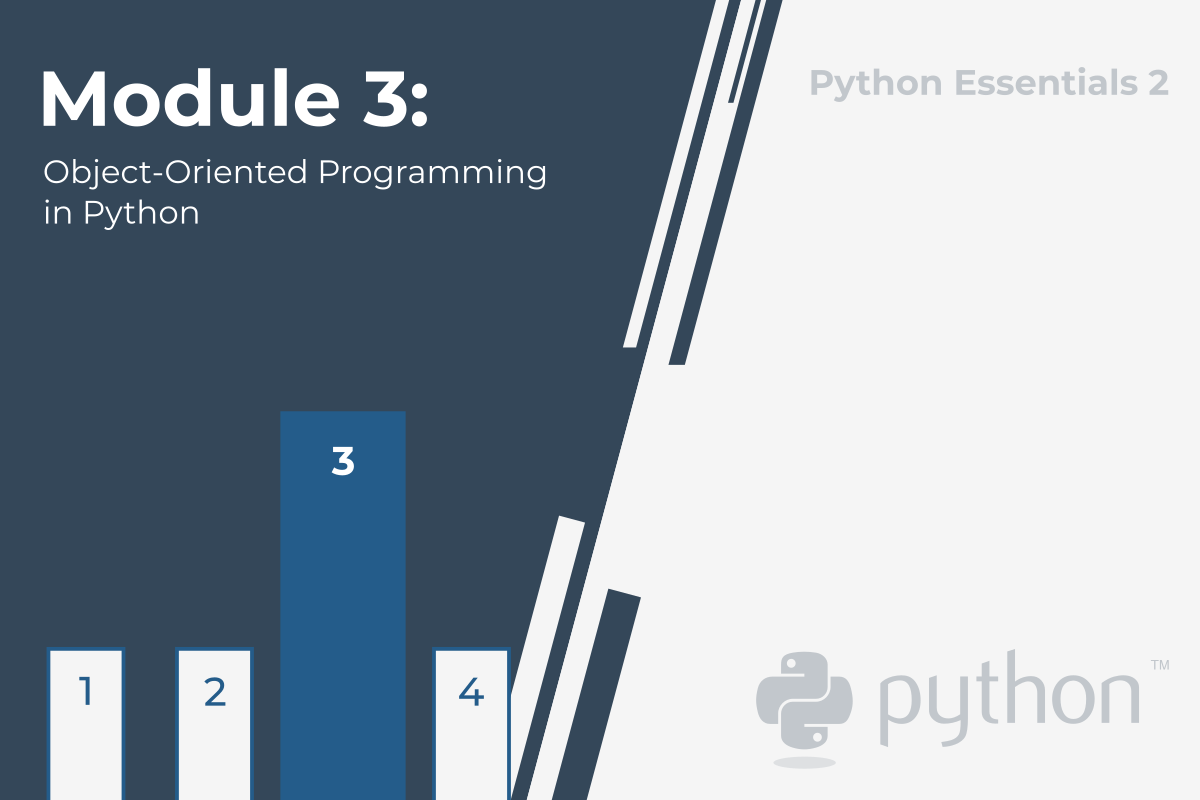
**Python Essentials 2:  
Módulo 3**

**Programação orientada a objetos**

Neste módulo, aprenderá sobre:

* Conceitos básicos de programação orientada a objetos (OOP)
* As diferenças entre as abordagens processual e de objecto (motivações e ganhos)
* Classes, objetos, propriedades e métodos;
* Como projetar classes reutilizáveis e criar objetos;
* Herança e polimorfismo;
* Exceções como objetos.



# Os conceitos básicos da abordagem orientada a objetos

Vamos dar um passo fora da programação informática e dos computadores em geral, e discutir questões de programação de objetos.

Quase todos os programas e técnicas que utilizou até agora enquadram-se no estilo processual de programação. É certo que fez uso de alguns objetos incorporados, mas ao referirmo-nos a eles, acabámos por mencionar o mínimo absoluto.

O estilo processual de programação foi a abordagem dominante no desenvolvimento de software durante décadas do IT, e ainda hoje é utilizado. Além disso, não vai desaparecer no futuro, uma vez que funciona muito bem para tipos específicos de projetos (geralmente, não muito complexos e não grandes, mas há muitas exceções a essa regra).

A abordagem ao objeto é bastante jovem (muito mais jovem do que a abordagem processual) e é particularmente útil quando aplicada a projetos grandes e complexos, realizados por grandes equipas constituídas por vários programadores.

Este tipo de compreensão da estrutura de um projeto facilita muitas tarefas importantes, por exemplo, a divisão do projeto em partes pequenas e independentes, e o desenvolvimento independente de diferentes elementos do projeto.

**O Python é uma ferramenta universal tanto para a programação de objetos como de procedimentos**. Pode ser utilizado com sucesso em ambas as esferas.

Além disso, pode criar muitas aplicações úteis, mesmo que não saiba nada sobre classes e objectos, mas tem de ter em mente que alguns dos problemas (por exemplo, manipulação da interface gráfica do utilizador) podem exigir uma abordagem estrita a objectos.

Felizmente, a programação de objetos é relativamente simples.



# Abordagem processual vs. orientada a objetos

Na **abordagem processual**, é possível distinguir dois mundos diferentes e completamente separados: **o mundo dos dados, e o mundo do código**. O mundo dos dados é povoado por variáveis de diferentes tipos, enquanto o mundo dos códigos é habitado por códigos agrupados em módulos e funções.

As funções são capazes de usar dados, mas não vice-versa. Além disso, as funções são capazes de abusar de dados, ou seja, de utilizar o valor de forma não autorizada (por exemplo, quando a função seno obtém um saldo de conta bancária como parâmetro).

Dissemos no passado que os dados não podem usar funções. Mas isto é inteiramente verdade? Existem alguns tipos especiais de dados que podem usar funções?

Sim, existem - os chamados métodos. Estes são funções que são invocadas de dentro dos dados, não ao lado deles. Se conseguir ver esta distinção, deu o primeiro passo na programação de objetos.

A **abordagem ao objeto** sugere uma maneira completamente diferente de pensar. Os dados e o código estão juntos no mesmo mundo, divididos em classes.

Cada **classe é como uma receita que pode ser utilizada quando se pretende criar um objeto útil** (é daí que vem o nome da abordagem). Pode produzir tantos objetos quantos forem necessários para resolver o seu problema.

Cada objeto tem um conjunto de características (são chamadas propriedades ou atributos - usaremos ambas as palavras sinonimamente) e é capaz de realizar um conjunto de atividades (que são chamadas métodos).

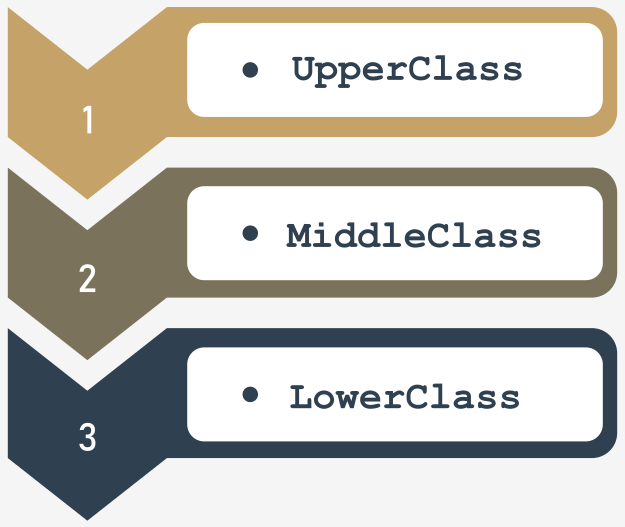
As receitas podem ser modificadas se forem inadequadas para fins específicos e, com efeito, podem ser criadas novas classes. Estas novas classes herdam propriedades e métodos das originais, e normalmente acrescentam algumas novas, criando ferramentas novas e mais específicas.

**Os objectos são encarnações** de ideias expressas em classes, como um cheesecake no prato é uma encarnação da ideia expressa numa receita impressa num velho livro de receitas.

Os objetos interagem uns com os outros, trocando dados ou ativando os seus métodos. Uma classe devidamente construída (e, portanto, os seus objetos) são capazes de proteger os dados sensíveis e de escondê-los de modificações não autorizadas.

Não existe uma fronteira clara entre dados e código: eles vivem como um em objetos.

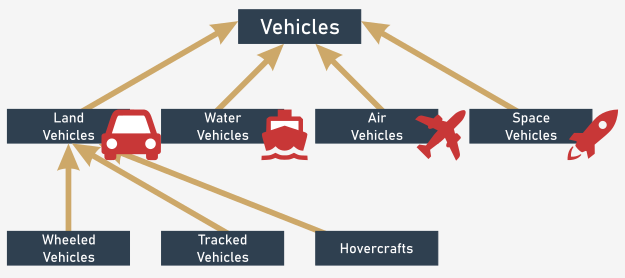
Todos estes conceitos não são tão abstratos como se pode suspeitar no início. Pelo contrário, todos eles são retirados de experiências da vida real, e por isso são extremamente úteis na programação de computadores: não criam vida artificial - **refletem factos, relações, e circunstâncias reais**.



**Hierarquias de classe**

A palavra *classe* tem muitos significados, mas nem todos eles são compatíveis com as ideias que queremos discutir aqui. A *classe* com a qual estamos preocupados é como uma *categoria*, como um resultado de semelhanças definidas com precisão.

Vamos tentar apontar algumas classes que são bons exemplos deste conceito.



Vamos olhar por um momento para os veículos. Todos os veículos existentes (e aqueles que ainda não existem) estão **relacionados por uma única característica importante**: a capacidade de se moverem. Também pode argumentar que um cão se move; um cão é um veículo? Não, não é. Temos de melhorar a definição, ou seja, enriquecê-la com outros critérios, distinguir veículos de outros seres e criar uma ligação mais forte. Tomemos em consideração as seguintes circunstâncias: os veículos são entidades artificialmente criadas utilizadas para o transporte, movidas por forças da natureza e dirigidas (conduzidas) por seres humanos.

Com base nesta definição, um cão não é um veículo.

A classe *veículos* é muito ampla. Demasiado ampla. Temos então de definir algumas **classes mais especializadas**. As classes especializadas são as **subclasses**. A classe *veículos* será uma **superclasse** para todas elas.

Nota: **a hierarquia cresce de cima para baixo, como as raízes das árvores, não como os galhos**. A classe mais geral, e mais ampla, está sempre no topo (a superclasse), enquanto as suas descendentes estão localizados abaixo (as subclasses).

A esta altura, já poderá provavelmente indicar algumas subclasses potenciais para a superclasse *Veículos*. Existem muitas classificações possíveis. Escolhemos subclasses com base no ambiente e dizemos que existem (pelo menos) quatro subclasses:

* land vehicles (veículos terrestres);
* water vehicles (veículos aquáticos);
* air vehicles (veículos aéreos);
* space vehicles (veículos espaciais).

Neste exemplo, discutiremos apenas a primeira subclasse - veículos terrestres. Se desejar, pode continuar com as classes restantes.

Os veículos terrestres podem ser ainda divididos, dependendo do método com o qual tocam o solo. Então, podemos enumerar:

* wheeled vehicles (veículos com rodas);
* tracked vehicles (Veículos com lagartas);
* hovercrafts.

A hierarquia que criámos é ilustrada pela figura.

Observe a direção das setas - elas apontam sempre para a superclasse. A classe de nível superior é uma exceção - não tem a sua própria superclasse.

Outro exemplo é a hierarquia do reino taxonômico dos animais.

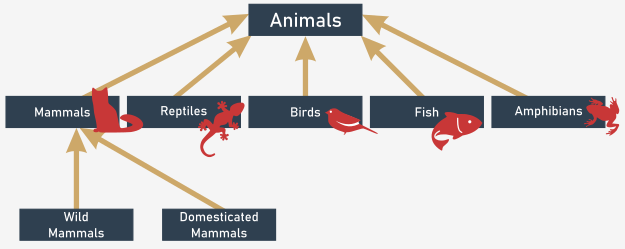
Podemos dizer que todos os *animais* (a nossa classe de nível superior) podem ser divididos em cinco subclasses:

* mamíferos;
* répteis;
* aves;
* peixes;
* anfíbios.

Vamos tomar o primeiro para uma análise mais aprofundada.

Identificamos as seguintes subclasses:

* mamíferos selvagens;
* mamíferos domesticados.



Tente alargar a hierarquia da forma que quiser, e encontre o lugar certo para os seres humanos.

# O que é um objeto?

Uma classe (entre outras definições) é um **conjunto de objetos**. Um objeto é **um ser pertencente a uma classe**.

Um objeto é **uma encarnação dos requisitos, características e qualidades atribuídas a uma classe específica**. Isto pode parecer simples, mas note-se as seguintes circunstâncias importantes. Classes formam uma hierarquia.

Isto pode significar que um objeto pertencente a uma classe específica pertence a todas as superclasses ao mesmo tempo. Pode também significar que qualquer objeto pertencente a uma superclasse não pode pertencer a nenhuma das suas subclasses.

Por exemplo: qualquer carro pessoal é um objeto pertencente à classe dos *veículos de rodas*. Significa também que o mesmo carro pertence a todas as superclasses da sua classe de origem; portanto, é também um membro da classe de *veículos*.

O seu cão (ou o seu gato) é um objeto incluído na classe dos *mamíferos domesticados*, o que significa explicitamente que também está incluído na classe dos *animais*.

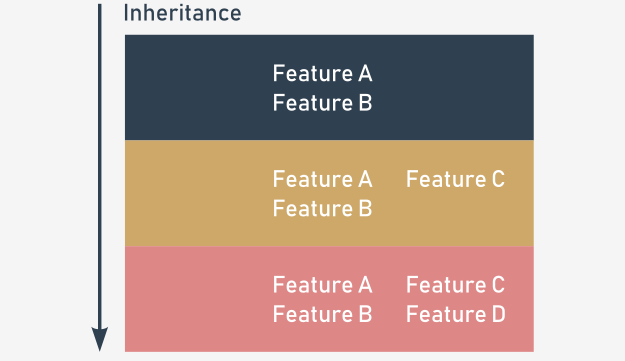
Cada **subclasse é mais especializada** (ou mais específica) do que a sua superclasse. Inversamente, cada **superclasse é mais geral** (mais abstrata) do que qualquer uma das suas subclasses.

