PREFEITURA DE
SANTANA DE PARNAÍBA

CURSO PROFISSIONALIZANTE DE PYTHON

Secretaria Municipal
de Emprego, Desenvolvimento,
SEMEDES Ciência, Tecnologia e Inovação

investe Parnaíba

Sumário

Introdução ao Curso

Esse curso tem a ideia de demonstrar aos alunos as possibilidades de uma das linguagens mais completas e que mais tem crescido no mundo da tecnologia. Python tem mil e uma utilidades, desde a programação de scripts que automatizam algumas tarefas até mesmo o Machine Learning(Aprendizado de Máquina).

Durante as sete semanas de curso os alunos terão uma visão do mercado de tecnologia, a introdução ao desenvolvimento de aplicações e a linguagem Python.

Nivelamento de Conhecimentos

O que é Lógica de Programação e a sua importância em nossas tarefas cotidianas

Tudo o que fazemos em nossas vidas tem relação com sequências lógicas e nem percebemos, pois fazemos tudo de forma automática. Você sabia disso? Não? Posso provar o que estou falando. Imagine o processo de ir tomar banho? Ou então, trocar uma lâmpada… ainda não acredita? Pense como você fez para chegar até aqui.

Conseguiu pensar? Refaça os passos e tarefas feitas até chegar aqui hoje. Vamos fazer um exercício simples, com suas palavras, descreve os processos para:

- Tomar banho.
- Trocar uma lâmpada.
- Como você fez para chegar aqui.

Tenha em mente que quanto mais detalhado melhor. Ao final do exercício, o professor vai pedir para que alguns de vocês leiam.

IMPORTANTE - preste muita atenção no que seus amigos estão apresentando, afinal cada pessoa é única e os seus processos para executar as mesmas tarefas que você serão diferentes, e tudo bem.

Vamos para mais um exemplo: o café que tomamos de manhã. Quando perguntam como tomamos nosso café, a maioria de nós responde que, ao acordarmos, preparamos o café com auxílio de uma cafeteira elétrica, colocamos ele em uma caneca e o tomamos.
Mas, ao destrinchar este processo, somos capazes de estipular uma sequência de passos que nos levaram ao ato final de beber este café. Esta sequência pode ser:

- Ao acordar, levanto da cama;

- Após levantar da cama, desço as escadas;
- Após descer as escadas, entro na cozinha;
- Após entrar na cozinha, pego o pó de café no armário;
- Após pegar o pó de café, o coloco dentro da cafeteira;
- Após colocar o pó na cafeteira, jogo água no compartimento específico;
- Após inserir todos os ingredientes na máquina, aperto o botão de ligar;
- Quando o café está pronto, pego a garrafa;
- Após pegar a garrafa, despeje o café dentro de uma caneca;
- Após colocar o café na caneca, bebo o café.

E agora, você pode notar o uso da Lógica em nosso dia-a-dia? Pensando nessa organização dos passos que precisamos dar para atender alguma necessidade que encontramos no decorrer das nossas tarefas nos deparamos com a Lógica de Programação, que nada mais é do que organizarmos o nosso pensamento para que fique de uma forma que o computador e a linguagem que escolhemos entendam o que estamos querendo dizer/fazer e dessa forma executar o que precisamos.

NOTA - Cada linguagem tem suas próprias particularidades, como sua sintaxe, seus tipos de dados e sua orientação, mas a lógica por trás de todas é a mesma.

E por que a Lógica de Programação é Importante?

Tem uma frase do Steve Jobs que é incrível e resume muita coisa:

> *"Everybody in this country should learn how to program a computer because it teaches you how to think."*

Traduzindo ela de forma muito literal ficaria algo como:

> *"Todo mundo neste país deveria aprender a programar porque isso te ensina a pensar."*

É muito comum que muitos estudantes de programação se perguntem por que a lógica de programação é tão importante.

A lógica de programação é importante porque é ela quem nos dá as ferramentas necessárias para executar o processo mais básico no desenvolvimento de alguma aplicação: a criação de seu **algoritmo**.

Antes de criar um software do zero, ou de resolver um problema de um já existente, é necessário descascá-lo até chegarmos ao seu núcleo. Em outras palavras, precisamos [1]compreendê-lo completamente, desde suas funções a seus objetivos finais — ou seja, pesquisar, rascunhar, dominá-lo na íntegra.[2]

Organização

Ao aprendermos a pensar logicamente, tendemos a uma maior organização de alguns processos. Seja em nossos ambientes pessoais ou dentro de nossa mente, a ordem é fator

1
2

determinante para que possamos render mais e ser mais produtivos ao realizar nossas tarefas. Com lógica de programação, a sua maneira de pensar irá mudar e, com isso, seus processos externos também serão positivamente impactados.

Raciocínio lógico

O raciocínio lógico de um programador é uma de suas maiores ferramentas de trabalho. Uma máquina é incapaz de compreender ordens não-lógicas, do mesmo modo que são incapazes de raciocinar sobre regras dispostas de maneira desordenada.

E é para isso que a lógica de programação existe: ao aguçar o raciocínio lógico do programador, ele está mais próximo da *maneira de pensar* de um computador e, portanto, mais habilidoso na hora de desenvolver um código eficiente.

Quanto mais capazes de compreender as coisas ao nosso redor de maneira técnica nós formos, mais eficiente será nosso raciocínio lógico.

Resolução de problemas

Nem só de criação vive um programador; a resolução de problemas também é muito comum em sua jornada de trabalho.

Sob essa perspectiva, podemos dizer que a programação também pode ser entendida como o processo de dividir um problema complexo em pequenas partes para, então, resolvê-las gradualmente a partir de trechos de código.

Sem lógica da programação isso não seria possível, uma vez que o desenvolvimento de um algoritmo de resolução de problemas depende diretamente dela.

Concentração

Como falamos anteriormente, quando nossa maneira de pensar muda, mudam também as formas com que executamos nossos processos externos. A concentração, por exemplo, é um deles.
Se você tem problemas em atingir um bom nível de concentração na hora de trabalhar ou estudar, compreender lógica de programação pode te ajudar.

Isso acontece porque, quanto mais claras as ações que precisamos desempenhar para atingir determinados objetivos, podemos ordená-las e executá-las, uma a uma, de maneira mais categórica e, por consequência, com mais concentração.

Entender a tecnologia

Já dissemos hoje que os computadores são incapazes de compreender ordens subjetivas e que não estejam estritamente ordenadas de acordo com a sua maneira de compreender, não é mesmo?

A partir disso, podemos concluir que, ao estudar lógica de programação, estamos estudando também a maneira como a tecnologia funciona de modo geral. Isso porque todos os processos existentes em TI dependem de um código que os sustenta.
Consequentemente, entender a lógica da programação é um caminho muito eficaz para compreender a tecnologia como um todo.

O papel dos algoritmos na lógica de programação

Para quem é leigo, a palavra algoritmo chega a assustar. Muitas vezes temos a ideia de que um algoritmo é um sistema super complexo e de difícil compreensão.

Esta é somente uma meia verdade. Algoritmos podem ser sistemas mais robustos — e um bom exemplo disso é o algoritmo do Google, extremamente poderoso e preciso ao entregar os resultados de busca a um usuário.

Mas nem todo algoritmo está neste nível de importância.

Em programação, algoritmos são um conjunto de instruções que um software ou aplicação deve seguir para executar uma tarefa, resolver um problema ou chegar a um objetivo distinto.

Pense em um algoritmo como uma receita de bolo, por exemplo. **Uma receita de bolo que nada mais é do que uma sequência lógica de etapas que, quando realizadas da maneira correta, resultam em uma deliciosa sobremesa.**

Do mesmo modo, um algoritmo, quando executado através de um código que o permite funcionar com excelência, resulta na resolução de um problema.

Como entender a lógica de programação?

A maneira mais eficiente de compreender a lógica de programação é com a sua **aplicação**, ou seja: programando.

A teoria deve ser estudada profundamente, mas a aplicação dará a um programador uma dimensão mais ampla e mais completa sobre a utilidade destes conceitos.

Com esta percepção mais aguçada, um programador torna-se mais apto a encontrar soluções mais inteligentes e simples de maneira mais rápida.

Entender lógica de programação faz parte do processo de entender programação como um todo e, para isso, é preciso conhecer todos os aspectos deste processo.

