultado:
            # a tartaruga venceu durante o sono da lebre
            env.process(imprimeVencedor(env, start, resultado, None, tartarugaEvent))
            yield lebreAcorda
            print('%3.1f A lebre acordou, amigo!' %(env.now))
            yield env.timeout(lebreTempo, value='lebre')
            print('%3.1f A lebre chega em 2º lugar' %(env.now))
            # termina o processo
            env.exit()
        else:
            # a lebre acordou antes da corrida terminar
            print('%3.1f A lebre acordou, amigo!' %(env.now))
            lebreEvent2 = env.timeout(lebreTempo, value='lebre')
            resultado = yield lebreEvent2 | tartarugaEvent
            # alguém venceu!
            env.process(imprimeVencedor(env, start, resultado, lebreEvent2, tartarugaEvent))
    else:
        # alguém venceu, antes da lebre pegar no sono!
        env.process(imprimeVencedor(env, start, resultado, lebreEvent, tartarugaEvent))
```

Quando executado, o modelo completo fornece como saída:

```
0.0 Iniciada a corrida!
3.2 A lebre capotou de sono?!?!
7.6 A tartaruga venceu em 7.6 minutos
8.2 A lebre acordou, amigo!
9.3 A lebre chega em 2º lugar
```

O que é importante destacar é que o comando `AnyOf` **não** paralisa um evento que ainda não foi processado. No saída exemplo, a tartaruga venceu e o processamento da linha:

```
yield lebreAcorda
```

Fez o modelo aguardar até que o evento da lebre acordar fosse processado (no instante determinado quando o evento foi criado). Não houve, portanto, um novo evento criado para a lebre, apenas aguardamos a conclusão do evento já engatilhado na memória, mas não concluído.

Teste seus conhecimentos

1. Generalize a função `corrida` para um número qualquer de competidores utilizando o operador `**kwargs` visto no Desafio 19.
2. Construa um gráfico com a evolução da prova, isto é, a distância percorrida por cada competidor x tempo de prova.

Propriedades úteis dos eventos

Um evento possui algumas propriedades que fazem a alegria de qualquer leitor:

- `Event.value:` o valor que foi passado para o evento no momento de sua criação;
- `Event.triggered:` `True,` caso o `Event` já tenha sido engatilhado, isto é, ele está na fila de eventos do SimPy e programado para ocorrer em determinado instante da simulação; `False,` caso contrário;
- `Event.processed:` `True,` caso o `Event` já tenha sido executado e `False,` caso contrário.

Existe uma dificuldade inicial que deve ser obrigatoriamente superada pelo programador: compreender a sequência de criação, disparo e execução de um evento em SimPy.

No momento da sua criação, todo evento surge como um objeto na memória do SimPy e, inicialmente, ele encontra-se no estado *não engatilhado* (`Event.triggered = False`). Quando o evento é programado para ocorrer em determinado instante da simulação, ele passa ao estado *engatilhado* (`Event.triggered = True`). Quando o evento é finalmente executado no instante determinado, seu estado passa a processado: (`Event.processed = True`).
Por exemplo, quando você adiciona ao modelo a linha de comando:

```
yield env.timeout(10)
```

O SimPy processará a linha na seguinte sequência de passos (figura):

1. Cria na memória um novo evento dentro do `Environment env;`
2. Engatilha o evento para ser processado dali a 10 unidades de tempo (minutos, por exemplo);
3. Quando a simulação atinge o instante de processamento esperado (passados os 10 minutos), o evento é processado. Automaticamente, o comando `yield` recebe o sinal de sucesso do processamento e o fluxo de execução do modelo retoma seu curso normal, processando a linha seguinte do modelo.

A figura ... elenca os diversos tipos de eventos que discutimos ao longo deste livro e sua sequência usual de criação->engatilhamento->processamento. Note como eles estão intrinsecamente ligados à questão do tempo de simulação.

Antes de avançar - e com o intuito de facilitar o aprendizagem do *lebrístico* leitor - acrecentaremos ao modelo da Lebre e da Tartaruga uma função para imprimir as propriedade dos eventos. Basicamente ela recebe uma lista de eventos e imprime na tela as propriedades de cada evento da lista:

```
def printEventStatus(env, eventList):
    # imprime o status das entidades
    for event in eventList:
        print('%3.1f value: %s\t programado: %s\t processado: %s'
            %(env.now, event.value, event.triggered, event.processed))
```

Basicamente, devemos criar uma lista a partir dos eventos de corrida da Lebre e da Tartaruga, para fornecer a lista para nossa função de impressão de propriedades dos eventos:

```
# cria os eventos de corrida de cada animal
lebreEvent = env.timeout(lebreTempo, value='lebre')
tartarugaEvent = env.timeout(tartarugaTempo, value='tartaruga')

# cria lista de eventos
eventList = [lebreEvent, tartarugaEvent]
printEventStatus(env, eventList)
```

O modelo completo, com a impressão das propriedades dos eventos, ficaria:

```
import simpy
import random

def corrida(env):
    # a lebre x tartaruga!
    # sorteia aleatoriamente os tempos dos animais
    lebreTempo = random.normalvariate(5,2)
    tartarugaTempo = random.normalvariate(5,2)

    # cria os eventos de corrida de cada animal
    lebreEvent = env.timeout(lebreTempo, value='lebre')
    tartarugaEvent = env.timeout(tartarugaTempo, value='tartaruga')
    eventList = [lebreEvent, tartarugaEvent]
    printEventStatus(env, eventList)

    # começou!
    start = env.now
    print('%3.1f Iniciada a corrida!' %(env.now))
    # simule até que alguém chegue primeiro
    resultado = yield lebreEvent | tartarugaEvent
    tempo = env.now - start
    # quem venceu?
    if lebreEvent not in resultado:
        print('%3.1f A tartaruga venceu em %3.1f minutos' %(env.now, tempo))
    elif tartarugaEvent not in resultado:
        print('%3.1f A lebre venceu em %3.1f minutos' %(env.now, tempo))
    else:
        print('%3.1f Houve um empate em %3.1f minutos' %(env.now, tempo))
    printEventStatus(env, eventList)

def printEventStatus(env, eventList):
    # imprime o status das entidades
    for event in eventList:
        print('%3.1f value: %s\t programado: %s\t processado: %s'
            %(env.now, event.value, event.triggered, event.processed))

random.seed(10)
env = simpy.Environment()
env.process(corrida(env))
env.run(until=10)
```

Quando simulado, o modelo fornece como resultado:

```
0.0 value: lebre         programado: True        processado: False
0.0 value: tartaruga     programado: True        processado: False
0.0 Iniciada a corrida!
5.3 A tartaruga venceu em 5.3 minutos
5.3 value: lebre         programado: True        processado: False
5.3 value: tartaruga     programado: True        processado: True
```

Repare no instante 0.0, que os eventos do tipo `timeout` são programados imediatamente após sua criação. Quando a tartaruga vence, o seus evento é considerado processado e a lebre não, algo esperado. Contudo, o que aconteceria se simulassemos o modelo por mais tempo? Ou seja,

se deixássemos o modelo processando por mais tempo, a lebre apareceria na linha de chegada? No modelo a seguir, acrescentamos à função corrida um evento `timeout,` logo após a chegada, que representaria a comemoração da tartaruga por 4 minutos:

```
def corrida(env):
    # a lebre x tartaruga!
    # sorteia aleatoriamente os tempos dos animais
    lebreTempo = random.normalvariate(5,2)
    tartarugaTempo = random.normalvariate(5,2)

    # cria os eventos de corrida de cada animal
    lebreEvent = env.timeout(lebreTempo, value='lebre')
    tartarugaEvent = env.timeout(tartarugaTempo, value='tartaruga')
    eventList = [lebreEvent, tartarugaEvent]
    printEventStatus(env, eventList)

    # começou!
    start = env.now
    print('%3.1f Iniciada a corrida!' %(env.now))
    # simule até que alguém chegue primeiro
    resultado = yield lebreEvent | tartarugaEvent
    tempo = env.now - start
    # quem venceu?
    if lebreEvent not in resultado:
        print('%3.1f A tartaruga venceu em %3.1f minutos' %(env.now, tempo))
    elif tartarugaEvent not in resultado:
        print('%3.1f A lebre venceu em %3.1f minutos' %(env.now, tempo))
    else:
        print('%3.1f Houve um empate em %3.1f minutos' %(env.now, tempo))
    printEventStatus(env, eventList)

    # a tartaruga inicia a comemoração!
    yield env.timeout(4)
    printEventStatus(env, eventList)
```

Como resultado, agora o modelo fornece como saída:

```
0.0 value: lebre         programado: True        processado: False
0.0 value: tartaruga     programado: True        processado: False
0.0 Iniciada a corrida!
5.3 A tartaruga venceu em 5.3 minutos
5.3 value: lebre         programado: True        processado: False
5.3 value: tartaruga     programado: True        processado: True
9.3 value: lebre         programado: True        processado: True
9.3 value: tartaruga     programado: True        processado: True
```

Repare que no instante 9.3 minutos o evento da lebre está processado. Isto é importantíssimo e não avance sem compreender o que está acontecendo: quando a linha:

```
# simule até que alguém chegue primeiro
resultado = yield lebreEvent | tartarugaEvent
```

Aguardou até que um dos eventos tivesse terminado, ela **não cancelou** o outro evento que, por ser um `timeout`, continuou na fila de eventos até que o SimPy pudesse processá-lo. (Note que o instante 9.3 não representa o instante de processamento do processo, provavelmente a lebre cruzou a linha de chegada após isso).

> Observação: até a presente versão do SimPy, não existe a possibilidade de se cancelar um evento já programado.

Conceitos desta seção

Conteúdo	Descrição
`Event.value`	o valor que foi passado para o evento no momento de sua criação.
`Event.triggered`	`True,` caso o `Event` já tenha sido engatilhado, isto é, ele está na fila de eventos do SimPy e programado para ocorrer em determinado instante da simulação; `False,` caso contrário.
`Event.processed`	`True,` caso o `Event` já tenha sido executado e `False,` caso contrário.

Desafio

> Desafio 25: modifique o modelo anterior para que ele aguarde até a chegada da lebre.

Solução do desafio 25

> Desafio 25: modifique o modelo anterior para que ele aguarde até a chegada da lebre.

Para este desafio, basta alterarmos a parte do modelo que lida com a decisão de quem venceu:

```
    # quem venceu?
    if lebreEvent not in resultado:
        print('%3.1f A tartaruga venceu em %3.1f minutos' %(env.now, tempo))
        printEventStatus(env, eventList)
        # aguarda o fim do evento da lebre
        yield lebreEvent
        printEventStatus(env, eventList)
        print('%3.1f A lebre chega em segundo...' %(env.now))
    elif tartarugaEvent not in resultado:
        print('%3.1f A lebre venceu em %3.1f minutos' %(env.now, tempo))
        printEventStatus(env, eventList)
        # aguarda o fim do evento da tartaruga
        yield tartarugaEvent
        printEventStatus(env, eventList)
        print('%3.1f A tartaruga chega em segundo...' %(env.now))
    else:
        print('%3.1f Houve um empate em %3.1f minutos' %(env.now, tempo))
        printEventStatus(env, eventList)
```

Quando simulado, o modelo anterior fornece:

```
0.0 value: lebre         programado: True        processado: False
0.0 value: tartaruga     programado: True        processado: False
0.0 Iniciada a corrida!
5.3 A tartaruga venceu em 5.3 minutos
5.3 value: lebre         programado: True        processado: False
5.3 value: tartaruga     programado: True        processado: True
5.4 value: lebre         programado: True        processado: True
5.4 value: tartaruga     programado: True        processado: True
5.4 A lebre chega em segundo...
```

Nessa replicação a lebre perdeu por 0,1 minutos apenas. Um cochilada imperdoável da lebre, MEU AMIGO!

Na próxima seção, veremos uma tática mais interessante para resolver o mesmo problema a partir da novidade a ser apresentada, os `callbacks.`

Teste seus conhecimentos

1. Crie um processo `chuva` que ocorre entre intervalos exponencialmente distribuídos com média de 5 minutos e dura, em média, 5 minutos exponencialmente distribuídos também. Quando a chuva começa, os corredores são 50% mais lentos durante o período. (Dica:

quando a chuva começar, construa novos eventos `timeout` e despreze os anteriores, mas calcule antes quais as velocidades dos corredores).

2. Modifique o modelo para que ele execute um número configurável de replicações e forneça como resposta a porcentagem das vezes em que cada competidor venceu.

Adicionando `callbacks` aos eventos

SimPy possui uma ferramenta tão curiosa quanto poderosa: os `callbacks.` Um `callback` é uma função que você *acrescenta* ao final de um evento. Por exemplo, considere que quando o evento da tartaruga (ou da lebre) termina, desejamos imprimir o vencedor na tela. Assim, quando o evento é *processado*, desejamos que ele processe a seguinte função, que recebe o evento como **único** parâmetro de entrada:

```
def campeao(event):
    # imprime a faixa de campeão
    print('%3.1f \o/ Tan tan tan (música do Senna) A %s é a campeã!\n'
                %(env.now, event.value))
```

Toda função para ser anexada como um `callback`, deve possuir como parâmetro de chamada apenas um único evento. Note também, que o valor de `env.now` impresso na tela é possível pois `env` é uma variável global para o Python. (Caso você ainda tenha alguma dificuldade para entender como o Python lida com variáveis globais e locais, um bom caminho é ler este bom guia).

Para anexarmos a função `campeão` a um evento, utilizaremos o método `callbacks.append(função_criada):`

```
    # cria os eventos de corrida de cada animal
    lebreEvent = env.timeout(lebreTempo, value='lebre')
    tartarugaEvent = env.timeout(tartarugaTempo, value='tartaruga')

    # acrecenta os callbacks
    lebreEvent.callbacks.append(campeao)
    tartarugaEvent.callbacks.append(campeao)
```

O código completo do modelo com `callbacks` ficaria:

```
import simpy
import random

def corrida(env):
    # a lebre x tartaruga!
    # sorteia aleatoriamente os tempos dos animais
    # cria os eventos que disparam as corridas
    lebreTempo = random.normalvariate(5,2)
    tartarugaTempo = random.normalvariate(5,2)
    # cria os eventos de corrida de cada animal
    lebreEvent = env.timeout(lebreTempo, value='lebre')
    tartarugaEvent = env.timeout(tartarugaTempo, value='tartaruga')