Note-se que presumimos que uma classe pode ter apenas uma superclasse - isto nem sempre é verdade, mas discutiremos este assunto um pouco mais tarde.

## Herança

Vamos definir um dos conceitos fundamentais da programação de objetos, denominado **herança**. Qualquer objeto ligado a um nível específico de uma hierarquia de classes **herda todos os traços (assim como os requisitos e qualidades) definidos dentro de qualquer uma das superclasses**.

A classe de origem do objeto pode definir novos traços (bem como requisitos e qualidades) que serão herdados por qualquer uma das suas subclasses.



Não deverá ter quaisquer problemas em adequar esta regra a exemplos específicos, quer se aplique a animais, quer a veículos.

## O que tem um objeto?

A convenção de programação de objetos pressupõe que **cada objeto existente pode ser equipado com três grupos de atributos**:

* um objeto tem um **nome** que o identifica de forma única dentro do seu namespace de origem (embora possa haver alguns objetos anónimos, também)
* um objeto tem um **conjunto de propriedades** individuais que o tornam original, único ou excecional (embora seja possível que alguns objetos possam não ter quaisquer propriedades)
* um objeto tem um **conjunto de capacidades para realizar atividades específicas**, capazes de mudar o próprio objeto, ou alguns dos outros objetos.

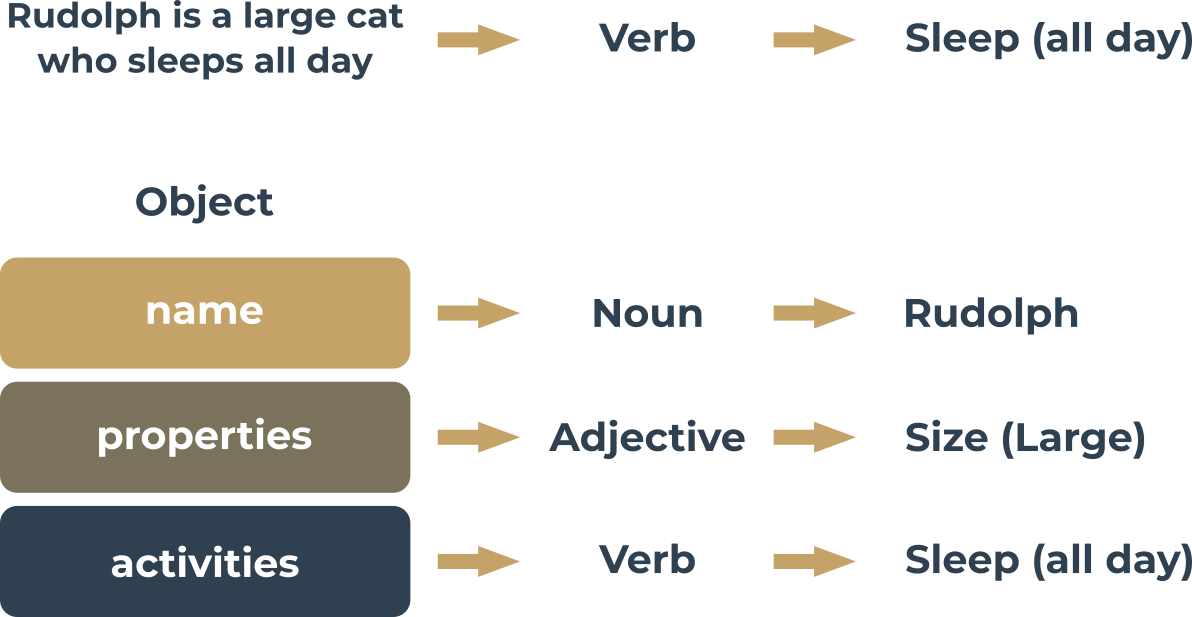
Há uma dica (embora isto nem sempre funcione) que pode ajudá-lo a identificar qualquer uma das três esferas acima. Sempre que descrever um objeto e utilizar:

* um substantivo - provavelmente define o nome do objeto;
* um adjetivo - provavelmente define a propriedade do objeto;
* um verbo - provavelmente define a atividade do objeto.

Duas frases de amostra devem servir como um bom exemplo:

* A pink Cadillac went quickly.  
    
  Nome do objeto = Cadillac  
  Classe mãe = Wheeled vehicles  
  Propriedade = Cor (pink)  
  Atividade = Go (quickly)

* Max is a large cat who sleeps all day.  
    
  Nome do objeto = Max  
  Classe mãe = Cat  
  Propriedade = Size (large)  
  Atividade = Sleep (all day)



# A sua primeira classe

A programação de objetos é a **arte de definir e expandir classes**. Uma classe é um modelo de uma parte muito específica da realidade, refletindo propriedades e atividades encontradas no mundo real.

As classes definidas no início são demasiado gerais e imprecisas para cobrir o maior número possível de casos reais.

Não há obstáculo à definição de subclasses novas e mais precisas. Herdarão tudo da sua superclasse, para que o trabalho que foi para a sua criação não seja desperdiçado.

A nova classe pode acrescentar novas propriedades e novas atividades, e por isso pode ser mais útil em aplicações específicas. Obviamente, pode ser utilizada como uma superclasse para qualquer número de subclasses recentemente criadas.

O processo não precisa de ter um fim. Pode criar tantas classes quantas precisar.

A classe que define nada tem a ver com o objeto: **a existência de uma classe não significa que qualquer um dos objetos compatíveis será automaticamente criado**. A própria classe não é capaz de criar um objeto - tem de ser você a criá-lo, e o Python permite-lhe fazer isso.

É tempo de definir a classe mais simples e de criar um objeto. Veja o exemplo abaixo:

class TheSimplestClass:

pass

Definimos aí uma classe. A classe é bastante pobre: não tem propriedades nem atividades. Na verdade, está **vazia**, mas isso não importa por agora. Quanto mais simples for a classe, melhor para os nossos propósitos.

**A definição começa com a keyword**class. A keyword é seguida por um **identificador que dará nome à classe** (nota: não confundir com o nome do objeto - estas são duas coisas diferentes).

A seguir, acrescenta **dois pontos** (:), visto as classes, como as funções, formarem o seu próprio bloco nested. O conteúdo dentro do bloco define todas as propriedades e atividades da classe.

A keyword pass preenche a classe com nada. Não contém quaisquer métodos ou propriedades.

## O seu primeiro objeto

A classe recentemente definida torna-se uma ferramenta capaz de criar novos objetos. A ferramenta tem de ser utilizada explicitamente, a pedido.

Imagine que quer criar um (exatamente um) objeto da classe TheSimplestClass .

Para tal, é necessário atribuir uma variável para armazenar o objeto recentemente criado dessa classe, e criar um objeto ao mesmo tempo.

Fá-lo da seguinte forma:

my\_first\_object = TheSimplestClass()

Nota:

* o nome da classe tenta fingir que é uma função - consegue ver isto? Discutiremos isso em breve;
* o objeto recentemente criado está equipado com tudo o que a classe traz; como esta classe está completamente vazia, o objeto também está vazio.

O ato de criar um objeto da classe selecionada é também chamado **instanciação** (uma vez que o objeto se torna uma **instância da classe**).

Vamos deixar as classes em paz por um breve momento, pois vamos agora dizer-lhe algumas palavras sobre *stacks*. Sabemos que o conceito de classes e objetos pode ainda não estar totalmente claro. Não se preocupe, explicaremos tudo muito em breve.

**Key takeaways**

1. Uma **classe** é uma ideia (mais ou menos abstrata) que pode ser usada para criar uma série de encarnações - tal encarnação é chamada um **objeto**.

2. Quando uma classe é derivada de outra classe, a sua relação é denominada **herança**. A classe que deriva da outra classe é nomeada uma **subclasse**. O segundo lado desta relação é denominado **superclasse**. Uma forma de apresentar uma tal relação é com um **diagrama de herança**, onde:

* as superclasses são sempre apresentadas **acima** das suas subclasses;
* relações entre classes são mostradas como setas dirigidas **da subclasse para a sua superclasse**.

3. Os objetos estão equipados com:

* um **nome** que os identifica e que nos permite distingui-los;
* um conjunto de **propriedades** (o conjunto pode estar vazio)
* um conjunto de **métodos** (também pode estar vazio)

4. Para definir uma classe Python, é necessário utilizar a keyword class . Por exemplo:

class This\_Is\_A\_Class:

pass

5. Para criar um objeto da classe previamente definida, é necessário utilizar a classe como se fosse uma função. Por exemplo:

this\_is\_an\_object = This\_Is\_A\_Class()

**Exercício 1**

Se assumíssemos que pitões, víboras e cobras são subclasses da mesma superclasse, como lhe chamaria?

**Exercício 2**

Tente nomear algumas subclasses de classe pitão.

**Exercício 3**

Pode nomear uma das suas classes apenas "classe"?

# O que é uma stack?

**Uma stack (pilha) é uma estrutura desenvolvida para armazenar dados de uma forma muito específica**. Imagine uma pilha de moedas. Não se pode colocar uma moeda em mais lado nenhum, a não ser no topo da pilha.

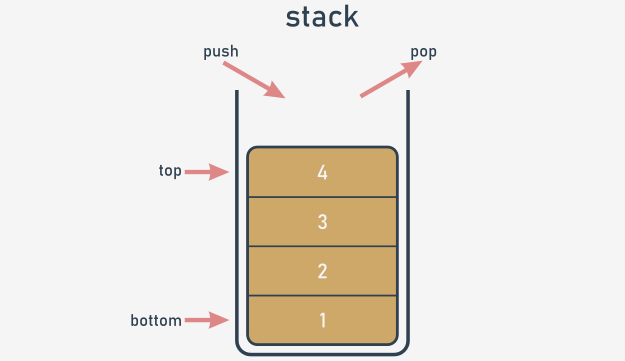
Da mesma forma, não se pode tirar uma moeda da pilha de qualquer outro lugar que não seja do topo da pilha. Se quiser obter a moeda que está no fundo, tem de retirar todas as moedas dos níveis superiores.

O nome alternativo para uma pilha (mas apenas em terminologia de TI) é **LIFO**.

É uma abreviação para uma descrição muito clara do comportamento da pilha: **Last In - First Out**. A moeda que chegou em último lugar à pilha sairá primeiro.

**Uma stack é um objeto** com duas operações elementares, convencionalmente denominadas **push** (quando um novo elemento é colocado no topo) e **pop** (quando um elemento existente é retirado do topo).

As stacks são muitas vezes utilizadas em muitos algoritmos clássicos, e é difícil imaginar a implementação de muitas ferramentas amplamente utilizadas sem o uso de stack.

  
  
Vamos implementar uma stack no Python. Esta será uma stack muito simples, e mostrar-lhe-emos como fazê-lo em duas abordagens independentes: processual e objetiva.

Vamos começar com a primeira.

**A stack - a abordagem processual**

Primeiro, é preciso decidir como armazenar os valores que chegarão à stack. Sugerimos a utilização do mais simples dos métodos, e **o emprego de uma lista** para este trabalho. Vamos assumir que o tamanho da stack não é limitado de forma alguma. Vamos também assumir que o último elemento da lista armazena o elemento superior.

A stack em si já está criada:

stack = []

Estamos prontos para **definir uma função que coloca um valor na stack**. Aqui estão os pressupostos para tal:

* o nome para a função é push;
* a função obtém um parâmetro (este é o valor a ser colocado na stack)
* a função não devolve nada;
* a função anexa o valor do parâmetro ao fim da stack;

Foi assim que o fizemos - dê uma vista de olhos:

def push(val):

stack.append(val)

Agora é tempo de uma **função tirar um valor da stack**. É assim que o pode fazer:

* o nome da função é pop;
* a função não obtém nenhum parâmetro;
* a função devolve o valor retirado da stack
* a função lê o valor da parte superior da stack e remove-o.

A função está aqui:

def pop():

val = stack[-1]

del stack[-1]

return val

Nota: a função não verifica se existe algum elemento na stack.

Vamos juntar todas as peças para pôr a stack em movimento. O **programa completo** empurra três números para a stack, puxa-os para fora e imprime os seus valores no ecrã. Pode vê-lo na janela do editor.

stack = []

def push(val):

stack.append(val)

def pop():

val = stack[-1]

del stack[-1]

return val

push(3)

push(2)

push(1)

print(pop())

print(pop())

print(pop())

O programa faz output do seguinte texto para o ecrã:

1

2

3

**output**

Teste-o.

**A stack - a abordagem processual vs. a abordagem orientada a objetos**

A stack processual está pronta. Claro que existem algumas fraquezas, e a implementação poderia ser melhorada de muitas maneiras (aproveitar as exceções para o trabalho é uma boa ideia), mas em geral a stack está totalmente implementada, e pode utilizá-la se for necessário.

Mas quanto mais vezes a utilizar, mais desvantagens encontrará. Aqui estão algumas delas:

* a variável essencial (a lista da stack) é altamente **vulnerável**; qualquer pessoa pode modificá-la de uma forma incontrolável, destruindo a stack, de facto; isto não significa que tenha sido feita de forma maliciosa - pelo contrário, pode acontecer como resultado de descuido, por exemplo, quando alguém confunde nomes de variáveis; imagine que escreveu acidentalmente algo do género:
* stack[0] = 0

O funcionamento da stackserá completamente desorganizado;

* pode também acontecer que um dia precise de mais do que uma stack; terá de criar outra lista para o armazenamento da stack, e provavelmente outras funções push e pop também;

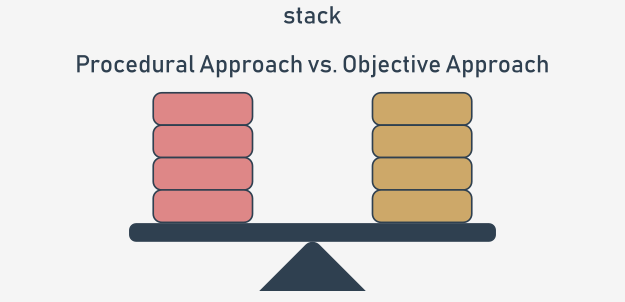
* também pode acontecer que precisa não só de funções push e pop , mas também de algumas outras conveniências; poderia certamente implementá-las, mas tente imaginar o que aconteceria se tivesse dezenas de stacks implementadas separadamente.