Variáveis e constantes

Constantes e variáveis são espaços de memória reservados em uma máquina para a manutenção de determinados dados.

Estes dados serão armazenados na memória de um software enquanto ele for executado. Na maioria dos casos, após o seu encerramento, estes dados deixarão de existir.

Os dados constantes, são fixos; os variáveis, variam.
Por exemplo: ao desenvolver um pequeno código para executar uma soma, podemos trabalhar com três variáveis.

As chamarei de X, Y e Z (o resultado)

As variáveis e constantes podem ser representadas por alguns tipos de dados. Vou colocar alguns abaixo.

Tipos de dados

Os tipos de dados são os formatos em que as constantes ou variáveis se apresentam.
Veja abaixo os quatro tipos mais comuns de dados utilizados em programação.

Texto

Este dado se refere aos dados que se apresentam através de uma sequência de caracteres de texto que possuem apenas letras.

Não é possível representar um dado de texto com números ou caracteres especiais, como símbolos.

Inteiro

O dado **inteiro** é a primeira classificação de dados **numéricos**. Este dado serve aos números inteiros, ou seja, sem divisões.

Podem ser positivos ou negativos, como por exemplo 25; 40; 78; -5 e etc.

Real

O dado **real** é a segunda classificação de dados **numéricos**. Este dado serve aos números quebrados, ou seja, que possuem casas decimais.

Também podem ser positivos ou negativos, como por exemplo 3,14; 2,5; -8,9 e etc.

Lógico

Este tipo de dado define constantes ou variáveis *booleanas* que exprimem condições; por exemplo, VERDADEIRO ou FALSO.

Estrutura de seleção e repetição

Uma estrutura de seleção e repetição é uma estrutura que permite que um trecho de código seja executado mais de uma vez. Estas estruturas também exprimem condições.

Os tipos de dados das estruturas de seleção costumam ser representados por ***if*** ou ***switch***. Para as estruturas de repetição, os dados costumam ser representados por ***while***, ***for*** e ***do-while***.

Pseudo-código

Pseudocódigo é uma forma genérica de escrever um algoritmo, utilizando uma linguagem simples (nativa a quem o escreve, de forma a ser entendida por qualquer pessoa) sem necessidade de conhecer a sintaxe de nenhuma linguagem de programação.

É aqui que o conceito de algoritmo (sequência lógica, um passo a passo para resolução de problemas) se mostra. Mas ele não se restringe ao mundo da computação, e se amplifica para diversas outras áreas. Quer um exemplo simples e prático? Uma receita de torta é, de certa forma, um algoritmo. Mas como assim?!

Uma receita, seja de torta, bolo ou qualquer outro preparo, depende de uma sequência de instruções ou passos para chegar ao resultado ideal.

É um conceito bastante similar ao de desenvolvimento de sistemas, afinal, quem programa precisará passar para o computador instruções claras, utilizando a linguagem escolhida, para que o software execute os comandos.

Como fazer um algoritmo em Pseudocódigo?

Escrever com pseudocódigo requer que você conheça alguns comandos básicos.

- **escreva (“ ”)** = *comando usado para imprimir uma mensagem na tela*;
- **leia ()** = *comando usado para ler valores digitados no teclado*;
- **inicio** = *comando para iniciar o programa principal*;
- **algoritmo** = *comando para indicar o início do programa*;
- **fimalgortimo** = *comando para finalizar o algoritmo*;
- **var** = *comando para declarar variáveis*.

Fluxograma

Fluxograma: é um tipo de diagrama, e pode ser entendido como uma representação esquemática de um processo ou algoritmo, muitas vezes feito através de gráficos que

ilustram de forma descomplicada a transição de informações entre os elementos que o compõem, ou seja, é a sequência operacional do desenvolvimento de um processo, o qual caracteriza: o trabalho que está sendo realizado, o tempo necessário para sua realização, a distância percorrida pelos documentos, quem está realizando o trabalho e como ele flui entre os participantes deste processo.

Os fluxogramas são muito utilizados em projetos de software para representar a lógica interna dos programas, mas podem também ser usados para desenhar processos de negócio e o workflow que envolve diversos atores corporativos no exercício de suas atribuições.

Estruturas Condicionais

Condicional Simples

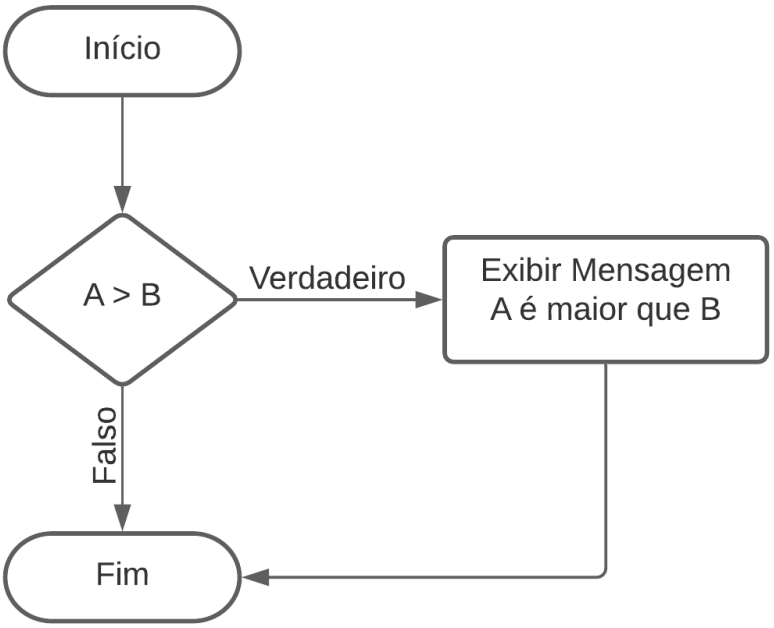
Ao iniciar os estudos em programação nos deparamos com as estruturas condicionais e de repetição, que são dois pilares em relação a lógica de programação.

Primeiramente, uma estrutura condicional é baseada em uma condição que se for atendida o algoritmo toma uma decisão. Nós podemos representar uma estrutura condicional conforme o algoritmo abaixo.

```
SE (a > b) ENTÃO
        Exibir mensagem "o número A é maior que o número B".
FIM SE
```

Note que o código acima representa a seguinte condição: se o número A for maior que o número B, o algoritmo irá entender que a condição é **verdadeira** e deve exibir a mensagem “o número A é maior que o número B”, se esta condição não for atendida, ou seja, se ela for **falsa**, o algoritmo não irá tomar nenhuma ação, pois ela não atende a condição.

Para facilitar o entendimento, podemos representar a estrutura condicional acima utilizando um fluxograma:



Caso o algoritmo precise tomar uma decisão dependendo da estrutura de condição, nós podemos utilizar a estrutura condicional composta.

Estruturas condicionais composta(SENÃO - ELSE)

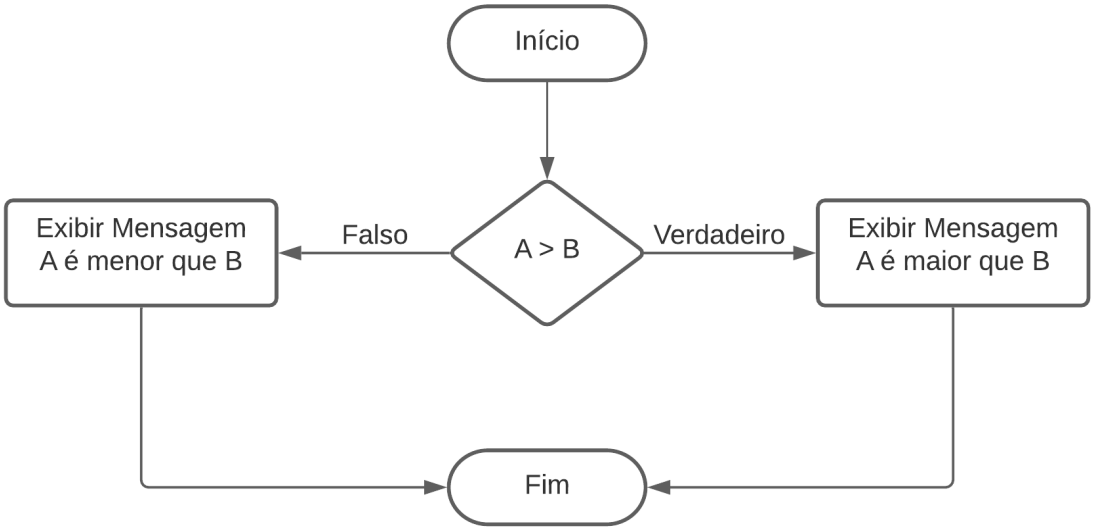
A diferença em relação a estrutura condicional simples é que se a condição for **falsa** nosso algoritmo também irá tomar uma ação neste caso, seguindo o exemplo anterior:

```
SE (a > b) ENTÃO
        Exibir mensagem "o número A é maior que o número B".
SENÃO
        Exibir mensagem "o número A é menor que o número B".
```

Note que o código acima representa a seguinte condição: se o número A for maior que o número B, o algoritmo irá entender que a condição é **verdadeira** e deve exibir a mensagem “o número A é maior que o número B”, se esta condição não for atendida, ou seja, se ela for **falsa**.