    # acrecenta os callbacks
    lebreEvent.callbacks.append(campeao)
    tartarugaEvent.callbacks.append(campeao)

    # começou!
    print('%3.1f Iniciada a corrida!' %(env.now))
    # simule até que alguém chegue primeiro
    yield lebreEvent | tartarugaEvent

def campeao(event):
    # imprime a faixa de campeão
    print('%3.1f \o/ Tan tan tan (música do Senna) A %s é a campeã!\n'
            %(env.now, event.value))


random.seed(10)
env = simpy.Environment()
proc = env.process(corrida(env))
env.run(until=10)
```

Note como o código ficou razoavelmente mais compacto, por eliminamos toda a codificação referente aos testes `if...then...else` para determinar que é o campeão.
Quando executado, o modelo fornece como resultado:

```
0.0 Iniciada a corrida!
5.3 \o/ Tan tan tan (música do Senna) A tartaruga é a campeã!
5.4 \o/ Tan tan tan (música do Senna) A lebre é a campeã!
```

OPS!

Aos 5.4 minutos a lebre, que chegou depois, foi declarada campeã também. Como isso aqui não é a Federação Paulista de Futebol, temos de corrigir o modelo e garantir que vença sempre quem chegar primeiro.

Uma solução prática seria criar uma variável global que se torna `True` quando o primeiro corredor ultrapassa a linha, de modo que a função `campeao` consiga distinguir se já temos um vencedor ou não:

```
import simpy
import random

vencedor = False    # se já temos um vencedor na corrida
```

A função `campeao` agora deve lidar com uma lógica de desvio de fluxo para determinar se evento representa um campeão ou não:

```
def campeao(event):
    global vencedor

    if not vencedor:
        # imprime a faixa de campeão
        print('%3.1f \o/ Tan tan tan (música do Senna) A %s é a campeã!'
                    %(env.now, event.value))
        # atualiza a variável global vencedor
        vencedor = True
    else:
        # imprime o perdedor
        print('%3.1f A %s chega em segundo lugar...'
                    %(env.now, event.value))
```

Quando simulado, o modelo fornece como saída:

```
0.0 Iniciada a corrida!
5.3 \o/ Tan tan tan (música do Senna) A tartaruga é a campeã!
5.4 A lebre chega em segundo lugar...
```

Você pode adicionar quantas funções de `callback` quiser ao seu evento, mas lembre-se que manipular um modelo diretamente por eventos tende a deixar o código ligeiramente confuso e a boa prática recomenda não economizar nos comentários.

Todo processo é um evento

Quando um processo é gerado pelo comando `env.process(),` o processo gerado é automaticamente tratado como um evento pelo SimPy. Você pode igualmente adicionar `callbacks` aos processos ou mesmo retornar um valor (como já vimos na seção....).
Por exemplo, vamos acrescentar uma função de `callback` para ao processo `corrida` que informa o final da corrida para o público:

```
def final(event):
    # imprime o aviso de final da corrida
    print('%3.1f Ok pessoal, a corrida acabou.' %env.now)
```

Precisamos agora, modificar apenas os comandos que inicializam a função `corrida,` anexando o `callback` criado:

```
random.seed(10)
env = simpy.Environment()
proc = env.process(corrida(env))
# adiciona ao processo proc a função callback final
proc.callbacks.append(final)

# executa até que o processo corrida termine
env.run(proc)
```

Note que, além de adicionarmos a função `final` como `callback` da função `corrida,` modificamos o comando `env.run()` para que ele simule até que a função `corrida` termine seu processamento. (Experimente substituir a linha `env.run(proc)` por `env.run(until=10)` e verifique o que acontece).

Quando executado, o modelo fornece como saída:

```
0.0 Iniciada a corrida!
5.3 \o/ Tan tan tan (musica do Senna) A tartaruga é a campeã!
5.3 Ok pessoal, a corrida acabou.
```

Uma boa pedida para `callbacks` é construir funções que calculem estatísticas de termino de processamento ou, como veremos na próxima seção, quando desejamos trabalhar com falhas.

Conceitos desta seção

Conteúdo	Descrição
`Event.callbacks.append(callbackFunction)`	adiciona um `callback` , representado pela função `callbackFunction` do modelo. Após o processamento do evento (`Event.processed = True`) a função de `callback` é executada. A função `callbackFunction,` obrigatoriamente deve ter como parâmetro apenas um evento.

Desafio

> Desafio 26: acrescente à função `final` um comando para armazenar quem venceu e em que tempo. Simule para 10 replicações e calcule a média e a porcentagem de vitórias de cada corredor.

Solução do desafio 26

> Desafio 26: acrescente à função `final` um comando para armazenar quem venceu e em que tempo. Simule para 10 replicações e calcule a média e a porcentagem de vitórias de cada corredor.

Neste caso, podemos criar uma lista `resultadoList` para armazenar a `tuple` (tempo, vencedor) de cada replicação. Os valores podem ser armazenados no instante em que a função `campeão` identifica o vencedor, isto é:

```
def campeao(event):
    global vencedor

    if not vencedor:
        # imprime a faixa de campeão
        print('%3.1f \o/ Tan tan tan (musica do Senna) A %s é a campeã!'
                    %(env.now, event.value))
        # atualiza a variável global vencedor
        vencedor = True
        # armazena o resultado na lista resultadoList
        resultadoList.append((env.now, event.value))

    else:
        # imprime o perdedor
        print('%3.1f A %s chega em segundo lugar...'
                    %(env.now, event.value))
```

Como o desafio pede que sejam executadas 10 replicações da corrida, devemos modificar a chamada do modelo para refletir isso. Inicialmente, criamos a lista `resultadoList` e, a seguir, criamos um laço para executar o modelo 10 replicações:

```
# lista para armazenar os resultados das corridas/replicações
resultadoList = []
random.seed(10)

# processa 10 replicações
for i in range(10):
    # True, se já temos um vencedor na corrida, False caso contrário
    vencedor = False
    env = simpy.Environment()
    proc = env.process(corrida(env))
    proc.callbacks.append(final)
    # executa até que o processo corrida termine
    env.run(proc)
```

Note que a variável global `vencedor` é atualizada dentro do laço (na seção seguinte, ela era criada logo no cabeçalho do modelo), afinal, a cada replicação precisamos garantir que as variáveis do modelo sejam reinicializadas.

Uma vez que as 10 replicações tenham sido executadas, basta calcular as estatísticas desejadas. O número de vitórias é facilmente extraída da lista de vencedores:

```
# separa a lista de resultados em duas listas
tempos, vencedores = zip(*resultadoList)
# conta o número de vitórias
lebreWin = vencedores.count('lebre')
tartarugaWin = vencedores.count('tartaruga')

# calcula a percentagem de vitórias da lebre
lebreP = lebreWin/(lebreWin+tartarugaWin)
```

Para o cálculo da média do tempo de corrida, temos diversas possibilidades. A escolhida aqui foi utilizar a função `mean` da biblioteca `numpy`:

```
# importa a função para cálculo da média a partir biblioteca numpy
from numpy import mean

print('Tempo médio de corrida: %3.1f\tLebre venceu: %3.1f%%\tTartaruga venceu: %3.1f%%'
       % (mean(tempos), lebreP*100, (1-lebreP)*100))
```

Quando executado, o modelo fornece como resposta:

```
0.0 Iniciada a corrida!
5.3 \o/ Tan tan tan (música do Senna) A tartaruga é a campeã!
5.3 Ok pessoal, a corrida acabou.
0.0 Iniciada a corrida!
5.1 \o/ Tan tan tan (música do Senna) A tartaruga é a campeã!
5.1 Ok pessoal, a corrida acabou.
0.0 Iniciada a corrida!
4.4 \o/ Tan tan tan (música do Senna) A tartaruga é a campeã!
4.4 Ok pessoal, a corrida acabou.
0.0 Iniciada a corrida!
6.1 \o/ Tan tan tan (música do Senna) A lebre é a campeã!
6.1 Ok pessoal, a corrida acabou.
0.0 Iniciada a corrida!
5.4 \o/ Tan tan tan (música do Senna) A lebre é a campeã!
5.4 Ok pessoal, a corrida acabou.
0.0 Iniciada a corrida!
0.5 \o/ Tan tan tan (música do Senna) A lebre é a campeã!
0.5 Ok pessoal, a corrida acabou.
0.0 Iniciada a corrida!
4.8 \o/ Tan tan tan (música do Senna) A lebre é a campeã!
4.8 Ok pessoal, a corrida acabou.
0.0 Iniciada a corrida!
3.5 \o/ Tan tan tan (música do Senna) A lebre é a campeã!
3.5 Ok pessoal, a corrida acabou.
0.0 Iniciada a corrida!
4.9 \o/ Tan tan tan (música do Senna) A lebre é a campeã!
4.9 Ok pessoal, a corrida acabou.
0.0 Iniciada a corrida!
5.4 \o/ Tan tan tan (música do Senna) A tartaruga é a campeã!
5.4 Ok pessoal, a corrida acabou.