A abordagem objetiva proporciona soluções para cada um dos problemas acima referidos. Vamos nomeá-las primeiro:

* a capacidade de ocultar (proteger) valores selecionados contra o acesso não autorizado chama-se **encapsulamento; os valores encapsulados não podem ser acedidos nem modificados se quiser utilizá-los exclusivamente**;

* quando se tem uma classe implementando todos os comportamentos de stack necessários, pode-se produzir quantas stacks se quiser; não é necessário copiar ou replicar qualquer parte do código;

* a capacidade de enriquecer a stack com novas funções vem da herança; pode-se criar uma nova classe (uma subclasse) que herda todos os traços existentes da superclasse, e acrescenta alguns novos.



Vamos agora escrever uma nova implementação de stack a partir do zero. Desta vez, vamos utilizar a abordagem objetiva, guiando-o passo a passo para o mundo da programação de objetos.

**A stack - a abordagem ao objeto**

Evidentemente, a ideia principal continua a ser a mesma. Usaremos uma lista como armazenamento da stack. Só temos que saber como colocar a lista na classe.

Comecemos do início absoluto - é assim que começa a stack objetiva:

class Stack:

Agora, esperamos duas coisas dela:

* queremos que a classe tenha **uma propriedade como armazenamento da stack** - temos de **"instalar" uma lista dentro de cada objeto da classe** (nota: cada objeto tem de ter a sua própria lista - a lista não deve ser partilhada entre diferentes stacks)
* em seguida, queremos que **a lista seja escondida** da vista dos utilizadores da classe.

Como é isto feito?

Ao contrário de outras linguagens de programação, o Python não tem meios de lhe permitir declarar tal propriedade sem mais nem menos.

Em vez disso, é necessário acrescentar uma declaração ou instrução específica. As propriedades têm que ser adicionadas à classe manualmente.

Como garante que tal atividade ocorre cada vez que a nova stack é criada?

Há uma forma simples de o fazer - é necessário **equipar a classe com uma função específica** - a sua especificidade é dupla:

* tem de ser nomeada de forma estrita;
* é invocada implicitamente, quando o novo objeto é criado.

Tal função é chamada de **construtor**, pois o seu objetivo geral é a **construção de um novo objeto**. O construtor deve saber tudo sobre a estrutura do objeto, e deve realizar todas as inicializações necessárias.

Vamos adicionar um construtor muito simples à nova classe. Veja o snippet:

class Stack:

def \_\_init\_\_(self):

print("Hi!")

stack\_object = Stack()

E agora:

* o nome do construtor é sempre \_\_init\_\_;
* tem de ter **pelo menos um parâmetro** (discutiremos isto mais tarde); o parâmetro é usado para representar o objeto recentemente criado - pode usar o parâmetro para manipular o objeto, e para o enriquecer com as propriedades necessárias; fará uso disto em breve;
* nota: o parâmetro obrigatório é geralmente chamado self - é apenas **uma convenção, mas deve segui-la** - simplifica o processo de leitura e compreensão do seu código.

O código está no editor. Execute-o agora.

class Stack: # Defining the Stack class.

def \_\_init\_\_(self): # Defining the constructor function.

print("Hi!")

stack\_object = Stack() # Instantiating the object.

Aqui está o seu output:

Hi!

**output**

Nota - não há vestígios de invocação do construtor dentro do código. Foi invocado implícita e automaticamente. Vamos fazer uso disso agora.

Qualquer mudança que faça no interior do construtor que modifique o estado do parâmetro self será refletido no objeto recém-criado.

Isto significa que se pode acrescentar qualquer propriedade ao objeto e a propriedade permanecerá lá até que o objeto termine a sua vida ou a propriedade seja explicitamente removida.

Agora vamos **adicionar apenas uma propriedade ao novo objeto** - uma lista para uma stack. Vamos nomeá-lo stack\_list.

Tal como aqui:

class Stack:

def \_\_init\_\_(self):

self.stack\_list = []

stack\_object = Stack()

print(len(stack\_object.stack\_list))

Nota:

* utilizámos a **notação pontilhada**, tal como quando invocamos métodos; esta é a convenção geral para aceder às propriedades de um objeto - é necessário dar um nome ao objeto, colocar um ponto (.) depois dele, e especificar o nome da propriedade desejada; não use parêntesis! Não deseja invocar um método - deseja **aceder a uma propriedade**;
* se definir o valor de uma propriedade pela primeira vez (como no construtor), está a criá-la; a partir desse momento, o objeto tem a propriedade e está pronto a usar o seu valor;
* fizemos algo mais no código - tentámos aceder à propriedade stack\_list de fora da classe imediatamente após o objeto ter sido criado; queremos verificar o comprimento atual da stack - conseguimos?

Sim, conseguimos - o código produz o seguinte output:

0

**output**

Isto não é que queremos da stack. Preferimos que stack\_list seja **ocultado do mundo exterior**. Será isso possível?

Sim, e é simples, mas não muito intuitivo.

Dê uma vista de olhos - adicionamos dois underscores antes do nome stack\_list - nada mais:

class Stack:

def \_\_init\_\_(self):

self.\_\_stack\_list = []

stack\_object = Stack()

print(len(stack\_object.\_\_stack\_list))

A alteração invalida o programa.

Porquê?

Quando qualquer componente de classe tem um **nome que começa com dois underscores (**\_\_**), torna-se privado** - isto significa que só pode ser acedido de dentro da classe.

Não se pode vê-lo do mundo exterior. É assim que o Python implementa o conceito de **encapsulamento**.

Execute o programa para testar as nossas suposições - uma exceção AttributeError deve ser levantada.

**A abordagem ao objeto: uma stack do zero**

Agora é altura de as duas funções (métodos) implementarem as operações *push* e *pop*. O Python assume que uma função deste tipo (uma atividade de classe) deve ser **imersa dentro do corpo da classe** - tal como um construtor.

Queremos invocar estas funções para valores push e pop . Isto significa que ambos devem ser acessíveis a todos os utilizadores da classe (em contraste com a lista previamente construída, que está escondida dos utilizadores comuns da classe).

Tal componente é chamado **público**, pelo que **não se pode começar o seu nome com dois (ou mais) underscores**. Há mais um requisito - **o nome não deve ter mais do que um underscore à direita**. Uma vez que nenhum underscore à direita satisfaz totalmente o requisito, pode-se assumir que o nome é aceitável.

As funções em si são simples. Dê uma vista de olhos:

class Stack:

def \_\_init\_\_(self):

self.\_\_stack\_list = []

def push(self, val):

self.\_\_stack\_list.append(val)

def pop(self):

val = self.\_\_stack\_list[-1]

del self.\_\_stack\_list[-1]

return val

stack\_object = Stack()

stack\_object.push(3)

stack\_object.push(2)

stack\_object.push(1)

print(stack\_object.pop())

print(stack\_object.pop())

print(stack\_object.pop())

No entanto, há algo realmente estranho no código. As funções parecem familiares, mas têm mais parâmetros do que as suas contrapartes processuais.

Aqui, ambas as funções têm um parâmetro chamado self na primeira posição da lista de parâmetros.

É necessário? Sim, é.

Todos os métodos têm de ter este parâmetro. Desempenha o mesmo papel que o primeiro parâmetro construtor.

**Permite ao método aceder a entidades (propriedades e atividades/métodos) realizadas pelo objeto atual**. Não se pode omiti-lo. Sempre que o Python invoca um método, envia implicitamente o objeto atual como o primeiro argumento.

Isto significa que um **método é obrigado a ter pelo menos um parâmetro, que é utilizado pelo próprio Python** - não tem qualquer influência sobre ele.

Se o seu método não necessita de parâmetros, este deve ser especificado de qualquer forma. Se for concebido para processar apenas um parâmetro, é necessário especificar dois, e o primeiro continua a ter o mesmo papel.

Há mais uma coisa que requer explicação - a forma como os métodos são invocados a partir do interior da variável \_\_stack\_list .

Felizmente, é muito mais simples do que parece:

* a primeira fase entrega o objeto como um todo → self;
* a seguir, precisa de chegar à \_\_stack\_list lista → self.\_\_stack\_list;
* com \_\_stack\_list pronto a ser utilizado, pode executar o terceiro e último passo → self.\_\_stack\_list.append(val).

A declaração de classe está completa, e todos os seus componentes foram listados. A classe está pronta a ser utilizada.

Ter uma tal classe abre algumas novas possibilidades. Por exemplo, pode agora ter mais do que uma stack a comportar-se da mesma maneira. Cada stack terá a sua própria cópia de dados privados, mas utilizará o mesmo conjunto de métodos.

Isto é exatamente o que queremos para este exemplo.

Analise o código:

class Stack:

def \_\_init\_\_(self):

self.\_\_stack\_list = []

def push(self, val):

self.\_\_stack\_list.append(val)

def pop(self):

val = self.\_\_stack\_list[-1]

del self.\_\_stack\_list[-1]

return val

stack\_object\_1 = Stack()

stack\_object\_2 = Stack()

stack\_object\_1.push(3)

stack\_object\_2.push(stack\_object\_1.pop())

print(stack\_object\_2.pop())

Existem **duas stacks criadas a partir da mesma classe base**. Elas trabalham **de forma independente**. Pode fazer mais delas se o desejar.

Execute o código no editor e veja o que acontece. Realize as suas próprias experiências.

Analise o snippet abaixo - criámos três objetos da classe Stack. A seguir, fizemos malabarismos com eles. Tente prever o valor de output no ecrã.

class Stack:

def \_\_init\_\_(self):

self.\_\_stack\_list = []

def push(self, val):

self.\_\_stack\_list.append(val)

def pop(self):

val = self.\_\_stack\_list[-1]

del self.\_\_stack\_list[-1]

return val

little\_stack = Stack()

another\_stack = Stack()

funny\_stack = Stack()

little\_stack.push(1)

another\_stack.push(little\_stack.pop() + 1)

funny\_stack.push(another\_stack.pop() - 2)

print(funny\_stack.pop())

Então, qual é o resultado? Execute o programa e verifique se estava certo.

Agora, vamos um pouco mais longe. Vamos **adicionar uma nova classe para lidar com stacks**.

A nova classe deve ser capaz de **avaliar a soma de todos os elementos atualmente armazenados na stack**.

Não queremos modificar a stack definida previamente. Já é suficientemente bom nas suas aplicações, e não queremos que seja alterado de forma alguma. Queremos uma nova stack com novas capacidades. Por outras palavras, queremos construir uma subclasse da classe já existente Stack .

O primeiro passo é fácil: basta **definir uma nova subclasse apontando para a classe que será usada como a superclasse**.

Isto é o que parece:

class AddingStack(Stack):

pass

A classe ainda não define nenhum componente novo, mas isso não significa que esteja vazia. **Recebe todos os componentes definidos pela sua superclasse** - o nome da superclasse é escrito antes dos dois pontos, diretamente após o nome da nova classe.

Isto é o que queremos da nova stack:

* queremos o método push não só para empurrar o valor para a stack, mas também para acrescentar o valor à variável sum ;
* queremos a função pop não só para tirar o valor da stack, mas também para subtrair o valor da variável sum .

Em primeiro lugar, vamos adicionar uma nova variável à classe. Será uma **variável privada**, como a lista de stack. Não queremos que ninguém manipule o valor sum .

Como já sabe, adicionar uma nova propriedade à classe é feito pelo construtor. Já sabe como fazer isso, mas há algo realmente intrigante no interior do construtor. Dê uma vista de olhos:

class AddingStack(Stack):

def \_\_init\_\_(self):

Stack.\_\_init\_\_(self)

self.\_\_sum = 0

A segunda linha do corpo do construtor cria uma propriedade chamada \_\_sum - irá armazenar o total de todos os valores da stack.

Mas a linha antes que pareça diferente. O que é que faz? É realmente necessário? Sim, é.

Ao contrário de muitas outras linguagens, o Python obriga-o a **invocar explicitamente o construtor de uma superclasse**. Omitir este ponto terá efeitos nocivos - o objeto será privado da lista \_\_stack\_list . Uma tal stack não funcionará corretamente.

Esta é a única vez em que se pode invocar explicitamente qualquer um dos construtores disponíveis - pode ser feito dentro do construtor da subclasse.

Observe a sintaxe:

* especifica o nome da superclasse (esta é a classe cujo construtor pretende executar)
* coloca um ponto (.) depois disso;
* especifica o nome do construtor;
* tem de apontar para o objeto (a instância da classe) que tem de ser inicializado pelo construtor - é por isso que tem de especificar o argumento e utilizar a variável self aqui; nota: **a invocação de qualquer método (incluindo construtores) de fora da classe nunca requer que se coloque o argumento**self**na lista do argumento** - invocar um método de dentro da classe exige o uso explícito do argumento self , e tem de ser colocado em primeiro lugar na lista.

Nota: é geralmente uma prática recomendada invocar o construtor da superclasse antes de quaisquer outras inicializações que deseje realizar dentro da subclasse. Esta é a regra que temos seguido no snippet.

class Stack:

def \_\_init\_\_(self):

self.\_\_stack\_list = []

def push(self, val):

self.\_\_stack\_list.append(val)

def pop(self):

val = self.\_\_stack\_list[-1]

del self.\_\_stack\_list[-1]

return val

class AddingStack(Stack):

def \_\_init\_\_(self):

Stack.\_\_init\_\_(self)

self.\_\_sum = 0

Em segundo lugar, vamos acrescentar dois métodos. Mas deixe-nos perguntar-lhe: está realmente a acrescentar? Já temos estes métodos na superclasse. Podemos fazer algo assim?

Sim, podemos. Isto significa que vamos **alterar a funcionalidade dos métodos, não os seus nomes**. Podemos dizer mais precisamente que a interface (a forma como os objetos são manuseados) da classe permanece a mesma quando se altera a implementação ao mesmo tempo.

Vamos começar com a implementação da função push . Isto é o que esperamos dela:

* acrescentar o valor à variável \_\_sum ;
* empurrar o valor para a stack.

Nota: a segunda atividade já foi implementada dentro da superclasse - por isso podemos utilizá-la. Além disso, temos de o utilizar, uma vez que não há outra forma de aceder à variável \_\_stackList .