Diferente do exemplo sobre estrutura condicional simples onde o algoritmo não tomava nenhuma ação, aqui ele toma uma decisão diferente, exibindo a mensagem “o número A é **menor** que o número B”.

Para facilitar o entendimento, podemos representar a estrutura condicional acima utilizando um fluxograma:



Estruturas de repetição

Para situações que será necessário repetir uma tarefa mais de uma vez, podemos contar com as estruturas de repetição.

ENQUANTO(While)

Uma das condições que podemos usar é o **enquanto**, ou seja, enquanto a expressão booleana for verdadeira o algoritmo executa o bloco proposto, por fim é necessário que algo dentro do bloco altere a condição.

Podemos exemplificar o uso do **enquanto** supondo um algoritmo que retorne o resultado de uma tabuada, onde:

i sendo uma variável inteira de valor 1.

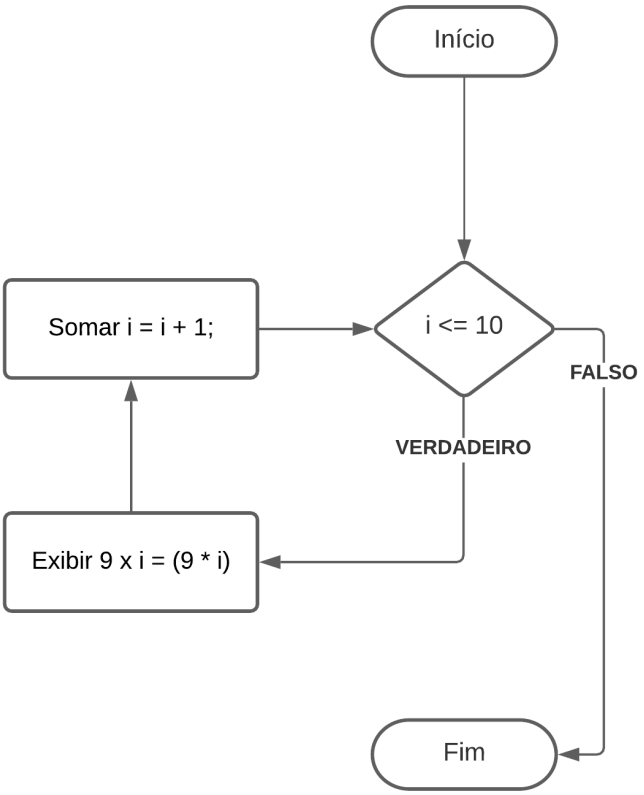
```
ENQUANTO i ≤ 10
        Exibir "9 x i = (9 * i)"
        Soma i = i + 1;
FIM DA CONDIÇÃO
```

Note que a condição que empregamos no algoritmo é de que a variável i irá repetir o bloco até que o valor dela seja igual a 10. **Enquanto** isso será exibido a mensagem “9 x i = (9 * i)”, a cada linha executada é somado a variável i o valor 1, essa soma é chamada de **iteração**. Na maioria das linguagens pode-se referenciar a interação com a sintaxe i++ (*variável seguida de dois sinais de mais*) ou *variável = variável + 1*.

Quando i chegar ao valor 11 o algoritmo irá parar de executar o bloco, desta forma o resultado esperado do exemplo acima será:

```
9 x 1 = 9
9 x 2 = 18
9 x 3 = 27
9 x 4 = 36
9 x 5 = 45
9 x 6 = 54
9 x 7 = 63
9 x 8 = 72
9 x 9 = 81
9 x 10 = 90
```

Para facilitar o entendimento, podemos representar a estrutura de repetição acima utilizando um fluxograma:



Outro laço que podemos utilizar é o PARA (for).

PARA (FOR)

A condição **PARA** tem o mesmo princípio que utilizar **enquanto (while),** porém este recurso é mais utilizado quando se sabe o número de iterações da repetição, como listar os valores de um vetor por exemplo, também vale ressaltar a legibilidade do código mais limpo.

Seguindo o exemplo acima da tabuada, o algoritmo esperado será:

```
PARA i = 1; i <= 10; i++;
        Exibir "9 x i = (9 * i)"
```

Note que ao usar o **PARA (for)** o contador (a variável i) é inicializado e incrementado na própria condição do laço junto a expressão booleana a ser atendida, deixando o código mais limpo e sem a necessidade de criar variáveis adicionais.

No fim o retorno aguardado será o mesmo que do exemplo que utilizamos a do laço **enquanto**.

Orientação a Objetos - Introdução

A OOP (programação orientada a objeto) é um paradigma de programação que pode ser usado para modelar o mundo real no código. Há vários benefícios no uso desse paradigma. Com a OOP, você pode criar implementações fáceis de modificar e estender com menos código.

Como parte de uma equipe de desenvolvimento, você escreverá um código que resolve problemas pequenos, juntamente com problemas grandes e mais complexos. A OOP é um paradigma de programação entre vários existentes, mas tem alguns constructos e princípios interessantes que ajudarão a estruturar seu pensamento.

Você decide avaliar a OOP aplicando-a a um problema pequeno e definido (um jogo de pedra, papel e tesoura) para ver se ele é adequado para você e sua equipe.

O que é programação orientada a objeto?

A OOP (programação orientada a objeto) é um paradigma de programação. Ela se baseia na ideia de *agrupar* dados e funções relacionados em "ilhas" de informações. Essas ilhas são conhecidas como *objetos*.

Não importa qual paradigma é usado, os programas usam a mesma série de etapas para resolver problemas:

1. Entrada de dados: os dados são lidos de algum lugar, que pode ser o armazenamento de dados como um sistema de arquivos ou um banco de dados.
2. Processamento: os dados são interpretados e possivelmente alterados para serem preparados para exibição.
3. Saída de dados: os dados são apresentados para que um usuário físico ou um sistema.

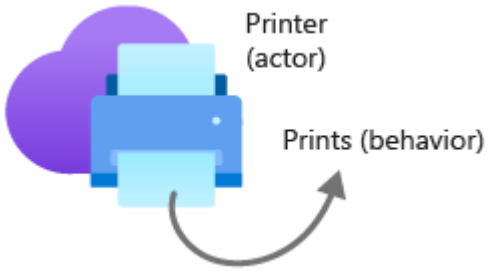
OOP versus programação procedural

4. Vamos tentar definir o que é OOP comparando-a com outro paradigma: programação procedural. A programação procedural se dispõe a resolver um determinado problema chamando procedimentos, que também são conhecidos como funções ou métodos. Funções e variáveis são construídas para abordar as várias fases descritas nas etapas anteriores.
5. O paradigma OOP não é diferente nesse aspecto. O que o destaca é a maneira como examina o mundo. Em comparação com a programação procedural, a OOP dá um passo para trás e examina todo o panorama. Em vez de trabalhar nos dados e levá-los de uma fase para a seguinte, a OOP tenta entender o mundo em que os dados são operados. Ela faz isso *modelando* o que vê.
    6. tema possa lê-los e interagir com eles.

Modelagem de OOP: identificar conceitos

Durante a fase de modelagem, você examina uma descrição de um domínio e tenta analisar o texto sobre o que ocorre. A primeira etapa é identificar os atores. Eles são chamados de atores porque *atuam* e executam uma ação. Por exemplo, uma impressora (ator) imprime (ação).

Depois que os atores são identificados, você examina *o que* eles fazem, que é o comportamento. Em seguida, você examina as descrições dos atores e de todos os dados necessários para executar a ação. Os atores são transformados em objetos, as características são codificadas como dados nos objetos e os comportamentos são funções que também são adicionadas ao objeto.