Tempo médio de corrida: 4.5     Lebre venceu: 60.0%     Tartaruga venceu: 40.0%
```

Teste seu conhecimento

1. Acrescente o cálculo do intervalo de confiança ao resultados do tempo médio de corrida e simule um número de replicações suficientes para garantir que este intervalo seja no máximo de 5% em torno da média, para um nível de significância de 95%.
2. Reformule o modelo para que ele aceite uma lista de corredores diferentes, não só uma lebre e uma tartaruga.

Interrupções de eventos

De modo semelhante ao que vimos com recursos, os eventos também podem ser interrompidos em SimPy. Como o SimPy aproveita-se dos comandos de interrupção já existentes no Python, pode-se utilizar o bloco `try... except` e assim capturar a interrupção em qualquer parte do modelo.

Considere um exemplo simples em que um jovem Jedi tem sua seção de meditação matinal interrompida por um cruel Lord Sith interessado em convertê-lo para o Lado Negro.

Construir um modelo de simulação deste sistema, é tarefa simples: precisamos de uma função que representa o Jedi meditando e outra que interrompe o processo em determinado momento (lembre-se que um processo é também um evento para o SimPy).

O SimPy tem dois métodos diferentes para interroper um evento: `interrupt` ou `fail.`

Interrompendo um evento com o método `interrupt`

Criaremos duas funções: `forca` que representa o processo de meditação e `ladoNegro` que representa o processo de interrupção da meditação. Inicialmente, interromperemos o processo `forca` por meio do método `interrupt`. Uma possível máscara para o modelo ficaria:

```
import simpy

def forca(env):
    # processo a ser interrompido

def ladoNegro(env, proc):
    # gerador de interrupção do processo proc

env = simpy.Environment()

forcaProc = env.process(forca(env))
ladoNegroProc = env.process(ladoNegro(env, forcaProc))
```

Para este exemplo, o processo de meditação é bem simples, pois estamos mais interessados em aprender sobre interrupções:

```
def forca(env):
    # processo a ser interrompido
    while True:
        yield env.timeout(1)
        print('%d Eu estou com a Força e a Força está comigo.' % env.now)
```

Portanto, a cada 1 unidade de tempo, o Jedi fala um frase para manter-se concentrado. O processo de interrupção por sua vez, recebe como parâmetro de entrada o processo (ou evento) a ser interrompido. Apenas para ilustrar melhor o exemplo, vamos considerar que após 3 unidades de tempo, a função interrompe o processo (ou evento) de meditação:

```
def ladoNegro(env, proc):
    # gerador de interrupção do processo proc
    yield env.timeout(3)
    print('%d Venha para o lado negro da força, nós temos CHURROS!' % env.now)
    # interrompe o processo proc
    proc.interrupt()
    print('%d Welcome, young Sith.' % env.now)
```

Portanto, a interrupção de um evento ou processo qualquer é invocada pelo método `.interrupt()`. Por exemplo, dado que processo ou evento `proc`, podemos interrompê-lo com a linha de código:

```
# interrompe o processo proc
proc.interrupt()
```

Se você executar o modelo anterior, a coisa até começa bem, mas depois surge uma supresa desagradável:

```
1 Eu estou com a Força e a Força está comigo.
2 Eu estou com a Força e a Força está comigo.
3 Venha para o lado negro da força, nós temos CHURROS!
3 Welcome, young Sith.
Traceback (most recent call last):

  File "<ipython-input-8-623c3bea2882>", line 1, in <module>
    runfile('C:/Book/Interruption/meditation.py', wdir='C:/Book/Interruption/')

  File "C:\Anaconda3\lib\site-packages\spyderlib\widgets\externalshell\sitecustomize.py", line
714, in runfile
    execfile(filename, namespace)

  File "C:\Anaconda3\lib\site-packages\spyderlib\widgets\externalshell\sitecustomize.py", line
89, in execfile
    exec(compile(f.read(), filename, 'exec'), namespace)

  File "C:/Book/Interruption/meditation.py", line 22, in <module>
    env.run()

  File "C:\Anaconda3\lib\site-packages\simpy\core.py", line 137, in run
    self.step()

  File "C:\Anaconda3\lib\site-packages\simpy\core.py", line 229, in step
    raise exc

Interrupt: Interrupt(None)
```

O que essa longa mensagem de erro nos faz lembrar é que o método `.interrupt()` vai além de interromper um mero evento do SimPy, ele interrompe o programa todo.

Mas, jovem leitor Jedi, temos duas maneiras de contornar o problema: com a lógica de controle de exceção do Python `try...except` ou com a propriedade `.defused,` como veremos a seguir.

Método de controle de interrupção 1: lógica de exceção `try... except`

Neste caso, a solução é razoavelmente simples, basta acrescentarmos ao final do programa (ou em outra parte conveniente) uma lógica de exceção do SimPy, `simpy.Interrupt,` como no exemplo a seguir:

```
import simpy

def forca(env):
    # processo a ser interrompido
    while True:
        yield env.timeout(1)
        print('%d Eu estou com a Força e a Força está comigo.' % env.now)

def ladoNegro(env, proc):
    # gerador de interrupção do processo proc
    yield env.timeout(3)
    print('%d Venha para o lado negro da força, nós temos CHURROS!' % env.now)
    # interrompe o processo proc
    proc.interrupt()
    print('%d Welcome, young Sith.' % env.now)

env = simpy.Environment()

forcaProc = env.process(forca(env))
ladoNegroProc = env.process(ladoNegro(env, forcaProc))

try:
    env.run()
except simpy.Interrupt:
    print('%d Eu estou com a Força e a Força está comigo.' % env.now)
```

Quando executado, o modelo anterior fornece:

```
1 Eu estou com a Força e a Força está comigo.
2 Eu estou com a Força e a Força está comigo.
3 Venha para o lado negro da força, nós temos CHURROS!
3 Welcome, young Sith.
3 Eu estou com a Força e a Força está comigo.
```

É importante notar que depois da interrupção `proc.interrupt()` o modelo ainda executa a última linha do processo `ladoNegro` (basicamente, imprime "Welcome, young Sith") para, a seguir, executar o comando dentro do `except simpy.Interrupt.`

Método de controle de interrupção 2: alterando o atributo `defused`

No caso anterior, o leitor deve ter notado que, ao interromper o processo, interrompemos a simulação por completo, pois nossa lógica de exceção está ao final do código.

E se quisésemos apenas paralizar o processo (ou evento) sem que isso impactasse em toda a simulação? Neste caso, SimPy fornece um atributo `defused` para cada evento que, quando alterado para `True`, faz com que a interrupção seja "desarmada".

Vamos alterar o atributo `defused` do processo interrompido no exemplo anterior:

```
import simpy

def forca(env):
    # processo a ser interrompido
    while True:
        yield env.timeout(1)
        print('%d Eu estou com a Força e a Força está comigo.' % env.now)

def ladoNegro(env, proc):
    # gerador de interrupção do processo proc
    yield env.timeout(3)
    print('%d Venha para o lado negro da força, nós temos CHURROS!' % env.now)
    # interrompe o processo proc
    proc.interrupt()
    # defused no processo para evitar a interrupção da simulação
    proc.defused = True
    print('%d Welcome, young Sith.' % env.now)

env = simpy.Environment()

forcaProc = env.process(forca(env))
ladoNegroProc = env.process(ladoNegro(env, forcaProc))

env.run()
```

Quando executado, o modelo anterior fornece:

```
1 Eu estou com a Força e a Força está comigo.
2 Eu estou com a Força e a Força está comigo.
3 Venha para o lado negro da força, nós temos CHURROS!
3 Welcome, young Sith.
```

Novamente a execução do processo de interrupção vai até o fim e a interrupção que poderia causar a paralização de todo o modelo é desarmada.

Portanto, se o objetivo é *desarmar* a interrupção, basta tornar `True` o atributo `defused` do evento.

Interrompendo um evento com o método `fail`

De modo semelhante a provocar um interrupção, podemos provocar uma *falha* no evento. O interessante, neste caso, é que podemos informar a falha

O que são funções geradoras? (Ou entendendo como funciona o SimPy) - Parte I

O comando `yield` é quem, como você já deve ter notado, dá o *ritmo* do seu modelo em SimPy. Para melhor compreender como funciona um programa em SimPy, precisamos entender, além do próprio comando `yield`, outro conceito fundamental em programação: as funções geradoras.

Começaremos pelo conceito de Iterador.

Iterador

Na programação voltada ao objeto, *iteradores* são *métodos* que permitem ao programa observar os valores dentro de um dado objeto.

Quando você percorre uma lista com o comando `for`,
por exemplo, está intrinsecamente utilizando um iterador:

```
lista = [10, 20, 30]
for i in lista:
   print (i)

10
20
30
```

No exemplo, `lista` é um *objeto* e o comando `for` é um **iterador** que permite vasculhar cada elemento dentro da lista, retornando sempre o elemento seguinte do objeto.

Funções geradoras

Elas existem e (estão entre nós há tempos...)

Uma **função geradora** é uma classe especial de função que tem como característica retornar, cada vez que é chamada, valores em sequência. O que torna uma função qualquer uma *função geradora* é a presença do comando `yield` em seu corpo.

O comando `yield` funciona, a primeira vez que a função é chamada, algo semelhante a um `return`, mas com um **superpoder:** toda vez que a função é chamada novamente, a execução começa a partir da linha seguinte ao `yield`.

Por exemplo, podemos imprimir os mesmo números da lista anterior, chamando 3 vezes a função `seqNum()`, definida a seguir:

```
def seqNum():
    n = 10
    yield n
    n += 10
    yield n
    n += 10
    yield n

for i in seqNum():
 print(i)