É assim que o método push se parece na subclasse:

def push(self, val):

self.\_\_sum += val

Stack.push(self, val)

Note-se a forma como invocámos a implementação anterior do método push (o disponível na superclasse):

* temos de especificar o nome da superclasse; isto é necessário para indicar claramente a classe que contém o método, para evitar confundi-la com qualquer outra função do mesmo nome;
* temos de especificar o objeto alvo e de o passar como primeiro argumento (não é implicitamente acrescentado à invocação neste contexto).

Dizemos que o método push foi substituído - o mesmo nome que na superclasse representa agora uma funcionalidade diferente.

Este é a nova função pop :

def pop(self):

val = Stack.pop(self)

self.\_\_sum -= val

return val

Até agora, definimos a variável \_\_sum , mas não fornecemos um método para obter o seu valor. Parece estar escondido. Como podemos revelá-lo e fazê-lo de uma forma que ainda o proteja de modificações?

Temos de definir um novo método. Vamos nomeá-lo get\_sum. A sua única tarefa será **devolver o valor**\_\_sum.

Aqui está:

def get\_sum(self):

return self.\_\_sum

Portanto, vejamos o programa no editor.

class Stack:

def \_\_init\_\_(self):

self.\_\_stack\_list = []

def push(self, val):

self.\_\_stack\_list.append(val)

def pop(self):

val = self.\_\_stack\_list[-1]

del self.\_\_stack\_list[-1]

return val

class AddingStack(Stack):

def \_\_init\_\_(self):

Stack.\_\_init\_\_(self)

self.\_\_sum = 0

def get\_sum(self):

return self.\_\_sum

def push(self, val):

self.\_\_sum += val

Stack.push(self, val)

def pop(self):

val = Stack.pop(self)

self.\_\_sum -= val

return val

stack\_object = AddingStack()

for i in range(5):

stack\_object.push(i)

print(stack\_object.get\_sum())

for i in range(5):

print(stack\_object.pop())

O código completo da classe está lá. Podemos verificar o seu funcionamento agora, e fazemo-lo com a ajuda de muito poucas linhas adicionais de código.

Como pode ver, adicionamos cinco valores subsequentes à stack, imprimimos a sua soma, e retiramo-los todos da stack.

Muito bem, esta foi uma introdução muito breve à programação de objetos de Python. Em breve vamos falar-lhe de tudo isto com mais detalhes.

# Key takeaways

1. Uma **stack** é um objeto concebido para armazenar dados utilizando o modelo **LIFO**. A stack realiza geralmente pelo menos duas operações, denominadas **push()** e **pop()**.

2. A implementação da stack num modelo processual levanta vários problemas que podem ser resolvidos pelas técnicas oferecidas pelo **OOP** (**O**bject **O**riented **P**rogramming):

3. Um **método** de classe é na realidade uma função declarada dentro da classe e capaz de aceder a todos os componentes da classe.

4. A parte da classe Python responsável pela criação de novos objetos é chamada de **construtor**, e é implementada como um método do nome \_\_init\_\_.

5. Cada declaração de método de classe deve conter pelo menos um parâmetro (sempre o primeiro) geralmente referido como self, e é usado pelos objetos para se identificarem.

6. Se quisermos esconder qualquer um dos componentes de uma classe do mundo exterior, devemos começar o seu nome com \_\_. Tais componentes são chamados **privados**.

**Exercício 1**

Assumindo que existe uma classe chamada Snakes, escreva a primeira linha da declaração de classe Python , expressando o facto de a nova classe ser na realidade uma subclasse de Snake.

**Exercício 2**

Faltou alguma coisa na seguinte declaração — o quê?

class Snakes

def \_\_init\_\_():

self.sound = 'Sssssss'

**Exercício 3**

Modifique o código para garantir que a propriedade venomous é privada.

class Snakes

def \_\_init\_\_(self):

self.venomous = True

# LAB Contar Stacks

Fácil/Médio

## Objetivos

* melhorar as competências do aluno na definição das classes;
* utilizar classes existentes para criar novas classes equipadas com novas funcionalidades.

## Cenário

Mostrámos recentemente como alargar as possibilidades de *Stack* definindo uma nova classe (ou seja, uma subclasse) que mantém todas as características herdadas e acrescenta alguns novos.

A sua tarefa é alargar o comportamento da classe Stack de modo a que a classe seja capaz de contar todos os elementos que são empurrados e popped (assumimos que a contagem de pops é suficiente). Utilize a classe Stack que fornecemos no editor.

Siga as dicas:

* introduza uma propriedade concebida para contar operações pop e nomeie-a de uma forma que garanta a sua ocultação;
* inicialize-a a zero no interior do construtor;
* forneça um método que devolva o valor atualmente atribuído ao contador (nomeie-o get\_counter()).

Complete o código no editor. Execute-o para verificar se o seu código faz o output 100.

## RESPOSTA:

class Stack:

def \_\_init\_\_(self):

self.\_\_stk = []

def push(self, val):

self.\_\_stk.append(val)

def pop(self):

val = self.\_\_stk[-1]

del self.\_\_stk[-1]

return val

class CountingStack(Stack):

def \_\_init\_\_(self):

#

# Fill the constructor with appropriate actions.

#

def get\_counter(self):

#

# Present the counter's current value to the world.

#

def pop(self):

#

# Do pop and update the counter.

#

stk = CountingStack()

for i in range(100):

stk.push(i)

stk.pop()

print(stk.get\_counter())

# LAB Queue, também conhecida como FIFO

Fácil/Médio

## Objetivos

* melhorar as competências do aluno na definição das classes a partir do zero;
* implementação de estruturas de dados standard como classes.

## Cenário

Como já sabe, uma *stack* é uma estrutura de dados que realiza o chamado modelo LIFO (Last In - First Out). É fácil e já se habituou perfeitamente a isso.

Vamos provar algo novo agora. Uma *queue* (fila de espera) é um modelo de dados caraterizado pelo termo **FIFO: First In - Fist Out**. Nota: uma queue regular (fila) que conhece das lojas ou correios funciona exatamente da mesma maneira - um cliente que chegou primeiro também é servido primeiro.

A sua tarefa é implementar a classe Queue com duas operações básicas:

* put(element), que coloca um elemento no final da queue;
* get(), que toma um elemento da frente da queue e o devolve como resultado (a queue não pode estar vazia para o executar com êxito.)

Siga as dicas:

* utilize uma lista como seu aramazenamento (tal como fizemos na stack)
* put() deve anexar elementos ao início da lista, enquanto get() deve remover os elementos do final da lista;
* defina uma nova exceção chamada QueueError (escolha uma exceção para derivá-la) e levante-a quando get() tenta operar numa lista vazia.

Complete o código que fornecemos no editor. Execute-o para verificar se o seu output é semelhante ao nosso.

## Output esperado

1

dog

False

Queue error

## RESPOSTA

class QueueError(IndexError):

pass

class Queue:

def \_\_init\_\_(self):

self.queue = []

def put(self, elem):

self.queue.insert(0, elem)

def get(self):

if len(self.queue) > 0:

elem = self.queue[-1]

del self.queue[-1]

return elem

else:

raise QueueError

que = Queue()

que.put(1)

que.put("dog")

que.put(False)

try:

for i in range(4):

print(que.get())

except:

print("Queue error")

# LAB Queue também conhecida como FIFO, parte 2

Fácil/Médio

## Objetivos

* Melhorar as competências do aluno na definição de subclasses;
* adicionar uma nova funcionalidade a uma classe existente.

## Cenário

A sua tarefa é alargar ligeiramente as capacidades da classe Queue . Queremos que ele tenha um método sem parâmetros que devolva True se a queue estiver vazia e False caso contrário.

Complete o código que fornecemos no editor. Execute-o para verificar se ele produz um resultado semelhante ao nosso.

Abaixo pode copiar o código que utilizámos no laboratório anterior:

Verifique

## Output esperado

1

dog

False

Queue empty

## RESPOSTA:

class QueueError(IndexError):

pass

class Queue:

def \_\_init\_\_(self):

self.queue = []

def put(self,elem):

self.queue.insert(0,elem)

def get(self):

if len(self.queue) > 0:

elem = self.queue[-1]

del self.queue[-1]

return elem

else:

raise QueueError

class SuperQueue(Queue):

def isempty(self):

return len(self.queue) == 0

que = SuperQueue()

que.put(1)

que.put("dog")

que.put(False)

for i in range(4):

if not que.isempty():

print(que.get())

else:

print("Queue empty")

**Variáveis de instância**

Em geral, uma classe pode ser equipada com dois tipos diferentes de dados para formar as propriedades de uma classe. Já viu um deles quando estávamos a olhar para stacks.

Este tipo de propriedade de classe existe quando e só quando é explicitamente criada e adicionada a um objeto. Como já sabe, isto pode ser feito durante a inicialização do objeto, realizada pelo construtor.

Além disso, pode ser feito em qualquer momento da vida do objeto. Mais, qualquer propriedade existente pode ser removida em qualquer altura.

Uma tal abordagem tem algumas consequências importantes:

* objetos diferentes da mesma classe **podem possuir conjuntos diferentes de propriedades**;
* deve haver uma forma de **verificar com segurança se um objeto específico possui a propriedade** que pretende utilizar (a menos que queira provocar uma exceção - vale sempre a pena considerar)
* cada objeto **carrega o seu próprio conjunto de propriedades** - eles não interferem uns com os outros de forma alguma.

Tais variáveis (propriedades) são chamadas **variáveis de instância**.

A palavra *instância* sugere que elas estão intimamente ligadas aos objetos (que são instâncias de classe), não às classes propriamente ditas. Vamos dar-lhes uma olhadela mais atenta.

Aqui está um exemplo:

class ExampleClass:

def \_\_init\_\_(self, val = 1):

self.first = val

def set\_second(self, val):

self.second = val

example\_object\_1 = ExampleClass()

example\_object\_2 = ExampleClass(2)

example\_object\_2.set\_second(3)

example\_object\_3 = ExampleClass(4)

example\_object\_3.third = 5

print(example\_object\_1.\_\_dict\_\_)

print(example\_object\_2.\_\_dict\_\_)

print(example\_object\_3.\_\_dict\_\_)

Precisa de uma explicação adicional antes de entrarmos em qualquer outro detalhe. Dê uma vista de olhos às três últimas linhas do código.

Os objetos Python, quando criados, são dotados de um **pequeno conjunto de propriedades e métodos pré-definidos**. Cada objeto tem-nos, quer os queira ou não. Um deles é uma variável chamada \_\_dict\_\_ (é um dicionário).

A variável contém os nomes e valores de todas as propriedades (variáveis) que o objeto carrega atualmente. Vamos utilizá-la para apresentar o conteúdo de um objeto em segurança.

Vamos agora mergulhar no código:

* a classe nomeada ExampleClass tem um construtor, que **cria incondicionalmente uma variável de instância** chamada first, e define-a com o valor passado através do primeiro argumento (da perspetiva do utilizador da classe) ou do segundo argumento (da perspetiva do construtor); note o valor por defeito do parâmetro - qualquer truque que se possa fazer com um parâmetro de função regular também pode ser aplicado aos métodos;

* a classe também tem um **método que cria outra variável de instância**, denominada second;

* criámos três objetos da classe ExampleClass, mas todas estas instâncias diferem:

* + example\_object\_1 só tem a propriedade nomeada first;

* + example\_object\_2 tem duas propriedades: first e second;

* + example\_object\_3 foi enriquecido com uma propriedade chamada third enquanto em movimento, fora do código da classe - isto é possível e totalmente admissível.

Os outputs do programa mostram claramente que os nossos pressupostos estão corretos - aqui está:

{'first': 1}

{'second': 3, 'first': 2}

{'third': 5, 'first': 4}

**output**

Há uma conclusão adicional que deve ser afirmada aqui: **a modificação de uma variável de instância de qualquer objeto não tem impacto em todos os objetos restantes**. As variáveis de instância estão perfeitamente isoladas umas das outras.

Dê uma vista de olhos ao exemplo modificado no editor.

class ExampleClass:

def \_\_init\_\_(self, val = 1):

self.\_\_first = val

def set\_second(self, val = 2):

self.\_\_second = val

example\_object\_1 = ExampleClass()

example\_object\_2 = ExampleClass(2)

example\_object\_2.set\_second(3)

example\_object\_3 = ExampleClass(4)

example\_object\_3.\_\_third = 5

print(example\_object\_1.\_\_dict\_\_)

print(example\_object\_2.\_\_dict\_\_)

print(example\_object\_3.\_\_dict\_\_)

É quase o mesmo que o anterior. A única diferença está nos nomes das propriedades. **Adicionámos dois underscores (**\_\_**)** à frente deles.

Como sabe, tal adição torna a variável de instância **privada** - torna-se inacessível a partir do mundo exterior.

O comportamento real destes nomes é um pouco mais complicado, por isso vamos executar o programa. Este é o output:

{'\_ExampleClass\_\_first': 1}

{'\_ExampleClass\_\_first': 2, '\_ExampleClass\_\_second': 3}

{'\_ExampleClass\_\_first': 4, '\_\_third': 5}

**output**

Consegue ver estes estranhos nomes cheios de underscores? De onde vieram eles?

Quando o Python vê que quer adicionar uma variável de instância a um objeto, e vai fazê-lo dentro de qualquer um dos métodos do objeto, ele **modula a operação** da seguinte forma:

* coloca um nome de classe antes do seu nome;
* coloca um underscore adicional no início.

É por isso que o \_\_first torna-se \_ExampleClass\_\_first.

**O nome é agora totalmente acessível a partir do exterior da classe**. Pode executar um código como este:

print(example\_object\_1.\_ExampleClass\_\_first)

e obterá um resultado válido, sem erros ou exceções.

Como pode ver, tornar uma propriedade privada é limitado.

**O mangling não funcionará se adicionar uma variável de instância privada fora do código de classe**. Neste caso, comportar-se-á como qualquer outra propriedade.

**Variáveis de classe**

Uma variável de classe é **uma propriedade que existe apenas numa cópia e é armazenada fora de qualquer objeto**.

Nota: não existe uma variável de instância se não houver nenhum objeto na classe; existe uma variável de classe numa cópia, mesmo que não haja objetos na classe.