A ideia é que os dados nos objetos podem ser alterados chamando funções nos próprios objetos. Também há a noção de que os objetos *interagem* uns com os outros para chegar a um resultado tangível.

Benefícios da OOP

Então, por que usar a OOP? Por que não usar algum outro paradigma? Para ficar claro, a OOP não é melhor ou pior do que outros paradigmas. Há prós e contras em tudo. A OOP tem alguns benefícios interessantes, que são os seguintes:

- Encapsulamento de dados: o encapsulamento de dados trata-se de ocultar dados do restante do sistema e permitir o acesso apenas a partes dele. O motivo é que os dados armazenam *estado* e esse estado pode ser composto por uma ou mais variáveis. Se essas variáveis precisarem ser alteradas ao mesmo tempo, você precisará protegê-las e permitir o acesso apenas por meio de métodos públicos para que as alterações sejam feitas de maneira previsível. A OOP tem mecanismos como níveis de acesso, em que os dados que estão em um objeto só podem ser acessados pelo próprio objeto ou podem ser tornados públicos.
- Simplicidade: a criação de sistemas grandes é uma tarefa complexa, com muitos problemas para serem resolvidos. A possibilidade de dividir a complexidade em problemas menores, em objetos, significa poder *simplificar* a tarefa geral.
- Facilidade de modificar: quando você depende de objetos e modela seu sistema com eles, é mais fácil rastrear quais partes do sistema precisam de modificação. Por exemplo, talvez seja necessário corrigir um bug ou adicionar um novo recurso.
- Facilidade de manutenção: manter o código, em geral, é difícil e fica mais difícil ao longo do tempo. Ele requer disciplina na forma de uma boa nomenclatura e uma arquitetura clara e consistente, entre outras coisas. O uso de objetos facilita a localização de uma área específica do seu código que precisa de manutenção.
- Capacidade de reutilização: a definição de um objeto pode ser usada muitas vezes em muitas partes do seu sistema ou potencialmente em outros sistemas também. Ao reutilizar o código, você economiza tempo e dinheiro, pois precisa escrever menos código e atingir o seu objetivo mais rapidamente.

Modelar um sistema OOP

Geralmente, o software é escrito para atender a uma necessidade de tornar algo mais rápido, mais eficiente e menos propenso a erros. As pessoas simplesmente não conseguem competir com software quando se trata de acelerar um processo em determinados casos. Usar a OOP é tanto exercer a modelagem quanto escrever o código para implementar a lógica dele. Modelagem trata-se de aprender a identificar os atores, os dados necessários e o tipo de interação que está ocorrendo. Você pode modelar um sistema só lendo uma descrição dele.

Estudo de caso de um sistema de gerenciamento de faturas

Vamos examinar um fluxo manual muito difícil para várias empresas, a saber, o *gerenciamento de faturas*. Muitas empresas recebem faturas que precisam ser pagas no prazo. Atrasar pagamentos gera multas, que se traduzem em dinheiro perdido. Antes que uma fatura possa ser paga, ela precisa ser *processada*. É comum que uma fatura passe por algumas mãos antes que seja registrada em algum lugar e o pagamento seja feito.

O processo geralmente começa com uma fase de classificação inicial em que a fatura é enviada ao departamento apropriado. Em seguida, a fatura é conferida e aprovada por alguém com o nível de autorização adequado. Por fim, ela é paga. Para uma pequena empresa, o proprietário pode realizar todas as etapas. Em uma grande empresa, muitas pessoas e processos podem estar envolvidos, o que torna o gerenciamento de faturas uma atividade complexa.

O que essa descrição tem a ver com a OOP? Se você adotasse o fluxo de trabalho anterior, que geralmente é manual, e o transformasse em software escrito, a primeira coisa que você faria é tentar modelar o sistema. Com o contexto do gerenciamento de faturas, você pode começar a ver atores (objetos), comportamentos e dados lendo uma descrição do domínio do problema.

Se você pensar no domínio descrito como tendo fases, entrada, processamento e saída, poderá começar a preencher coisas como na seguinte tabela:

Fase	**O que**
Entrada	Fatura
Processando	Classificando, Aprovação, Rejeição
Saída	Pagamento

A tabela anterior descreve o que acontece em cada fase. Você conseguiu encontrar os dados, as coisas que acontecem com os dados durante o processamento e o resultado final, que é um pagamento. Neste ponto, você ainda pode resolver o fluxo de trabalho do sistema de gerenciamento de faturas com qualquer paradigma que desejar. Como você o leva daqui para a OOP?

Localizar objetos, dados e comportamento

Você encontra os diferentes artefatos do seu sistema fazendo perguntas como:

- Quem interage com quem?
- Quem faz o quê com quem?

Com essas perguntas em mente, você pode criar instruções. Vamos destacar os diferentes artefatos nessas instruções para que fique claro quais partes são importantes para nosso sistema.

1. O *serviço de e-mail entrega* uma *fatura* para o sistema.
2. A *fatura* é *classificada* por um *código de referência* ou manualmente por um *classificador* para ir para o departamento correto.
3. A *fatura* é *aprovada* ou *rejeitada* por um *aprovador* com base em fatores como, por exemplo, exatidão e tamanho do *valor*.
4. A *fatura* é *paga* por um *processador de pagamentos* usando as *informações de pagamento* fornecidas.

Agora você pode extrair objetos, dados e o comportamento das frases e organizá-los em uma tabela, desta forma:

Fase	**Ator**	**Comportamento**	**Dados**
Entrada	Serviço de email	Entrega	Fatura
Entrada	Sistema	Recebe	Fatura

Processando	Classificadores ou sistema	Classifica ou encaminha	Fatura (código de referência)
Processando	Aprovador	Aprova ou rejeita	Fatura (valor)
Saída	Processador de pagamentos	Paga	Fatura (informações do pagamento)

Muita coisa aconteceu com a descrição inicial do sistema de gerenciamento de faturas. Atores (objetos) foram encontrados. Dados importantes foram identificados e *agrupados* com objetos identificados. O comportamento também foi encontrado, o que deixa mais claro quais atores (objetos) interagem entre si. Como resultado, você conseguiu identificar *quem* faz qual comportamento com *quem*. Você fez uma análise inicial neste ponto, que é um ótimo começo. Mas a pergunta permanece: como você transforma essa análise em código? A resposta a essa pergunta é o que vamos resolver ao longo deste módulo.

Usar classes e variáveis para transferir o modelo OOP para o código

É interessante avaliar a OOP (programação orientada a objeto) criando o jogo pedra, papel e tesoura. Para fazer isso, primeiro você precisa modelá-lo em um formato OOP.

Precisaremos aplicar alguns conceitos básicos de OOP, como classes, objetos e estado. Esta unidade explora as seguintes partes:

- Conceitos de OOP importantes: para analisar nos termos da OOP, você precisa entender alguns conceitos básicos como classes, objetos e estado. Você precisa saber qual é a diferença entre eles e como eles se relacionam uns com os outros.
- Variáveis na OOP: você precisa saber como lidar com variáveis e como adicioná-las aos seus objetos.

O que é um objeto?

O conceito de objetos foi mencionado algumas vezes já como parte da tentativa de *modelar* domínios do problema. Um objeto é um ator. É algo que faz alguma coisa dentro de um sistema. Em decorrência da execução de uma ação, ela altera o estado dentro de si mesma ou em outros objetos.

Vamos imaginar um objeto no mundo real. Você está em um estacionamento. O que você vê? É provável que você veja muitos carros, de diferentes formas, tamanhos e cores. Para descrever um carro, você pode usar propriedades como marca, modelo, cor e tipo de carro. Se você atribui valores a essas propriedades, fica claro rapidamente se você está falando sobre um Ferrari vermelho, Jeep com tração nas quatro rodas, um Mustang amarelo e assim por diante.

Em outra cena, imagem de um baralho em Las Vegas. Você examina dois cartões diferentes, que são dois objetos. Você percebe que eles têm algumas propriedades comuns, a saber, naipes. O naipe dos objetos pode ser paus, copas, ouros ou espadas. Os valores deles podem ser ás, rei, nove e assim por diante.

O que é uma classe?

Uma classe é um tipo de dados que funciona como uma definição de modelo para um determinado tipo de objeto.