10
20
30
```

Note que a *função geradora* `seqNum` é um *objeto* e que o loop `for` permite acessar os elementos retornados por cada `yield`.

A primeira fez que o loop `for` chamou a função `seqNum()` o código é executado até a linha do `yield n,` que retorna o valor `10` (afinal, n=10 neste momento).

A segunda fez que o loop `for` chama a função, ela não recomeça da primeira linha, de fato, ela é executada a partir da **linha seguinte ao comando** `yield` e retorna o valor `20`, pois `n` foi incrementado nessa segunda passagem.

Na terceira chamada à função, a execução retoma a partir da linha seguinte ao segundo `yield` e o próximo valor de *n* será o anterior incrementado de 10.

Uma função geradora é, de fato, um *iterador* e você normalmente vai utilizá-la dentro de algum *loop* `for` como no caso anterior. Outra possibilidade é você chamá-la diretamente pelo comando `next` do Python, como será visto no próximo exemplo.

Exemplo: Que tal uma função que nos diga a posição atual de um Zumbi que só pode andar uma casa por fez no plano? A função geradora a seguir acompanha o andar cambaleante do zumbi:

```
import random

def zombiePos():
    x, y, = 0, 0 # posição inicial do zumbie
    while True:
        yield x, y,  "Brains!"
        x += random.randint(-1, 1)
        y += random.randint(-1, 1)

zombie = zombiePos()

print(next(zombie))
print(next(zombie))
print(next(zombie))
```

Diferentemente do caso anterior, criamos um zumbi a partir da linha:

```
zombie = zombiePos()
```

Cada novo passo do pobre infeliz é obtido pelo comando:

```
next(zombie))
```

O bacana, no caso, é que podemos criar 2 zumbis passeando pela relva:

```
import random

def zombiePos():
    x, y, = 0, 0 # zombie initial position
    while True:
        yield x, y, "Brains!"
        x += random.randint(-1, 1)
        y += random.randint(-1, 1)

zombie1 = zombiePos()
zombie2 = zombiePos()
print(next(zombie1), next(zombie2))
print(next(zombie1), next(zombie2))
print(next(zombie1), next(zombie2))
```

Na seção a seguir discutiremos o papel da função geradora em um modelo de simulação. Por ora, sugerimos as seguintes fontes de consulta caso você procure um maior aprofundamento sobre o `yield` e as funções geradoras:

- PEP 225 - Descritivo técnico do yield no Python
- Utilizando Geradores na Python Brasil
- Uma boa explicação na StackOverflow: What does the yield keyword do?

O que são funções geradoras? (Ou como funciona o SimPy?) - Parte II

SimPy vs. funções geradoras

No episódio anterior...

> Uma **função geradora** é uma classe especial de funções que têm como característica retornar, cada vez que são chamadas, valores em sequência. O que torna uma função qualquer uma *função geradora* é a presença do comando `yield` em seu corpo.

Para compreendermos a mecânica do SimPy (e da maioria dos softwares de simulação), devemos lembrar que os processos de um modelo de simulação nada mais são que eventos (ou atividades ou ações) que interagem entre si de diversas maneiras, tais como: congelando outro evento por tempo determinado, disparando novos eventos ou mesmo interrompendo certo evento já em execução.

Já sabemos que as entidades e eventos em SimPy são modelados como **processos** dentro de um dado **environment**. Cada processo é basicamente uma função iniciada por `def` como qualquer outra construída em Python, mas que contém a palavrinha mágica `yield.` Assim, como descrito no item anterior, todo **processo** em SimPy é também uma **função geradora**.

Um evento bastante elementar em SimPy é o `timeout()` ou, na sua forma mais usual:

```
yield env.timeout(tempo_de_espera)
```

Imagine por um momento que você é a própria encarnação do SimPy, lidando com diversos eventos, processos, recursos etc. Repentinamente, você, Mr. SimPy, depara-se com a linha de código anterior. Mr. SimPy vai processar a linha em duas etapas principais:

1. A palavra `yield` suspende imediatamente o processo ou, de outro modo, impede que a execução avance para linha seguinte (como esperado em toda função geradora);
2. Com o processo suspenso, a função `env.timeout(tempo_de_espera)` é executada e só após o seu derradeiro término, o processamento retorna para a linha seguinte do programa.

Portanto, quando um processo encontra um `yield`, ele é suspenso até o instante em que o evento deve ocorrer, quando o SimPy então *dispara* o novo evento. O que o SimPy faz no caso é **criar um evento a ser disparado** dali a um tempo igual ao `tempo_de_espera.`

Naturalmente, quando num modelo de simulação temos muito eventos interpostos, cabe ao SimPy coordenar os disparos e suspensões dos eventos corretamente ao longo da simulação, respeitando um calendário único do programa - é nesta parte que você deve se emocionar com a

habilidade dos programadores que codificaram o calendário de eventos dentro do SimPy...

Em resumo, SimPy é um controlador de eventos, gerados pelo seu programa. Ele recebe seus eventos, ordena pelo momento de execução correto (ou prioridade, quando existem eventos simultâneos no tempo) e armazena uma lista de eventos dentro do `environment.` Se uma função dispara um novo evento, cabe ao SimPy adicionar o evento na lista de eventos, de modo ordenado pelo momento de execução (ou da prioridade daquele evento sobre os outros).

Simulação de Agentes em SimPy!

APOCALIPSE ZUMBI!

Entrada e saída de dados por planilha eletrônica