As variáveis de classe são criadas de forma diferente dos seus irmãos de instância. O exemplo dir-lhe-á mais:

class ExampleClass:

counter = 0

def \_\_init\_\_(self, val = 1):

self.\_\_first = val

ExampleClass.counter += 1

example\_object\_1 = ExampleClass()

example\_object\_2 = ExampleClass(2)

example\_object\_3 = ExampleClass(4)

print(example\_object\_1.\_\_dict\_\_, example\_object\_1.counter)

print(example\_object\_2.\_\_dict\_\_, example\_object\_2.counter)

print(example\_object\_3.\_\_dict\_\_, example\_object\_3.counter)

Veja:

* existe uma atribuição na primeira lista da definição da classe - define a variável chamada counter até 0; inicializar a variável dentro da classe mas fora de qualquer um dos seus métodos torna a variável uma variável de classe;
* o acesso a tal variável tem o mesmo aspeto que o acesso a qualquer atributo de instância - pode vê-la no corpo do construtor; como pode ver, o construtor incrementa a variável por um; com efeito, a variável conta todos os objetos criados.

A execução do código causará o seguinte output:

{'\_ExampleClass\_\_first': 1} 3

{'\_ExampleClass\_\_first': 2} 3

{'\_ExampleClass\_\_first': 4} 3

Duas conclusões importantes vêm do exemplo:

* variáveis de classe **não são mostradas num objeto**\_\_dict\_\_ (isto é natural porque as variáveis de classe não são partes de um objeto) mas pode sempre tentar olhar para a variável do mesmo nome, mas ao nível da classe - vamos mostrar-lhe isto muito em breve;
* uma variável de classe **apresenta sempre o mesmo valor** em todas as instâncias de classe (objetos)

A mutilação do nome de uma variável de classe tem os mesmos efeitos que aqueles com que já está familiarizado.

Veja o exemplo no editor. Consegue adivinhar o seu output?

class ExampleClass:

\_\_counter = 0

def \_\_init\_\_(self, val = 1):

self.\_\_first = val

ExampleClass.\_\_counter += 1

example\_object\_1 = ExampleClass()

example\_object\_2 = ExampleClass(2)

example\_object\_3 = ExampleClass(4)

print(example\_object\_1.\_\_dict\_\_, example\_object\_1.\_ExampleClass\_\_counter)

print(example\_object\_2.\_\_dict\_\_, example\_object\_2.\_ExampleClass\_\_counter)

print(example\_object\_3.\_\_dict\_\_, example\_object\_3.\_ExampleClass\_\_counter)

Execute o programa e verifique se as suas previsões estavam corretas. Tudo funciona como esperado, não é verdade?

Dissemos-lhe antes que as variáveis de classe existem mesmo quando nenhuma instância de classe (objeto) tenha sido criada.

Agora vamos aproveitar a oportunidade para lhe mostrar **a diferença entre estas duas variáveis**\_\_dict\_\_, a da classe e a do objeto.

Veja o código no editor. A prova está lá.

class ExampleClass:

varia = 1

def \_\_init\_\_(self, val):

ExampleClass.varia = val

print(ExampleClass.\_\_dict\_\_)

example\_object = ExampleClass(2)

print(ExampleClass.\_\_dict\_\_)

print(example\_object.\_\_dict\_\_)

Vamos analisar mais de perto:

1. Definimos uma classe chamada ExampleClass;

1. A classe define uma variável de classe chamada varia;

1. O construtor de classe define a variável com o valor do parâmetro;

1. Nomear a variável é o aspeto mais importante do exemplo porque:
   * Alterar a atribuição para self.varia = val criaria uma variável de instância com o mesmo nome que o da classe;
   * Alterar a atribuição para varia = val operaria na variável local de um método; (encorajamo-lo vivamente a testar os dois casos acima referidos - isto facilitar-lhe-á a lembrar-se da diferença)
2. A primeira linha do código fora da classe imprime o valor do atributo ExampleClass.varia ; nota - usamos o valor antes que o primeiro objeto da classe seja instanciado.

Execute o código no editor e verifique o seu output.

Como pode ver, a classe \_\_dict\_\_ contém muito mais dados do que a contraparte do seu objeto. A maioria deles são inúteis agora - aquele que queremos que verifique cuidadosamente mostra o valor varia corrente.

Observe que o \_\_dict\_\_ do objeto está vazio - o objeto não tem variáveis de instância.

# Verificação da existência de um atributo

A atitude do Python em relação à instanciação de objetos levanta uma questão importante - em contraste com outras linguagens de programação, **não se pode esperar que todos os objetos da mesma classe tenham o mesmo conjunto de propriedades**.

Assim como no exemplo no editor. Olhe com atenção.

class ExampleClass:

def \_\_init\_\_(self, val):

if val % 2 != 0:

self.a = 1

else:

self.b = 1

example\_object = ExampleClass(1)

print(example\_object.a)

print(example\_object.b)

O objeto criado pelo construtor pode ter apenas um dos dois atributos possíveis: a ou b.

A execução do código produzirá o seguinte output:

1

Traceback (most recent call last):

File ".main.py", line 11, in

print(example\_object.b)

AttributeError: 'ExampleClass' object has no attribute 'b'

**output**

Como pode ver, o acesso a um atributo de objeto (classe) inexistente causa uma AttributeError exceção.

A instrução try-except dá-lhe a oportunidade de evitar problemas com propriedades inexistentes.

É fácil - veja o código no editor.

class ExampleClass:

def \_\_init\_\_(self, val):

if val % 2 != 0:

self.a = 1

else:

self.b = 1

example\_object = ExampleClass(1)

print(example\_object.a)

try:

print(example\_object.b)

except AttributeError:

pass

Como se pode ver, esta ação não é muito sofisticada. Essencialmente, apenas varremos o problema para debaixo do tapete.

Felizmente, há mais uma forma de lidar com o problema.

O Python fornece uma **função capaz de verificar com segurança se algum objeto/classe contém uma propriedade especificada**. A função é chamada hasattr, e espera que lhe sejam transmitidos dois argumentos:

* A classe ou o objeto a ser verificado;
* o nome da propriedade cuja existência tem de ser comunicada (nota: tem de ser uma string contendo o nome do atributo, e não apenas o nome)

A função devolve True ou False.

É assim que a pode utilizar:

class ExampleClass:

def \_\_init\_\_(self, val):

if val % 2 != 0:

self.a = 1

else:

self.b = 1

example\_object = ExampleClass(1)

print(example\_object.a)

if hasattr(example\_object, 'b'):

print(example\_object.b)

Não se esqueça que a função hasattr() também pode operar em classes. Pode utilizá-la **para descobrir se uma variável de classe está disponível**, tal como aqui no exemplo do editor.

class ExampleClass:

attr = 1

print(hasattr(ExampleClass, 'attr'))

print(hasattr(ExampleClass, 'prop'))

A função devolve True se a classe especificada contiver um determinado atributo, e False caso contrário.

Consegue adivinhar o output do código? Execute-o para verificar as suas suposições.

E mais um exemplo - veja o código abaixo e tente prever o seu output:

class ExampleClass:

a = 1

def \_\_init\_\_(self):

self.b = 2

example\_object = ExampleClass()

print(hasattr(example\_object, 'b'))

print(hasattr(example\_object, 'a'))

print(hasattr(ExampleClass, 'b'))

print(hasattr(ExampleClass, 'a'))

Foi bem sucedido? Execute o código para verificar as suas previsões.

Muito bem, chegámos ao fim desta secção. Na secção seguinte vamos falar de métodos, uma vez que os métodos conduzem os objetos e os tornam ativos.

# Key takeaways

1. Uma **variável de instância** é uma propriedade cuja existência depende da criação de um objeto. Cada objeto pode ter um conjunto diferente de variáveis de instância.

Além disso, eles podem ser livremente adicionados aos, e removidos dos, objetos durante a sua vida útil. Todas as variáveis de instância de objeto são armazenadas dentro de um dicionário dedicado, chamado \_\_dict\_\_, contido em cada objeto separadamente.

2. Uma variável de instância pode ser privada quando o seu nome começa por \_\_, mas não se esqueça que tal propriedade ainda é acessível de fora da classe, utilizando um **nome mangled** construído como \_ClassName\_\_PrivatePropertyName.

3. Uma **variável de classe** é uma propriedade que existe exatamente numa cópia, e não precisa de nenhum objeto criado para ser acessível. Tais variáveis não são mostradas como conteúdo \_\_dict\_\_ .

Todas as variáveis de classe de uma classe são armazenadas dentro de um dicionário dedicado, chamado \_\_dict\_\_, contido em cada classe separadamente.

4. Uma função chamada hasattr() pode ser utilizada para determinar se um qualquer objeto/classe contém uma propriedade especificada.

Por exemplo:

class Sample:

gamma = 0 # Class variable.

def \_\_init\_\_(self):

self.alpha = 1 # Instance variable.

self.\_\_delta = 3 # Private instance variable.

obj = Sample()

obj.beta = 2 # Another instance variable (existing only inside the "obj" instance.)

print(obj.\_\_dict\_\_)

Output do código:

{'alpha': 1, '\_Sample\_\_delta': 3, 'beta': 2}

**output**

**Exercício 1**

Quais das Python propriedades de classe são variáveis de instância, e quais são variáveis de classe? Quais delas são privadas?  
  
class Python:

population = 1

victims = 0

def \_\_init\_\_(self):

self.length\_ft = 3

self.\_\_venomous = False

Verifique

**Exercício 2**

Vai negar a \_\_venomous propriedade do objeto version\_2 , ignorando o facto de que a propriedade é privada. Como vai fazer isto?

version\_2 = Python()

Verifique

**Exercício 3**

Escreva uma expressão que verifica se o objeto version\_2 contém uma propriedade de instância chamada constrictor (sim, constr**i**ctor!).

Verifique

**Métodos em detalhe**

Vamos resumir todos os factos relativos à utilização de métodos em classes Python.

Como já sabe, um **método é uma função incorporada dentro de uma classe**.

Existe um requisito fundamental - um **método é obrigado a ter pelo menos um parâmetro** (não existem métodos sem parâmetros - um método pode ser invocado sem argumento, mas não declarado sem parâmetros).

O primeiro (ou único) parâmetro é geralmente chamado self. Sugerimos que siga a convenção - é normalmente usada, e causará algumas surpresas ao usar outros nomes para a mesma.

O nome self sugere a finalidade do parâmetro - **identifica o objeto para o qual o método é invocado**.

Se vai invocar um método, não deve passar o argumento para o parâmetro self - o Python irá configurá-lo por si.

O exemplo no editor mostra a diferença.

class Classy:

def method(self):

print("method")

obj = Classy()

obj.method()

Output do código:

method

**output**

Note a forma como criámos o objeto - **tratámos o nome da classe como uma função**, devolvendo um objeto recentemente instanciado da classe.

Se quiser que o método aceite outros parâmetros self, deve:

* colocá-los depois de self na definição do método;
* entregá-los durante a invocação sem especificar self (como anteriormente)

Tal como aqui:

class Classy:

def method(self, par):

print("method:", par)

obj = Classy()

obj.method(1)

obj.method(2)

obj.method(3)

Output do código:

method: 1

method: 2

method: 3

O parâmetro self é utilizado **para obter acesso à instância do objeto e às variáveis de classe**.

O exemplo mostra as duas formas de utilizar self:

class Classy:

varia = 2

def method(self):

print(self.varia, self.var)

obj = Classy()

obj.var = 3

obj.method()

Output do código:

2 3

**output**

A função self é também utilizado **para invocar outros métodos objeto/classe de dentro da classe**.

Tal como aqui:

class Classy:

def other(self):

print("other")

def method(self):

print("method")

self.other()

obj = Classy()

obj.method()

Output do código:

method

other

**output**

Se nomear um método como este: \_\_init\_\_, não será um método regular - será um **construtor**.

Se uma classe tem um construtor, é invocada automática e implicitamente quando o objeto da classe é instanciado.

O construtor:

* é **obrigado a ter o parâmetro**self(é definido automaticamente, como de costume);
* **pode (mas não é necessário) ter mais parâmetros** do que apenas self; se isto acontecer, a forma como o nome da classe é utilizado para criar o objeto deve refletir a definição \_\_init\_\_ ;
* **pode ser utilizada para configurar o objeto**, ou seja, inicializar devidamente o seu estado interno, criar variáveis de instância, instanciar quaisquer outros objetos se a sua existência for necessária, etc.

Veja o código no editor. O exemplo mostra um construtor muito simples a trabalhar.

class Classy:

def \_\_init\_\_(self, value):

self.var = value

obj\_1 = Classy("object")

print(obj\_1.var)

Execute-o. Output do código:

object

**output**

Observe que o construtor:

* **não pode devolver um valor**, uma vez que foi concebido para devolver um objeto recém-criado e nada mais;
* **não pode ser invocado diretamente do objeto ou de dentro da classe** (pode invocar um construtor de qualquer uma das subclasses do objeto, mas discutiremos esta questão mais tarde).

Como \_\_init\_\_ é um método, e um método é uma função, pode-se fazer os mesmos truques com construtores/métodos que se fazem com funções comuns.

O exemplo no editor mostra como definir um construtor com um valor de argumento padrão. Teste-o.

class Classy:

def \_\_init\_\_(self, value = None):

self.var = value

obj\_1 = Classy("object")

obj\_2 = Classy()

print(obj\_1.var)

print(obj\_2.var)

Output do código:

object

None

**output**

Tudo o que dissemos sobre **mangling de nome de propriedade** aplica-se também a nomes de métodos - um método cujo nome começa com \_\_ está (parcialmente) oculto.

O exemplo mostra este efeito:

class Classy:

def visible(self):

print("visible")

def \_\_hidden(self):

print("hidden")

obj = Classy()

obj.visible()

try:

obj.\_\_hidden()

except:

print("failed")

obj.\_Classy\_\_hidden()

Output do código:

visible

failed

hidden

**output**

Execute o programa e teste-o.

# A vida interior das classes e objetos

Cada classe de Python e cada objeto de Python é pré-equipada com um conjunto de atributos úteis que podem ser utilizados para examinar as suas capacidades.

Já conhece um destes - é a propriedade \_\_dict\_\_ .

Vamos observar como lida com os métodos - vejamos o código no editor.

class Classy:

varia = 1

def \_\_init\_\_(self):

self.var = 2

def method(self):

pass

def \_\_hidden(self):

pass

obj = Classy()

print(obj.\_\_dict\_\_)

print(Classy.\_\_dict\_\_)

Execute-o para ver o seu output. Verifique o output com cuidado.