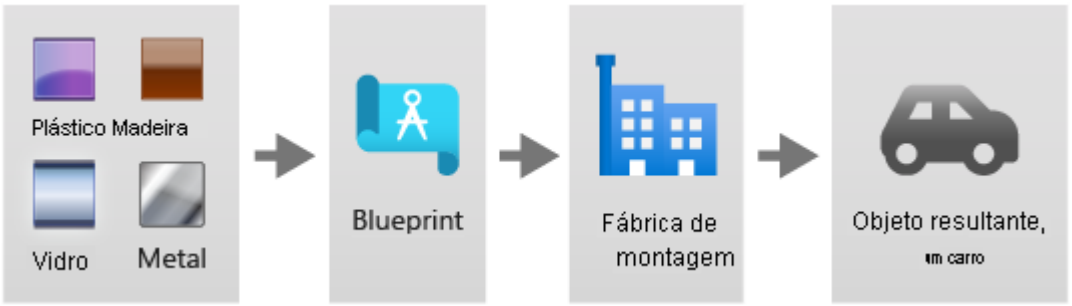
Você aprendeu que um sistema OOP tem objetos, e esses objetos têm propriedades. Há um conceito semelhante a um objeto, que é classe. Então, o que é uma classe? Uma classe é um blueprint de um objeto. Se a classe é o blueprint de um carro, o objeto é o carro que você dirige. A classe é o que você escreve no código. O objeto é o que você obtém quando solicita que o ambiente de runtime execute seu código.

Esta é uma tabela de alguns exemplos de classes e os objetos resultantes:

Classe	**Objeto**
Blueprint de um carro	Honda Accord, Jeep Wrangler
Gato	Garfield, o gato
Descrição de sorvete	Morango, chocolate ou baunilha

A maneira como você cria um objeto com base em uma classe é semelhante à maneira como você criaria um carro com base em um blueprint. Quando você cria um objeto, seu programa solicita ao sistema operacional alguns recursos, como a memória, para poder construir o objeto. Por outro lado, quando um carro é feito com

base em um blueprint, a fábrica solicita recursos como metal, borracha e vidro para poder montar o carro.



Criar uma classe

Uma classe em Python é criada usando a palavra-chave `class` e dando a ela um nome, como neste exemplo:

```
class Car:
```

Criar um objeto com base em uma classe

Quando você cria um objeto com base em uma classe, dizemos que você o *instancia*. O que você faz é pedir ao sistema operacional que, com esse modelo (a classe) e esses valores iniciais, ele forneça memória suficiente e crie um objeto. Essencialmente, instanciar é outra palavra para criar.

Para instanciar um objeto, você adiciona parênteses ao nome da classe. O que você obtém é um objeto que você pode optar por atribuir a uma variável, dessa forma:

```
car = Car()
```

`car` é a variável que contém a instância do objeto. O momento de instanciação, ou criação do objeto, é quando você chama `Car()`.

Variáveis na OOP versus variáveis em programas procedurais

Com base na programação procedural, você está acostumado a ter variáveis para conter as informações e acompanhar o estado. Você pode definir essas variáveis sempre que precisar delas em seu arquivo. A *inicialização* de uma variável típica pode ter esta aparência:

```
pi = 3.14
```

Você também tem variáveis na OOP, embora elas sejam *anexadas* a objetos em vez de serem definidas por conta própria. Você faz referência a variáveis em um objeto como *atributos*. Quando um atributo é anexado a um objeto, ele é usado de uma das duas maneiras:

- Descrever o objeto: um exemplo de uma variável de *descrição* é, por exemplo, a cor de um carro ou o número de manchas em uma girafa.
- Conter o estado: uma variável pode ser usada para descrever o estado de um objeto. Um exemplo de estado é o andar de um elevador ou se ele está em operação ou não.

Adicionar atributos a uma classe

Saber quais os atributos (variáveis) que devem ser adicionados à sua classe faz parte da modelagem. Você aprendeu a criar uma classe no código. Então como adicionar um atributo a ela? Você precisa informar à classe quais atributos ela deve ter no momento da construção, quando um objeto está sendo instanciado. Há uma função especial que está sendo chamada no momento da criação, chamada de *construtor*.

Construtor

Muitas linguagens de programa têm a noção de um construtor, a função especial que só é invocada quando o objeto é criado pela primeira vez. O construtor será chamado apenas uma vez. Nesse método, você cria os atributos que o objeto deve ter. Além disso, você atribui valores iniciais aos atributos criados.

Em Python, o construtor tem o nome `__init()__`. Você também precisa passar uma palavra-chave especial, `self`, como um parâmetro para o construtor. A palavra-chave `self` refere-se à instância do objeto. Qualquer atribuição a essa palavra-chave significa que o atributo termina na instância do objeto. Se você não

adicionar um atributo a `self`, ele será tratado como uma variável temporária que não existirá depois que a execução de `__init()__` for concluída.

Adicionar e inicializar atributos em uma classe

Vejamos um exemplo de configuração de atributos em um construtor:

```
class Elevator:

  def __init__(self, starting_floor):

    self.make = "The elevator company"

    self.floor = starting_floor


# To create the object


elevator = Elevator(1)

print(elevator.make) # "The Elevator company"

print(elevator.floor) # 1
```

O exemplo anterior descreve a classe `Elevator` com duas variáveis, `make` e `floor`. Uma importante vantagem do código é que `__init__()` é chamado implicitamente. Você não chama o método `__init__()` por nome, mas ele é chamado quando o objeto é criado, nesta linha de código:

```
elevator = Elevator(1)
```

Uso incorreto de `self`

Para enfatizar como a palavra-chave `self` funciona, considere o seguinte código no qual dois atributos, `color` e `make`, estão sendo atribuídos no construtor `__init__()`:

```
class Car:

  def __init__():

    self.color = "Red" # ends up on the object

    make = "Mercedes" # becomes a local variable in the constructor


car = Car()

print(car.color) # "Red"

print(car.make) # would result in an error, `make` does not exist on the object
```

Se a finalidade fosse tornar `color` e `make` os atributos da classe `Car`, você precisaria modificar o código. No construtor, verifique se ambos os atributos são atribuídos a `self`, desta forma:

```
self.color = "Red" # ends up on the object

self.make = "Mercedes"
```

Exercício – Modelar e realizar scaffold do seu jogo

Um jogo não é diferente do *sistema de faturas* mencionado em uma unidade anterior. Ele ainda tem as mesmas partes no mundo da OOP (programação orientada a objeto). Essas partes são objetos, dados e comportamento. Assim como fizemos com o sistema de faturas, você pode *modelar* um jogo como pedra, papel e tesoura descrevendo primeiro o domínio. Em seguida, tente listar o que é o quê. A modelagem do jogo é o que você vai fazer em seguida. Você também escreverá um código que poderá ser criado posteriormente.

Analisar pedra, papel e tesoura para a modelagem OOP

Este exercício é um exercício de modelagem. Você receberá uma descrição do domínio do problema. Em seguida, deverá retirar as palavras-chave importantes do texto e organizá-las em uma tabela.

Descrição do problema

Pedra, papel e tesoura é um jogo com dois participantes. O jogo tem rodadas. Em cada rodada, um participante escolhe um símbolo de pedra, papel ou tesoura, e o outro participante faz o mesmo. O vencedor da rodada é determinado pela comparação dos símbolos escolhidos. As regras do jogo estabelecem que pedra ganha de tesoura, tesoura vence (corta) papel e papel vence (embrulha) pedra. O vencedor da rodada recebe um ponto. O jogo continua pelo tanto de rodadas que os participantes quiserem. O vencedor é o participante com o maior número de pontos.

Modelar o jogo

1. Copie e cole o texto anterior em um documento. Realce as palavras-chave importantes colocando-as em negrito ou itálico.

2. Pedra, papel e tesoura é um *jogo* com dois participantes. O jogo tem *rodadas*. Em cada rodada, um *participante* escolhe um *símbolo* de *pedra*, *papel* ou *tesoura*, e o outro *participante* faz o mesmo. O *vencedor* da rodada é determinado pela *comparação* dos símbolos escolhidos. As *regras* do jogo estabelecem que pedra ganha de tesoura, tesoura vence (corta) papel e papel vence (embrulha) pedra. O vencedor da rodada recebe um *ponto*. O jogo continua pelo tanto de rodadas que os participantes quiserem. O vencedor é o participante com o maior número de pontos.

3. Em seguida, crie uma tabela com as colunas `Phase`, `Actor`, `Behavior` e `Data`. Organize as palavras realçadas onde você acha que elas devem ser colocadas.

Esta é a aparência da tabela resultante:

Fase	**Ator**	**Comportamento**	**Dados**
Entrada	Participante	Escolhe um símbolo	Símbolo salvo como *escolha* em Participant(choice)
Processando	GameRound	Compara a escolha tendo em mente as regras do jogo	*Resultado* inspecionado
Processando	GameRound	Recebe pontos com base no valor do resultado	*Pontos* adicionados ao Participant(point) vencedor
Processando	Game	Verificar resposta continuar	A resposta é true, continuar; caso contrário, sair

Saída	Game	Nova crédito de rodada do jogo ou fim do jogo

Criar classes e estado

A tabela anterior informa a história de como o jogo progride por diferentes fases. Concentrando-se nas duas colunas `Behavior` e `Data`, você pode *fazer scaffold* de um código inicial que ajudará a dar suporte à criação do jogo.

1. Crie um arquivo `rock-paper-scissor.py` usando o terminal e abra o editor:

touch rock-paper-scissor.py

code .

Dê a ele o seguinte conteúdo e salve o arquivo (CTRL + S ou Command + S no macOS):

class Participant:

class GameRound:

class Game:

1. Você tem as classes necessárias criadas para seu jogo. Em seguida, você precisa pensar em quais dados você tem e em qual classe colocá-los.
2. Continue trabalhando com o mesmo arquivo e atualize o código para que:class Participant:

def __init__(self):

self.points = 0

```
        self.choice = ""

class GameRound:

class Game:

    def __init__(self):

        self.endGame = False

        self.participant = Participant()

        self.secondParticipant = Participant()
```

A classe `Participant` tenha recebido os atributos `points` e `choice` conforme indicado pela primeira e terceira linhas da tabela.

`Game` tenha recebido os campos `endGame` devido à quarta linha. Além disso, a classe `Game` tem dois participantes, `participant` e `secondParticipant`. Havia duas funções que uma variável em um objeto poderia ter. As funções podem ser um estado, como o andar de um elevador, ou um atributo descritivo. Os atributos `points` e `choice` são variáveis de estado neste contexto. Os participantes na classe `Game` são atributos descritivos porque um jogo *tem* participantes.

Parabéns! Você adicionou classes ao seu jogo e criou dados – atributos que você atribuiu às classes criadas. Neste ponto, você tem um bom código inicial. Ele não faz muita coisa ainda porque precisa de comportamento. Você adicionará o comportamento na próxima unidade do exercício.

Adicionar comportamento com métodos

A meta final de um sistema é produzir uma saída útil. Para chegar lá, você precisa processar a entrada. Durante o *processamento*, talvez você precise da ajuda de vários métodos e dados para fazer isso. Na OOP (programação orientada a objeto), seus métodos e dados são colocados em objetos. Para processar a entrada e produzir um resultado, na OOP, você precisa de métodos.

Métodos na OOP

Independentemente do paradigma usado, os métodos podem executar uma ação. Essa ação pode ser uma computação que só se baseia em entradas ou pode alterar o valor de uma variável.

Métodos em objetos, na OOP, são de dois tipos:

- Métodos externos, que outros objetos podem invocar.
- Métodos internos, que não podem ser acessados por outros objetos. Além disso, esses métodos ajudam a executar uma tarefa iniciada por uma invocação para um método externo.

Independentemente do tipo de método, eles podem alterar o valor do atributo de um objeto, em outras palavras, o *estado* dele.

A noção de estado e *quem* e *o que* pode alterá-lo, é um assunto importante. É uma parte importante da criação de classes e objetos. Essas perguntas nos levam à nossa próxima seção, *encapsulamento*.

Encapsulamento: proteger seus dados

A ideia geral do encapsulamento é que os dados em um objeto são *internos*, algo que só diz respeito ao objeto. Os dados são necessários para o objeto e os métodos para fazerem o trabalho deles, que é executar uma tarefa. Quando você diz que os dados são *internos*, está dizendo que eles devem ser protegidos contra manipulação externa ou, em vez disso, *manipulação* externa não controlada. A pergunta é "por quê"?

Por que você precisa disto

Vamos explicar o motivo pelo qual os dados não devem ser diretamente tocados por outro objeto. Estes são alguns exemplos:

- Você não precisa conhecer os elementos internos. Ao dirigir um carro, você pisa em um pedal para controlar a embreagem ou para acelerar ou frear. Como você está operando seu carro em um nível mais alto, não se importa com o que acontece na parte de baixo, como o carro executa a ação. É a mesma coisa com o código. Na maioria das vezes, você não precisa saber como um objeto faz alguma coisa, desde que haja um método que você possa invocar que faz o que você deseja.
- Você não deve conhecer o mecanismo interno. Em vez de fazer um pedal interagir com o carro, imagine que você tenha uma chave de fenda ou um kit de soldagem para tentar acelerar. Parece assustador, certo? É porque é assustador mesmo. Ou digamos que você tenha um exemplo mais concreto, uma classe square, com o seguinte código:

```
class Square:
  def __init__(self):
    self.height = 2
    self.width = 2
  def set_side(self, new_side):
    self.height = new_side
    self.width = new_side

square = Square()
square.height = 3 # not a square anymore
```

- No exemplo de quadrado, você tem a noção do que é um quadrado definindo a variável `height`. Da forma como o quadrado é codificado, ele precisa invocar o método `set_side()` para que o quadrado funcione corretamente. Deixar o objeto cuidar dos próprios dados é considerado mais seguro. Em quase todas as instâncias, você deve optar por interagir por meio de um método versus definir os dados explicitamente.

Níveis de acesso

Como você pode proteger sua classe e seu objeto contra manipulação indesejada de dados? A resposta é: com *níveis de acesso*. Você pode ocultar dados do mundo exterior e de outros objetos marcando dados e funções com palavras-chave específicas. Essas palavras-chave são conhecidas como modificadores de acesso.

A maneira como o Python realiza a ocultação de dados é adicionando prefixos a nomes de atributos. Um sublinhado à esquerda, _, é uma mensagem para o mundo exterior de que esses dados provavelmente não devem ser tocados. Quando você modifica a classe square, você acaba tendo este código:

```
class Square:
    def __init__(self):
      self._height = 2
      self._width = 2
    def set_side(new_side):
      self._height = new_side
      self._width = new_side

  square = Square()
  square._height = 3 # not a square anymore
```

Um sublinhado à esquerda ainda permite que os dados sejam modificados, aos quais o Python se refere como *protegidos*. Podemos fazer isso melhor? Sim, com dois sublinhados à esquerda, __, que é conhecido como *privado*. Sua classe square agora se parece com este código:

```
class Square:
    def __init__(self):
      self.__height = 2
      self.__width = 2
    def set_side(new_side):
      self.__height = new_side
      self.__width = new_side

  square = Square()
  square.__height = 3 # raises AttributeError
```

Ótimo, então estamos seguros. Protegemos nossos dados? Bem, não totalmente. O Python apenas altera o nome dos dados subjacentes. Inserindo este código, você ainda pode alterar o valor dele:

```
square = Square()
square._Square__height = 3 # is allowed
```

Muitas outras linguagens que implementam a proteção de dados resolvem esse problema de maneira diferente. O Python é exclusivo, pois a proteção de dados é mais semelhante a níveis de sugestão em vez de ser estritamente implementada.

O que são getters e setters?

Dissemos até aqui que os dados, em geral, não devem ser tocados por alguém de fora. Os dados são a preocupação do objeto. Assim como acontece com todas as regras e recomendações fortes, há exceções. Às vezes, você precisa alterar os dados. Alterar é mais simples do que precisar adicionar uma quantidade significativa de código.

Os getters e setters, que também são conhecidos como *acessadores* e *modificadores*, são métodos dedicados à leitura ou à alteração de seus dados. Os getters desempenham o papel de tornar seus dados internos legíveis para o mundo exterior, o que não parece tão ruim, não é? Setters são métodos que podem alterar seus dados diretamente. A ideia é que um setter atue como uma proteção para que não seja possível definir um valor *indevido*. Vamos abrir nossa classe square novamente e ver getters e setters em ação:

```
class Square:
    def __init__(self):
      self.__height = 2
      self.__width = 2
    def set_side(self, new_side):
      self.__height = new_side
      self.__width = new_side
    def get_height(self):
      return self.__height
    def set_height(self, h):
      if h >= 0:
        self.__height = h
      else:
        raise Exception("value needs to be 0 or larger")


square = Square()
```

```
square.__height = 3 # raises AttributeError
```

O método `set_height()` impede que você defina o valor como algo negativo. Se você fizer isso, ele vai gerar uma exceção.