Encontre todos os métodos e atributos definidos. Localize o contexto no qual eles existem: dentro do objeto ou dentro da classe.

\_\_dict\_\_ é um dicionário. Outra propriedade integrada que vale a pena mencionar é \_\_name\_\_, que é uma string.

A propriedade contém **o nome da classe** Não é nada de excitante, apenas uma cadeia.

Nota: o atributo \_\_name\_\_ está ausente do objeto - **só existe dentro das classes**.

Se quiser **encontrar a classe de um determinado objeto**, pode usar uma função chamada type(), que é capaz (entre outras coisas) de encontrar uma classe que tenha sido utilizada para instanciar qualquer objeto.

Olhe para o código no editor, execute-o, e veja por si mesmo.

class Classy:

pass

print(Classy.\_\_name\_\_)

obj = Classy()

print(type(obj).\_\_name\_\_)

Output do código:

Classy

Classy

**output**

Observe que uma declaração como esta:

print(obj.\_\_name\_\_)

irá causar um erro.

\_\_module\_\_ é também uma string - **armazena o nome do módulo que contém a definição da classe**.

Vamos verificá-la - execute o código no editor.

class Classy:

pass

print(Classy.\_\_module\_\_)

obj = Classy()

print(obj.\_\_module\_\_)

Output do código:

\_\_main\_\_

\_\_main\_\_

**output**

Como sabe, qualquer módulo chamado \_\_main\_\_ não é na realidade um módulo, mas o **ficheiro atualmente em execução**.

\_\_bases\_\_ é um tuple. O **tuple contém classes** (não nomes de classe) que são superclasses diretas para a classe.

A ordem é a mesma que é utilizada dentro da definição de classe.

Mostrar-lhe-emos apenas um exemplo muito básico, uma vez que queremos destacar **como funciona a herança**.

Além disso, vamos mostrar-lhe como utilizar este atributo quando discutirmos os aspetos objetivos das exceções.

Nota: **apenas as classes têm este atributo** - os objetos não o têm.

Definimos uma função chamada printbases(), concebida para apresentar claramente o conteúdo do tuple.

Veja o código no editor. Analise e execute-o.

class SuperOne:

pass

class SuperTwo:

pass

class Sub(SuperOne, SuperTwo):

pass

def printBases(cls):

print('( ', end='')

for x in cls.\_\_bases\_\_:

print(x.\_\_name\_\_, end=' ')

print(')')

printBases(SuperOne)

printBases(SuperTwo)

printBases(Sub)

Ele terá como output:

( object )

( object )

( SuperOne SuperTwo )

**output**

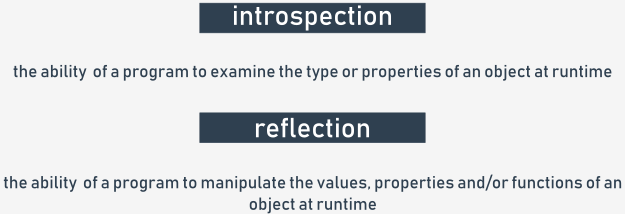
Nota: **uma classe sem superclasses explícitas aponta para o objeto** (uma classe Python predefinida) como seu antepassado direto.

**Reflexão e introspeção**

Todos estes meios permitem ao programador Python realizar duas importantes atividades específicas a muitas linguagens objetivas. São elas:

* **introspeção**, que é a capacidade de um programa para examinar o tipo ou propriedades de um objeto em runtime;
* **reflexão**, que vai um passo além, e é a capacidade de um programa para manipular os valores, propriedades e/ou funções de um objeto em runtime.

Por outras palavras, não é necessário conhecer uma definição completa de classe/objeto para manipular o objeto, pois o objeto e/ou a sua classe contêm os metadados que lhe permitem reconhecer as suas características durante a execução do programa.



**Investigar classes**

O que se pode descobrir sobre as classes em Python? A resposta é simples - tudo.

Tanto a reflexão como a introspeção permitem a um programador fazer qualquer coisa com cada objeto, independentemente da sua proveniência.

Analise o código no editor.

class MyClass:

pass

obj = MyClass()

obj.a = 1

obj.b = 2

obj.i = 3

obj.ireal = 3.5

obj.integer = 4

obj.z = 5

def incIntsI(obj):

for name in obj.\_\_dict\_\_.keys():

if name.startswith('i'):

val = getattr(obj, name)

if isinstance(val, int):

setattr(obj, name, val + 1)

print(obj.\_\_dict\_\_)

incIntsI(obj)

print(obj.\_\_dict\_\_)

A função chamada incIntsI() obtém um objeto de qualquer classe, digitaliza o seu conteúdo a fim de encontrar todos os atributos inteiros com nomes que comecem por i, e incrementa-os por um.

Impossível? De modo algum!

É assim que funciona:

* linha 1: definir uma classe muito simples...
* linhas 3 a 10: ...e preenchê-la com alguns atributos;
* linha 12: esta é a nossa função!
* linha 13: digitalizar o atributo \_\_dict\_\_ , procurando todos os nomes de atributos;
* linha 14: se um nome começar com i...
* linha 15: ...use a função getattr() para obter o seu valor atual; nota: getattr() toma dois argumentos: um objeto, e o seu nome de propriedade (como uma string), e devolve o valor do atributo atual;
* linha 16: verificar se o valor é do tipo inteiro, e utilizar a função isinstance() para este fim (discutiremos isto mais tarde);
* linha 17: se a verificação correr bem, aumente o valor da propriedade fazendo uso da função setattr() ; a função toma três argumentos: um objeto, o nome da propriedade (como uma string) e o novo valor da propriedade.

Output do código:

{'a': 1, 'integer': 4, 'b': 2, 'i': 3, 'z': 5, 'ireal': 3.5}

{'a': 1, 'integer': 5, 'b': 2, 'i': 4, 'z': 5, 'ireal': 3.5}

**output**

E é tudo!

# Key takeaways

1. Um método é uma função incorporada dentro de uma classe. O primeiro (ou único) parâmetro de cada método é normalmente nomeado self, que se destina a identificar o objeto para o qual o método é invocado a fim de aceder às propriedades do objeto ou invocar os seus métodos.

2. Se uma classe contém um **construtor** (um método chamado \_\_init\_\_) não pode devolver qualquer valor e não pode ser invocada diretamente.

3. Todas as classes (mas não objetos) contêm uma propriedade chamada \_\_name\_\_, que armazena o nome da classe. Além disso, uma propriedade chamada \_\_module\_\_ armazena o nome do módulo no qual a classe foi declarada, enquanto a propriedade nomeada \_\_bases\_\_ é um tuple contendo as superclasses de uma classe.

Por exemplo:

class Sample:

def \_\_init\_\_(self):

self.name = Sample.\_\_name\_\_

def myself(self):

print("My name is " + self.name + " living in a " + Sample.\_\_module\_\_)

obj = Sample()

obj.myself()

Output do código:

My name is Sample living in a \_\_main\_\_

**output**

**Exercício 1**

A declaração da classe Snake é dada abaixo. Enriqueça a classe com um método chamado increment(), adicionando 1 à \_\_victims propriedade.  
  
class Snake:

def \_\_init\_\_(self):

self.victims = 0

Verifique

**Exercício 2**

Redefina o construtor de classe Snake , para que tenha um parâmetro para inicializar o campo victims com um valor passado para o objeto durante a construção.

Verifique

**Exercício 3**

Consegue prever o output do seguinte código?

class Snake:

pass

class Python(Snake):

pass

print(Python.\_\_name\_\_, 'is a', Snake.\_\_name\_\_)

print(Python.\_\_bases\_\_[0].\_\_name\_\_, 'can be', Python.\_\_name\_\_)

# LAB A classe Timer

Fácil/Médio

## Objetivos

* melhorar as competências do aluno na definição das classes a partir do zero;
* definir e utilizar variáveis de instância;
* definir e utilizar métodos.

## Cenário

Precisamos de uma classe capaz de contar segundos. Fácil? Não tanto quanto se possa pensar, pois vamos ter algumas expectativas específicas.

Leia-os cuidadosamente, pois a classe sobre a qual está a escrever será usada para lançar foguetes que realizam missões internacionais a Marte. É uma grande responsabilidade. Contamos consigo!

A sua classe será chamada Timer. O seu construtor aceita três argumentos representando **horas** (um valor a partir do intervalo [0...23] - vamos utilizar o tempo militar), **minutos** (a partir do intervalo [0...59]) e **segundos** (a partir do intervalo [0...59]).

Zero é o valor padrão para todos os parâmetros acima. Não há necessidade de efetuar quaisquer verificações de validação.

A própria classe deve fornecer as seguintes facilidades:

* objetos da classe devem ser "imprimíveis", ou seja, devem ser capazes de se converter implicitamente em strings da seguinte forma: "hh:mm:ss", com zeros iniciais adicionados quando qualquer um dos valores for inferior a 10;
* a classe deve ser equipada com métodos sem parâmetros chamados next\_second() e previous\_second(), incrementando o tempo armazenado dentro dos objetos em +1/-1 segundo, respetivamente.

Use as seguintes dicas:

* todas as propriedades do objeto devem ser privadas;
* considere escrever uma função separada (não um método!) para formatar a string de tempo.

Complete o template que lhe fornecemos no editor. Execute o seu código e verifique se o output tem o mesmo aspeto que o nosso.

## Output esperado

23:59:59

00:00:00

23:59:59

## RESPOSTA

def two\_digits(val):

s = str(val)

if len(s) == 1:

s = '0' + s

return s

class Timer:

def \_\_init\_\_(self, hours=0, minutes=0, seconds=0):

self.\_\_hours = hours

self.\_\_minutes = minutes

self.\_\_seconds = seconds

def \_\_str\_\_(self):

return two\_digits(self.\_\_hours) + ":" + \

two\_digits(self.\_\_minutes) + ":" + \

two\_digits(self.\_\_seconds)

def next\_second(self):

self.\_\_seconds += 1

if self.\_\_seconds > 59:

self.\_\_seconds = 0

self.\_\_minutes += 1

if self.\_\_minutes > 59:

self.\_\_minutes = 0

self.\_\_hours += 1

if self.\_\_hours > 23:

self.\_\_hours = 0

def prev\_second(self):

self.\_\_seconds -= 1

if self.\_\_seconds < 0:

self.\_\_seconds = 59

self.\_\_minutes -= 1

if self.\_\_minutes < 0:

self.\_\_minutes = 59

self.\_\_hours -= 1

if self.\_\_hours < 0:

self.\_\_hours = 23

timer = Timer(23, 59, 59)

print(timer)

timer.next\_second()

print(timer)

timer.prev\_second()

print(timer)

# Nível de dificuldade

Fácil/Médio

## Objetivos

* melhorar as competências do aluno na definição das classes a partir do zero;
* definir e utilizar variáveis de instância;
* definir e utilizar métodos.

## Cenário

A sua tarefa é implementar uma classe chamada Weeker. Sim, os seus olhos não o enganam - este nome vem do facto de que os objetos dessa classe serão capazes de armazenar e manipular dias de uma semana.

O construtor de classe aceita um argumento - uma string. A string representa o nome do dia da semana e os únicos valores aceitáveis devem vir do seguinte conjunto:

Mon Thu Wed Thu Fri Sat Sun

Invocar o construtor com um argumento de fora deste conjunto deveria levantar a exceção WeekDayError (defina-a você mesmo; não se preocupe, em breve falaremos sobre a natureza objetiva das exceções). A classe deve fornecer as seguintes facilidades:

* os objetos da classe devem ser "imprimíveis", ou seja, devem ser capazes de se converter implicitamente em strings da mesma forma que os argumentos do construtor;
* a classe deve ser equipada com métodos de um parâmetro chamado add\_days(n) e subtract\_days(n), sendo **n** um número inteiro e atualizando o dia da semana armazenado dentro do objeto de forma a refletir a mudança de data pelo número de dias indicado, para a frente ou para trás.
* todas as propriedades do objeto devem ser privadas;

Complete o modelo que fornecemos no editor e execute o seu código e verifique se o seu output tem o mesmo aspeto que o nosso.

## Output esperado

Mon

Thu

Sun

Sorry, I can't serve your request.

## RESPOSTA

class WeekDayError(Exception):

pass

class Weeker:

\_\_names = ['Mon', 'Thu', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']

def \_\_init\_\_(self, day):

try:

self.\_\_current = Weeker.\_\_names.index(day)

except ValueError:

raise WeekDayError

def \_\_str\_\_(self):

return Weeker.\_\_names[self.\_\_current]

def add\_days(self, n):

self.\_\_current = (self.\_\_current + n) % 7

def subtract\_days(self, n):

self.\_\_current = (self.\_\_current - n) % 7

try:

weekday = Weeker('Mon')

print(weekday)

weekday.add\_days(15)

print(weekday)

weekday.subtract\_days(23)

print(weekday)

weekday = Weeker('Monday')

except WeekDayError:

print("Sorry, I can't serve your request.")

# LAB Pontos num plano

Fácil/Médio

## Objetivos

* melhorar as competências do aluno na definição das classes a partir do zero;
* definir e utilizar variáveis de instância;
* definir e utilizar métodos.

## Cenário

Vamos visitar um lugar muito especial - um plano com o sistema de coordenadas cartesianas (pode saber mais sobre este conceito aqui: <https://en.wikipedia.org/wiki/Cartesian_coordinate_system>).

Cada ponto localizado no plano pode ser descrito como um par de coordenadas habitualmente chamadas **x** e **y**. Esperamos que seja capaz de escrever uma classe Python que armazena ambas as coordenadas como números float. Além disso, queremos que os objetos desta classe avaliem as distâncias entre qualquer um dos dois pontos situados no plano.

A tarefa é bastante fácil se se empregar a função denominada hypot() (disponível através do módulo *math*) que avalia o comprimento da hipotenusa de um triângulo retângulo (mais detalhes aqui: <https://en.wikipedia.org/wiki/Hypotenuse>) e aqui: <https://docs.python.org/3.7/library/math.html#trigonometric-functions>.