Usar decoradores para getters e setters

Os decoradores são uma entidade importante em Python. Eles fazem parte de uma entidade maior chamada *metaprogramação*. Os decoradores são funções que usam sua função como uma entrada. A ideia é codificar a funcionalidade reutilizável como *funções decoradoras* e, em seguida, *decorar* outras funções com ela. A finalidade é dar à sua função um recurso que ela não tinha antes. Um decorador pode, por exemplo, adicionar campos ao seu objeto, medir o tempo necessário para invocar uma função e fazer muito mais.

No contexto da OOP e getters e setters, um decorador específico `@property` pode ajudar você a remover um código clichê quando adiciona getters e setters. O decorador `@property` faz o seguinte para você:

- Cria um campo de suporte: quando você decora uma função com o decorador `@property`, ele cria um campo de suporte específico. Você poderá substituir esse comportamento se desejar, mas é bom ter um comportamento padrão.
- Identifica um setter: um método setter pode alterar o campo de suporte.
- Identifica um getter: essa função deve retornar o campo de suporte.
- Identifica uma função de exclusão: essa função pode excluir o campo.

Vamos ver esse decorador em ação:

```
class Square:

  def __init__(self, w, h):

    self.height = h

    self.__width = w


  def set_side(self, new_side):

    self.__height = new_side
```

```
    self.__width = new_side


  @property
  def height(self):
    return self.__height


  @height.setter
  def height(self, new_value):
    if new_value >= 0:
      self.__height = new_value
    else:
      raise Exception("Value must be larger than 0")
```

No código anterior, a função `height()` é decorada pelo decorador `@property`. Essa ação de *decoração* cria o campo privado `__height`. O campo `__height` não está definido no construtor `__init__()` porque o decorador já faz isso. Há também outra decoração acontecendo, ou seja, `@height.setter`. Essa decoração aponta para um método `height()` semelhante ao *setter*. O novo método height usa um outro parâmetro `value` como o segundo parâmetro.

A capacidade de manipular a altura separada da largura ainda causará um problema. Você precisará entender o que a classe faz antes de você considerar a possibilidade de permitir getters e setters, pois você está introduzindo o risco.

Exercício – adicionar comportamento ao seu jogo

Agora que você adicionou classes e dados ao seu jogo de pedra, papel e tesoura, é hora de adicionar a última parte, que é o comportamento. Você adicionará o comportamento na forma de métodos. Você adicionará métodos às suas classes e o resultado final será um jogo de pedra, papel e tesoura funcional.

Implementar comportamento

Você já aprendeu que, para criar um programa em estilo OOP (programação orientada a objeto), primeiro você modela e depois codifica. A modelagem produziu a saída de uma tabela que representava por quais objetos, dados e comportamento o seu programa parece ser composto. Esta é a mesma tabela novamente.

Fase	**Ator**	**Comportamento**	**Dados**
Entrada	Participante	Escolhe um símbolo	Símbolo salvo como *escolha* em Participant(choice)
Processando	GameRound	Compara a escolha tendo em mente as regras do jogo	*Resultado* inspecionado
Processando	GameRound	Recebe pontos com base no valor do resultado	*Pontos* adicionados ao Participant(point) vencedor
Processando	Game	Verificar resposta continuar	A resposta é true, continuar; caso contrário, sair
Saída	Game	Nova crédito de rodada do jogo ou fim do jogo	

Desta vez, você se concentrará na coluna `Behavior` e preencherá a coluna com métodos que serão adicionados às suas classes. Além disso, você adicionará código a esses métodos para que eles funcionem da maneira que deveriam.

Este é o código até o momento. Vamos estendê-lo nas seguintes etapas:

1. Você deve ter um arquivo `rock-paper-scissor.py`.

Caso contrário, execute as seguintes etapas:

2. Crie o arquivo touch rock-paper-scissor.py
3. code .
4. Crie o arquivo `rock-paper-scissor` e abra o editor:

e adicione o código abaixo:

```
class Participant:
  def __init__(self):
     self.points = 0
     self.choice = ""

class GameRound:

class Game:
   def __init__(self):
   self.endGame = False
   self.participant = Participant()
   self.secondParticipant = Participant()
```

Iniciar um jogo

A primeira parte do jogo envolve configurá-lo, o que significa criar uma instância do próprio jogo e levar o jogo para um ponto em que ele fique aguardando a ação dos participantes.

1. Substitua o conteúdo de `rock-paper-scissors.py` por este código:

```
class Participant:
  def __init__(self, name):
    self.name = name
    self.points = 0
    self.choice = ""
  def choose(self):
    self.choice = input("{name}, select rock, paper or scissor: ".format(name=
self.name))
    print("{name} selects {choice}".format(name=self.name, choice = self.choice))

class GameRound:
  def __init__(self, p1, p2):
    p1.choose()
    p2.choose()
  def compareChoices(self):
    print("implement")
  def awardPoints(self):
    print("implement")

class Game:
  def __init__(self):
    self.endGame = False
    self.participant = Participant("Spock")
    self.secondParticipant = Participant("Kirk")
  def start(self):
    game_round = GameRound(self.participant, self.secondParticipant)

  def checkEndCondition(self):
    print("implement")
  def determineWinner(self):
    print("implement")

game = Game()
game.start()
```

Você adicionou os métodos de sua tabela a cada objeto. As alterações feitas podem ser expressas por uma tabela para que seja mais fácil ver qual comportamento levou ao método que está sendo adicionado.

Comportamento	**Método**	**Ator**
Escolhe um símbolo	choose()	Participante
Compara opções	compareChoices()	GameRound
Recebe pontos	awardPoints()	GameRound
Verificar resposta continuar	checkEndCondition()	Game
Crédito final do jogo	determineWinner()	Game

A maior parte do comportamento na tabela anterior corresponde a métodos com nomes semelhantes. A exceção é *Crédito final do jogo*, que se torna `determineWinner()`. O motivo é que, como parte do final de um jogo, convém verificar quem ganhou e imprimir essas informações.

Fica a seu critério dar outro nome a esse método.

Execute o código invocando `python3`:

```
python3 rock-paper-scissors.py
```

O programa produz uma saída como esta:

```
Spock, select rock, paper or scissor:
```

Essa saída significa que o programa está aguardando sua ação.

Verifique se o programa funciona selecionando `rock` e, em seguida, selecionando Enter. Selecione `paper` e Enter novamente:
A saída será semelhante a este texto:

```
Spock, select rock, paper or scissor: rock
Spock selects rock
Kirk, select rock, paper or scissor: paper
Kirk selects paper
```

Implementar regras

Na descrição do problema, você lê que algumas escolhas vencem outras. Por exemplo, pedra vence tesoura, tesoura vence papel e assim por diante. É tentador escrever um código semelhante ao seguinte:

```
if choice1 == "rock" and choice2 == "scissor":
  return 1
elif choice1 == "paper" and choice2 == "scissor":
  return -1
else:
  # something else

# and so on
```

Resulta em uma quantidade significativa de código escrito e torna-se um pouco difícil. E se o jogo precisar expandir o conjunto de regras, o que poderá tornar o código ainda mais difícil de ser mantido?

Felizmente, há uma forma melhor. Uma abordagem melhor é pensar nas regras como uma matriz. A ideia de usar uma matriz é expressar qual combinação ganha de outras combinações. Uma jogada vencedora recebe um `1`, um empate recebe um `0` e uma jogada perdedora recebe um `-1`. A seguinte matriz destina-se a pedra, papel e tesoura:

Escolha	Pedra	Papel	Tesoura
Pedra	0	-1	1
Papel	1	0	-1
Tesoura	-1	1	0

Você pode implementar a tabela anterior em Python usando uma matriz multidimensional, desta forma:

```
rules = [
 [0, -1, 1],
 [1, 0, -1],
 [-1, 1, 0]
]


rules[0][1] # Rock vs Paper = -1, Paper wins over Rock
```

Localize a classe `Participant` e o método `toNumericalChoice()`:

```
def toNumericalChoice(self):
 switcher = {
   "rock": 0,
   "paper": 1,
   "scissor": 2
 }
 return switcher[self.choice]
```

o método anterior converterá sua entrada de cadeia de caracteres de linha de comando em um inteiro. Isso facilitará a determinação de quem ganhou uma rodada.