É assim que imaginamos a classe:

* é chamada Point;
* o seu construtor aceita dois argumentos (**x** e **y** respetivamente), ambos a zero, por defeito;
* todas as propriedades devem ser privadas;
* a classe contém dois métodos sem parâmetros chamados getx() e gety(), que devolvem cada uma das duas coordenadas (as coordenadas são armazenadas de forma privada, pelo que não podem ser acedidas diretamente do interior do objeto);
* a classe fornece um método chamado distance\_from\_xy(x,y), que calcula e devolve a distância entre o ponto armazenado dentro do objeto e o outro ponto dado como um par de floats;
* a classe fornece um método chamado distance\_from\_point(point), que calcula a distância (como o método anterior), mas a localização do outro ponto é dada como outro Ponto objeto de classe;

Complete o modelo que fornecemos no editor e execute o seu código e verifique se o seu output tem o mesmo aspeto que o nosso.

## Output esperado

1.4142135623730951

1.4142135623730951

## RESPOSTA

import math

class Point:

def \_\_init\_\_(self, x=0.0, y=0.0):

self.\_\_x = x

self.\_\_y = y

def getx(self):

return self.\_\_x

def gety(self):

return self.\_\_y

def distance\_from\_xy(self, x, y):

return math.hypot(abs(self.\_\_x - x), abs(self.\_\_y - y))

def distance\_from\_point(self, point):

return self.distance\_from\_xy(point.getx(), point.gety())

point1 = Point(0, 0)

point2 = Point(1, 1)

print(point1.distance\_from\_point(point2))

print(point2.distance\_from\_xy(2, 0))

# LAB Triângulo

Médio

## Objetivos

* melhorar as competências do aluno na definição das classes a partir do zero;
* utilizar a composição.

## Cenário

Vamos agora incorporar a classe Point (ver Lab 3.4.1.14) dentro de outra classe. Além disto, vamos colocar três pontos numa classe, o que nos permitirá definir um triângulo. Como podemos fazer isto?

A nova classe será chamada Triangle e esta é a lista das nossas expetativas:

* o construtor aceita três argumentos - todos eles são objetos da classe Point ;
* os pontos são armazenados dentro do objeto como uma lista privada;
* a classe fornece um método sem parâmetros chamado perimeter(), que calcula o perímetro do triângulo descrito pelos três pontos; o perímetro é uma soma de todos os comprimentos das pernas (mencionamo-lo para registo, embora estejamos certos de que o conhece perfeitamente.)

Complete o template que fornecemos no editor. Execute o seu código e verifique se o perímetro avaliado é igual ao nosso.

Em baixo pode copiar o código da classe Point que usámos no lab anterior:

Verifique

## Output esperado

3.414213562373095

## RESPOSTA

import math

class Point:

def \_\_init\_\_(self, x=0.0, y=0.0):

self.\_\_x = x

self.\_\_y = y

def getx(self):

return self.\_\_x

def gety(self):

return self.\_\_y

def distance\_from\_xy(self, x, y):

return math.hypot(abs(self.\_\_x - x), abs(self.\_\_y - y))

def distance\_from\_point(self, point):

return self.distance\_from\_xy(point.getx(), point.gety())

class Triangle:

def \_\_init\_\_(self, vertice1, vertice2, vertice3):

self.\_\_vertices = [vertice1, vertice2, vertice3]

def perimeter(self):

per = 0

for i in range(3):

per += self.\_\_vertices[i].distance\_from\_point(self.\_\_vertices[(i + 1) % 3])

return per

triangle = Triangle(Point(0, 0), Point(1, 0), Point(0, 1))

print(triangle.perimeter())

# Herança - porquê e como?

Antes de começarmos a falar sobre heranças, queremos apresentar um novo e prático mecanismo utilizado pelas classes e objetos Python - **a forma como o objeto é capaz de se introduzir a si próprio**.

Vamos começar com um exemplo. Veja o código no editor.

class Star:

def \_\_init\_\_(self, name, galaxy):

self.name = name

self.galaxy = galaxy

sun = Star("Sun", "Milky Way")

print(sun)

O programa imprime apenas uma linha de texto, que no nosso caso é a seguinte:

<\_\_main\_\_.Star object at 0x7f1074cc7c50>

**output**

Se executar o mesmo código no seu computador, verá algo muito semelhante, embora o número hexadecimal (a substring a começar por 0x) será diferente, uma vez que é apenas um identificador de objeto interno utilizado pelo Python, e é improvável que pareça o mesmo quando o mesmo código é executado num ambiente diferente.

Como pode ver, a impressão aqui não é realmente útil, e algo mais específico, ou apenas mais bonito, pode ser mais preferível.

Felizmente, o Python oferece precisamente tal função.

Quando o Python precisa que qualquer classe/objeto seja apresentado como uma string (colocar um objeto como argumento na invocação de função print() encaixa-se nesta condição) ele tenta invocar um método chamado \_\_str\_\_() do objeto e usar a string que ele devolve.

O método por defeito \_\_str\_\_() devolve a string anterior - feia e não muito informativa. Pode alterá-lo apenas **definindo o seu próprio método do nome**.

Acabámos de fazer isso - veja o código no editor.

class Star:

def \_\_init\_\_(self, name, galaxy):

self.name = name

self.galaxy = galaxy

def \_\_str\_\_(self):

return self.name + ' in ' + self.galaxy

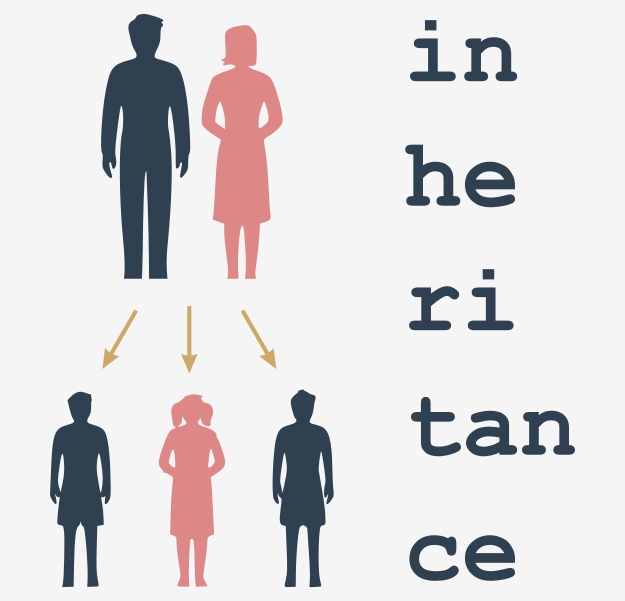
sun = Star("Sun", "Milky Way")

print(sun)

Este novo método \_\_str\_\_() faz uma string consistindo nos nomes da estrela e da galáxia - nada de especial, mas os resultados da impressão parecem melhores agora, certo?

Consegue adivinhar o output? Execute o código para verificar se estava certo.

O termo herança (inheritance) é mais antigo do que a programação de computadores, e descreve a prática comum de passar vários bens de uma pessoa para outra após a morte da primeira. O termo, quando relacionado à programação de computadores, tem um significado completamente diferente.



Vamos definir o termo para os nossos propósitos:

A herança é uma prática comum (na programação de objetos) de **passar atributos e métodos da superclasse (definida e existente) para uma classe recém-criada, chamada subclasse**.

Por outras palavras, a herança é **uma forma de construir uma nova classe, não a partir do zero, mas usando um repertório de traços já definidos**. A nova classe herda (e esta é a chave) todo o equipamento já existente, mas é capaz de adicionar alguns novos se necessário.

Graças a isso, é possível **construir classes mais especializadas (mais concretas)** usando alguns conjuntos de regras e comportamentos gerais predefinidos.

O fator mais importante do processo é a relação entre a superclasse e todas as suas subclasses (nota: se *B* é uma subclasse de *A* e *C* é uma subclasse de *B*, isto também significa que *C* é uma subclasse de *A*, uma vez que a relação é totalmente transitória).

Um exemplo muito simples de herança de **dois níveis** é apresentado aqui:

class Vehicle:

pass

class LandVehicle(Vehicle):

pass

class TrackedVehicle(LandVehicle):

pass

Todas as classes apresentadas estão vazias por agora, pois vamos mostrar-lhe como funcionam as relações mútuas entre as super e as subclasses. Em breve iremos preenchê-las com conteúdos.

Podemos dizer que:

* A classe Vehicle é a superclasse para ambas as classes LandVehicle e TrackedVehicle ;
* A classe LandVehicle é uma subclasse de Vehicle e uma superclasse de TrackedVehicle ao mesmo tempo;
* A classe TrackedVehicle é uma subclasse de ambas as classes Vehicle e LandVehicle .

O conhecimento acima vem da leitura do código (por outras palavras, nós conhecemo-lo porque o podemos ver).

O Python sabe o mesmo? É possível perguntar ao Python sobre isso? Sim, é.

**Herança: issubclass()**

O Python oferece uma função que é capaz de **identificar uma relação entre duas classes**, e embora o seu diagnóstico não seja complexo, pode **verificar se uma determinada classe é uma subclasse de qualquer outra classe**.

É este o seu aspeto:

issubclass(ClassOne, ClassTwo)

A função devolve True Se ClassOne for uma subclasse de ClassTwo, e False caso contrário.

Vamos vê-lo em ação - pode surpreendê-lo. Veja o código no editor. Leia-o com atenção.

class Vehicle:

pass

class LandVehicle(Vehicle):

pass

class TrackedVehicle(LandVehicle):

pass

for cls1 in [Vehicle, LandVehicle, TrackedVehicle]:

for cls2 in [Vehicle, LandVehicle, TrackedVehicle]:

print(issubclass(cls1, cls2), end="\t")

print()

Existem dois loops nested. O seu objetivo é **verificar todos os pares de classes ordenados possíveis, e imprimir os resultados da verificação para determinar se o par corresponde à relação subclasse-superclasse**.

Execute o código. O programa produz o seguinte output:

True False False

True True False

True True True

**output**

Vamos tornar o resultado mais legível:

| **↓ é uma subclasse de →** | **Vehicle** | **LandVehicle** | **TrackedVehicle** |
| --- | --- | --- | --- |
| **Vehicle** | True | False | False |
| **LandVehicle** | True | True | False |
| **TrackedVehicle** | True | True | True |

Há uma observação importante a fazer: **cada classe é considerada como uma subclasse de si mesma**.

**Herança: isinstance()**

Como já sabe, **um objeto é uma incarnação de uma classe**. Isto significa que o objeto é como um bolo cozido utilizando uma receita que está incluída dentro da classe.

Isto pode gerar alguns problemas importantes.

Vamos supor que tem um bolo (por exemplo, como um argumento passado para a sua função). Você quer saber qual a receita utilizada para o fazer. Porquê? Porque quer saber o que esperar dele, por exemplo, se contém nozes ou não, o que é uma informação crucial para algumas pessoas.

Da mesma forma, pode ser crucial se o objeto tiver (ou não tiver) certas características. Por outras palavras, **se é um objeto de uma determinada classe ou não**.

Tal facto pode ser detetado pela função chamada isinstance():

isinstance(objectName, ClassName)

As funções devolvem True se o objeto for uma instância da classe, ou False caso contrário.

**Ser uma instância de uma classe significa que o objeto (o bolo) foi preparado utilizando uma receita contida quer na classe quer numa das suas superclasses**.

Não se esqueça: se uma subclasse contém pelo menos o mesmo equipamento que qualquer uma das suas superclasses, significa que os objetos da subclasse podem fazer o mesmo que os objetos derivados da superclasse, portanto, é uma instância da sua home class e de qualquer uma das suas superclasses.

Vamos testá-lo. Analise o código no editor.

class Vehicle:

pass

class LandVehicle(Vehicle):

pass

class TrackedVehicle(LandVehicle):

pass

my\_vehicle = Vehicle()

my\_land\_vehicle = LandVehicle()

my\_tracked\_vehicle = TrackedVehicle()

for obj in [my\_vehicle, my\_land\_vehicle, my\_tracked\_vehicle]:

for cls in [Vehicle, LandVehicle, TrackedVehicle]:

print(isinstance(obj, cls), end="\t")

print()

Criámos três objetos, um para cada uma das classes. Em seguida, utilizando dois loops nested, verificamos todos os pares possíveis da classe de objeto **para descobrir se os objetos são instâncias das classes**.

Execute o código.

Isto é o que obtemos:

True False False

True True False

True True True

**output**

Vamos novamente tornar o resultado mais legível:

| **↓ é uma instância de →** | **Vehicle** | **LandVehicle** | **TrackedVehicle** |
| --- | --- | --- | --- |
| **my\_vehicle** | True | False | False |
| **my\_land\_vehicle** | True | True | False |
| **my\_tracked\_vehicle** | True | True | True |

Será que a tabela confirma as nossas expetativas?

# Herança: o operador is .

Há também um operador Python que vale a pena mencionar, pois refere-se diretamente a objetos - aqui está ele:

object\_one is object\_two

**O operador**is**verifica se duas variáveis (**object\_one**e**object\_two**aqui) se referem ao mesmo objeto**.

Não se esqueça que **as variáveis não armazenam os objetos em si, mas apenas os manípulos que apontam para a memória interna do Python**.

Atribuir um valor de uma variável de objeto a outra variável não copia o objeto, mas apenas o seu manípulo. É por isso que um operador como is pode ser muito útil em circunstâncias particulares.

Dê uma vista de olhos ao código no editor.

class SampleClass:

def \_\_init\_\_(self, val):

self.val = val

object\_1 = SampleClass(0)

object\_2 = SampleClass(2)

object\_3 = object\_1

object\_3.val += 1

print(object\_1 is object\_2)

print(object\_2 is object\_3)

print(object\_3 is object\_1)

print(object\_1.val, object\_2.val, object\_3.val)

string\_1 = "Mary had a little "

string\_2 = "Mary had a little lamb"

string\_1 += "lamb"

print(string\_1 == string\_2, string\_1 is string\_2)

Vamos analisá-lo:

* existe uma classe muito simples equipada com um construtor simples, criando apenas uma propriedade. A classe é utilizada para instanciar dois objetos. O primeiro é então atribuído a outra variável, e a sua propriedade val é incrementada por um.
* depois, o operador is é aplicado três vezes para verificar todos os pares de objetos possíveis, e todos os valores de propriedade val também são impressos.
* a última parte do código realiza outra experiência. Após três atribuições, ambas as cadeias contêm os mesmos textos, mas **estes textos são armazenados em objetos diferentes**.