Localize a classe `GameRound` e adicione o método `compareChoices()`:

```
def compareChoices(self, p1, p2):
  return self.rules[p1.toNumericalChoice()][p2.toNumericalChoice()]
```

o código permitirá que você compare duas escolhas e determine o vencedor.
Na mesma classe, adicione o método `getResultAsString()`:

```
        def getResultAsString(self, result):
          res = {
            0: "draw",
            1: "win",
            -1: "loss"
          }
          return res[result]
```

esse método ajudará a determinar o resultado e imprimirá um texto fácil de entender na tela.
Ainda na mesma classe, substitua o conteúdo de `__init__()` por este código:

```
        def __init__(self, p1, p2):
          self.rules = [
            [0, -1, 1],
            [1, 0, -1],
            [-1, 1, 0]
          ]

          p1.choose()
          p2.choose()
          result = self.compareChoices(p1,p2)
          print("Round resulted in a {result}".format(result =
        self.getResultAsString(result) ))
```

O código anterior introduz o campo `rules`, que contém uma implementação das regras para pedra, papel e tesoura. Além disso, a chamada a `self.compareChoices()` *compara* duas escolhas feitas. Por fim, há uma linha que imprime resultados amigáveis para leitores na tela.

Pontuar o jogo

A pontuação do jogo trata-se de atribuir pontos ao jogador correto após o término do jogo. O jogador vencedor recebe um ponto, um empate ou nenhum ponto caso perca.

1. Localize a classe `Participant` e adicione o método `incrementPoint()`:

```
def incrementPoint(self):
    self.points += 1
```

2 . Localize a classe `GameRound` e adicione o seguinte código ao final do método `__init()__`:

```
if result > 0:
  p1.incrementPoint()
elif result < 0:
  p2.incrementPoint()
```

Adicionar uma consulta de continuação

Uma consulta de continuação é perguntar, no final da rodada do jogo, se o jogador quer continuar. Se o usuário opta por não continuar, é interessante gerar os resultados atuais e o vencedor, se houver.

1. Localize a classe `Game` e implemente o método `determineWinner()`:

```
def determineWinner(self):
  resultString = "It's a Draw"
  if self.participant.points > self.secondParticipant.points:
    resultString = "Winner is {name}".format(name=self.participant.name)
  elif self.participant.points < self.secondParticipant.points:
    resultString = "Winner is {name}".format(name=self.secondParticipant.name)
  print(resultString)
```

Na mesma classe, implemente o método `checkEndCondition()`:

```
def checkEndCondition(self):
  answer = input("Continue game y/n")
  if answer == 'y':
    GameRound(self.participant, self.secondParticipant)
    self.checkEndCondition()
  else:
    print("Game ended, {p1name} has {p1points}, and {p2name} has
{p2points}".format(p1name = self.participant.name, p1points= self.participant.points,
p2name=self.secondParticipant.name, p2points=self.secondParticipant.points))
    self.determineWinner()
    self.endGame = True
```

Na mesma classe, substitua o código do método `start()` por este código:

```
def start(self):
  while not self.endGame:
    GameRound(self.participant, self.secondParticipant)
    self.checkEndCondition()
```

Execute o comando `python3 rock-paper-scissors.py` para testar o seu programa:

```
python3 rock-paper-scissors.py
```

Selecione `rock` e `paper` como entradas e insira `n` quando for solicitado a continuar.

A saída é semelhante ao seguinte texto:

```
Spock, select rock, paper or scissor: rock
Spock selects rock
Kirk, select rock, paper or scissor: paper
Kirk selects paper
Round resulted in a loss
Continue game y/n: n
Game ended, Spock has 0, and Kirk has 1
Winner is Kirk
```

Exercício – estender a implementação do jogo com regras adicionadas

Sua empresa está satisfeita com a implementação da OOP (programação orientada a objeto) do jogo pedra, papel e tesoura. Sendo assim, querem que você mude o jogo para pedra, papel, tesoura, lagarto, Spock porque é o que as crianças legais estão jogando hoje em dia.

Adicionar as escolhas lagarto e Spock

Pode parecer que adicionar mais duas escolhas como lagarto e Spock é muita coisa a ser mudada. Graças à maneira como você implementou as regras, as alterações necessárias são secundárias.

No entanto, quais são as regras para lagarto e Spock?

A tesoura decapita o lagarto, tesoura corta papel, papel embrulha a pedra, a pedra esmaga o lagarto, o lagarto envenena Spock, Spock destrói a tesoura, a tesoura decapita o lagarto, o lagarto come o papel, o papel refuta Spock, Spock vaporiza a pedra e, como sempre, a pedra esmaga a tesoura.

A descrição anterior pode ser convertida nesta tabela de regras atualizada:

Escolha	**Pedra**	**Papel**	**Tesoura**	**Lagarto**	**Spock**
Pedra	0	-1	1	1	-1
Papel	1	0	-1	-1	1
Tesoura	-1	1	0	1	-1
Lagarto	-1	1	-1	0	1
Spock	1	-1	1	-1	0

Localize a classe `Participant` e atualize o método `toNumericalChoice()` para ter a seguinte aparência:

```
def toNumericalChoice(self):
  switcher = {
    "rock": 0,
    "paper": 1,
    "scissor": 2,
    "lizard": 3,
    "spock": 4
  }
  return switcher[self.choice]
```

Localize a classe `GameRound`. No método `__init__()`, altere a variável `self.rules` para este código:

```
self.rules = [
  [0, -1, 1, 1, -1],
  [1, 0, -1, -1, 1],
  [-1, 1, 0, 1, -1],
  [-1, 1, -1, 0, 1],
  [1, -1, 1, -1, 0]
]
```

Execute `python3 rock-paper-scissor.py` para experimentar suas alterações:

```
python3 rock-paper-scissor.py
```

Selecione `spock` e `paper` para ver que as regras funcionam corretamente. Seu resultado deve ser semelhante ao seguinte exemplo:

```
Spock, select rock, paper, scissor, lizard or spock: spock
Spock selects spock
Kirk, select rock, paper, scissor, lizard or spock: paper
Kirk selects paper
Round resulted in a loss
Continue game y/n: n
Game ended, Spock has 0, and Kirk has 1
Winner is Kirk
```

Parabéns! Você adicionou duas escolhas, *lagarto* e *Spock*, ao jogo com mínimo esforço para seu código.

Verificação de conhecimentos

Escolha a melhor resposta para cada pergunta. Em seguida, selecione Verificar suas respostas.

1. Qual é a diferença entre uma classe e um objeto?
   - ☐ São a mesma coisa.
   - ☐ Uma classe é um blueprint. Um objeto é a instância concreta que você cria com base no blueprint.

- ☐ Uma classe tem dados e um objeto não.
- ☐ Uma classe tem métodos e um objeto não.

2. Como o Python implementa acessadores?
   - ☐ Ele usa prefixos.
   - ☐ Um sublinhado, _, no início do nome indica que esta variável é *protegida*.
   - ☐ Dois sublinhados, __, tornam a variável *privada* e gerarão uma exceção, se atribuída.
   - ☐ Ele usa as palavras-chave *privado* e *público*.
   - ☐ Tudo em uma classe Python é oculto por padrão e deve ser exposto por meio da adição de métodos que retornam os valores de variáveis.
   - ☐ Adicione o decorador @public ou @private para tornar algo público ou privado.

3. Escolha a melhor explicação para a palavra-chave `self`.

   - ☐ É uma palavra-chave que se refere à instância do objeto.
   - ☐ Você também pode usar a palavra-chave `this`.
   - ☐ A palavra-chave `self` refere-se à classe subjacente do objeto.
   - ☐ O único momento em que a palavra-chave `self` é usada é para fazer referência aos membros do objeto e ela é passada como um parâmetro.
   - ☐ A palavra-chave `self` refere-se à instância do objeto. Variáveis que fazem parte da instância do objeto precisam ser atribuídas a `self`. A palavra-chave `self` também é passada para cada método de uma classe que precisa acessar a instância do objeto.