O código imprime:

False

False

True

1 2 1

True False

**output**

Os resultados demonstram que object\_1 e object\_3 são na verdade os mesmos objetos, enquanto string\_1 e string\_2 não são, apesar de seu conteúdo ser o mesmo.

# De que forma o Python encontra propriedades e métodos

Vamos agora ver como o Python lida com métodos de herança.

Dê uma vista de olhos no exemplo no editor.

class Super:

def \_\_init\_\_(self, name):

self.name = name

def \_\_str\_\_(self):

return "My name is " + self.name + "."

class Sub(Super):

def \_\_init\_\_(self, name):

Super.\_\_init\_\_(self, name)

obj = Sub("Andy")

print(obj)

Vamos analisá-lo:

* há uma classe chamada Super, que define o seu próprio construtor, usado para atribuir a propriedade do objeto, chamada name.
* a classe também define o método \_\_str\_\_() , o que torna a classe capaz de apresentar a sua identidade na forma de um texto claro.
* a classe é então utilizada como base para criar uma subclasse chamada Sub. A classe Sub define o seu próprio construtor, que invoca o da superclasse. Observe como o fizemos: Super.\_\_init\_\_(self, name).
* nomeámos explicitamente a superclasse, e apontámos para o método para invocar \_\_init\_\_(), fornecendo todos os argumentos necessários.
* instanciámos um objeto de classe Sub e imprimimo-lo.

Output do código:

My name is Andy.

**output**

Nota: Como não há um método \_\_str\_\_() dentro da classe Sub , a string impressa deve ser produzida dentro da classe Super . Isto significa que o método \_\_str\_\_() foi herdado pela classe Sub .

Veja o código no editor.

class Super:

def \_\_init\_\_(self, name):

self.name = name

def \_\_str\_\_(self):

return "My name is " + self.name + "."

class Sub(Super):

def \_\_init\_\_(self, name):

super().\_\_init\_\_(name)

obj = Sub("Andy")

print(obj)

Modificámo-lo para lhe mostrar outro método de acesso a qualquer entidade definida dentro da superclasse.

No último exemplo, nomeámos explicitamente a superclasse. Neste exemplo, fazemos uso da função super() , que **acede à superclasse sem a necessidade de saber o seu nome**:

super().\_\_init\_\_(name)

A função super() cria um contexto em que não é necessário (além disso, não se deve) passar o argumento self ao método invocado - é por isso que é possível ativar o construtor da superclasse usando apenas um argumento.

Nota: pode utilizar este mecanismo não só **para invocar o construtor da superclasse, mas também para ter acesso a qualquer um dos recursos disponíveis dentro da superclasse**.

Vamos tentar fazer algo semelhante, mas com propriedades (mais precisamente: com **variáveis de classe**).

Dê uma vista de olhos no exemplo no editor.

class Super:

supVar = 1

class Sub(Super):

subVar = 2

obj = Sub()

print(obj.subVar)

print(obj.supVar)

Como pode ver, a classe Super define uma variável de classe chamada supVar, e a classe Sub define uma variável chamada subVar.

Ambas as variáveis são visíveis dentro do objeto da classe Sub - é por isso que o código faz o output:

2

1

O mesmo efeito pode ser observado com **variáveis de instância** - veja o segundo exemplo no editor.

class Super:

def \_\_init\_\_(self):

self.supVar = 11

class Sub(Super):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.subVar = 12

obj = Sub()

print(obj.subVar)

print(obj.supVar)

O construtor de classe Sub cria uma variável de instância chamada subVar, enquanto o construtor Super faz o mesmo com uma variável chamada supVar. Como anteriormente, ambas as variáveis são acessíveis de dentro do objeto da classe Sub.

O output do programa é:

12

11

**output**

Nota: a existência da variável supVar é obviamente condicionada pela invocação do construtor de classe Super . Omiti-la resultaria na ausência da variável no objeto criado (experimente-o você mesmo).

Agora é possível formular uma declaração geral descrevendo o comportamento do Python.

Quando se tenta aceder à entidade de qualquer objeto, o Python tentará (por esta ordem):

* encontrá-lo **dentro do próprio objeto**;
* encontrá-lo **em todas as classes** envolvidas na linha de herança do objeto, de baixo para cima;

Se ambos os itens acima falharem, uma **exceção (**AttributeError**) é levantada**.

A primeira condição pode precisar de alguma atenção adicional. Como sabe, todos os objetos derivados de uma determinada classe podem ter diferentes conjuntos de atributos, e alguns dos atributos podem ser acrescentados ao objeto muito tempo após a sua criação.

O exemplo no editor resume isto numa **linha de herança de três níveis**. Analise-o cuidadosamente.

class Level1:

variable\_1 = 100

def \_\_init\_\_(self):

self.var\_1 = 101

def fun\_1(self):

return 102

class Level2(Level1):

variable\_2 = 200

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.var\_2 = 201

def fun\_2(self):

return 202

class Level3(Level2):

variable\_3 = 300

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.var\_3 = 301

def fun\_3(self):

return 302

obj = Level3()

print(obj.variable\_1, obj.var\_1, obj.fun\_1())

print(obj.variable\_2, obj.var\_2, obj.fun\_2())

print(obj.variable\_3, obj.var\_3, obj.fun\_3())

Todos os comentários que fizemos até agora estão relacionados com **herança única**, quando uma subclasse tem exatamente uma superclasse. Esta é a situação mais comum (e a mais recomendada também).

O Python, no entanto, oferece muito mais aqui. Nas próximas lições, vamos mostrar-lhe alguns exemplos de **herança múltipla**.

**A herança múltipla ocorre quando uma classe tem mais de uma superclasse**. Sintaticamente, tal herança é apresentada como uma lista separada por vírgulas de superclasses colocadas dentro de parêntesis, após o novo nome da classe - tal como aqui:

class SuperA:

var\_a = 10

def fun\_a(self):

return 11

class SuperB:

var\_b = 20

def fun\_b(self):

return 21

class Sub(SuperA, SuperB):

pass

obj = Sub()

print(obj.var\_a, obj.fun\_a())

print(obj.var\_b, obj.fun\_b())

A classe Sub tem duas superclasses: SuperA e SuperB. Isto significa que a classe Sub **herda todos os bens oferecidos por ambos**SuperA**e**SuperB.

O código imprime:

10 11

20 21

**output**

Agora é tempo de introduzir um termo completamente novo - **overriding**.

O que pensa que acontecerá se mais do que uma das superclasses definir uma entidade com um determinado nome?

Vamos analisar o exemplo no editor.

class Level1:

var = 100

def fun(self):

return 101

class Level2(Level1):

var = 200

def fun(self):

return 201

class Level3(Level2):

pass

obj = Level3()

print(obj.var, obj.fun())

Ambas as classes Level1 e Level2 definem um método chamado fun() e uma propriedade chamada var. Isto significa que o objecto de classe Level3 poderá ter acesso a duas cópias de cada entidade? De modo algum.

**A entidade definida mais tarde (no sentido da herança) sobrepõe-se à mesma entidade definida mais cedo**. É por isso que o código produz o seguinte output:

200 201

**output**

Como pode ver, o argumento de keyword var e método fun() a partir da classe Level2 substituem as entidades com os mesmos nomes derivados da classe Level1 .

Esta característica pode ser intencionalmente utilizada para modificar comportamentos padrão (ou previamente definidos) de classe quando qualquer uma das suas classes precisa de agir de uma forma diferente da dos seus antepassados.

Podemos também dizer que **o Python procura uma entidade de baixo para cima**, e está plenamente satisfeito com a primeira entidade do nome desejado.

Como funciona quando uma classe tem dois antepassados que oferecem a mesma entidade, e eles estão no mesmo nível? Por outras palavras, o que se deve esperar quando uma classe emerge usando herança múltipla? Vamos ver isto.

Vejamos o exemplo no editor.

class Left:

var = "L"

var\_left = "LL"

def fun(self):

return "Left"

class Right:

var = "R"

var\_right = "RR"

def fun(self):

return "Right"

class Sub(Left, Right):

pass

obj = Sub()

print(obj.var, obj.var\_left, obj.var\_right, obj.fun())

A classe Sub herda bens de duas superclasses, Left e Right (estes nomes destinam-se a ser significativos).

Não há dúvida de que a variável de classe var\_right vem da classe Right , e var\_left vem de Left respetivamente.

Isto é evidente. Mas de onde var vem? É possível adivinhá-lo? O mesmo problema é encontrado com o método fun() - será invocado a partir de Left ou a partir de Right? Vamos executar o programa - o seu output é:

L LL RR Left

**output**

Isto prova que ambos os casos pouco claros têm uma solução dentro da Left classe. Será esta uma premissa suficiente para formular uma regra geral? Sim, é.

Podemos dizer que **o Python procura componentes de objeto** na seguinte ordem:

* **dentro do próprio objeto**;
* **nas suas superclasses**, de baixo para cima;
* se houver mais do que uma classe num determinado caminho de herança, o Python analisa-os da esquerda para a direita.

Precisa de mais alguma coisa? Basta fazer uma pequena emenda no código - substituir: class Sub(Left, Right): por: class Sub(Right, Left):, em seguida, execute o programa novamente e veja o que acontece.

O que vê agora? Nós vemos:

R LL RR Right

**output**

Vê o mesmo, ou algo diferente?

# Como construir uma hierarquia de classes

A construção de uma hierarquia de classes não é apenas arte por causa da arte.

Se dividir um problema entre classes e decidir qual delas deve ser localizada no topo e qual deve ser colocada na base da hierarquia, tem de analisar cuidadosamente a questão, mas antes de lhe mostrarmos como o fazer (e como não o fazer), queremos destacar um efeito interessante. Não é nada de extraordinário (é apenas uma consequência das regras gerais apresentadas anteriormente), mas lembrá-la pode ser fundamental para compreender como funcionam alguns códigos, e como o efeito pode ser utilizado para construir um conjunto flexível de classes.

Dê uma vista de olhos ao código no editor.

class One:

def do\_it(self):

print("do\_it from One")

def doanything(self):

self.do\_it()

class Two(One):

def do\_it(self):

print("do\_it from Two")

one = One()

two = Two()

one.doanything()

two.doanything()

Vamos analisá-lo:

* existem duas classes, nomeadas One e Two, enquanto Two é derivada de One. Nada de especial. No entanto, uma coisa parece notável - o método do\_it() .
* o método do\_it()é **definido duas vezes**: originalmente dentro One e subsequentemente dentro Two. A essência do exemplo reside no facto de ser **invocado apenas uma vez** - no interior One.

A questão é - qual dos dois métodos será invocado pelas duas últimas linhas do código?

A primeira invocação parece ser simples, e é simples, na verdade - invocando doanything() a partir do objeto chamado one irá obviamente ativar o primeiro dos métodos.

A segunda invocação precisa de alguma atenção. Também é simples, se tivermos em mente como o Python encontra componentes de classe. A segunda invocação irá lançar do\_it() na forma existente dentro da classe Two , independentemente do facto de a invocação ter lugar dentro da classe One .

Com efeito, o código produz o seguinte output:

do\_it from One

do\_it from Two

**output**

Nota: a situação em que **a subclasse é capaz de modificar o seu comportamento de superclasse (tal como no exemplo) é chamada polimorfismo**. A palavra vem do grego (polys: "muitas" e morphe, "forma"), o que significa que uma e a mesma classe pode tomar várias formas dependendo das redefinições feitas por qualquer uma das suas subclasses.

O método, redefinido em qualquer uma das superclasses, alterando assim o comportamento da superclasse, é chamado **virtual**.

Por outras palavras, nenhuma classe é dada de uma vez por todas. O comportamento de cada classe pode ser modificado em qualquer altura por qualquer uma das suas subclasses.

Vamos mostrar-lhe **como usar o polimorfismo para alargar a flexibilidade de classe**.

Veja o exemplo no editor.

import time

class TrackedVehicle:

def control\_track(left, stop):

pass

def turn(left):

control\_track(left, True)

time.sleep(0.25)

control\_track(left, False)

class WheeledVehicle:

def turn\_front\_wheels(left, on):

pass

def turn(left):

turn\_front\_wheels(left, True)

time.sleep(0.25)

turn\_front\_wheels(left, False)

Parece-se com alguma coisa? Sim, claro que sim. Refere-se ao exemplo mostrado no início do módulo quando falamos sobre os conceitos gerais da programação objetiva.

Pode parecer estranho, mas não utilizámos a herança de forma alguma - apenas para mostrar que ela não nos limita, e conseguimos obter a nossa.

Definimos duas classes distintas capazes de produzir dois tipos diferentes de veículos terrestres. A principal diferença entre eles está na forma como viram. Um veículo com rodas apenas vira as rodas da frente (geralmente). Um veículo com lagartas tem de parar uma das lagartas.

Consegue seguir o código?

* um veículo com lagartas faz uma curva ao parar e deslocar-se numa das suas lagartas (isto é feito pelo método control\_track() , que será implementado mais tarde)
* um veículo com rodas gira quando as suas rodas da frente giram (isto é feito pelo método turn\_front\_wheels() )
* o método turn() utiliza o método adequado para cada veículo em particular.

Consegue ver **o que está errado com o código**?

Os métodos turn() parecem demasiado semelhantes para os deixar nesta forma.

Vamos reconstruir o código - vamos introduzir uma superclasse para reunir todos os aspetos semelhantes dos veículos de condução, deslocando todos os aspetos específicos para as subclasses.

Veja o código no editor novamente.

import time

class Vehicle:

def change\_direction(left, on):

pass

def turn(left):

change\_direction(left, True)

time.sleep(0.25)

change\_direction(left, False)

class TrackedVehicle(Vehicle):

def control\_track(left, stop):

pass

def change\_direction(left, on):

control\_track(left, on)

class WheeledVehicle(Vehicle):

def turn\_front\_wheels(left, on):

pass

def change\_direction(left, on):

turn\_front\_wheels(left, on)

Isto é o que fizemos:

* definimos uma superclasse chamada Vehicle, que usa o método turn() para implementar um esquema geral de viragem, enquanto a viragem propriamente dita é feita por um método denominado change\_direction(); nota: o método anterior está vazio, pois vamos colocar todos os detalhes na subclasse (tal método é muitas vezes chamado **método abstrato**, pois demonstra apenas alguma possibilidade que será instanciada mais tarde)
* definimos uma subclasse chamada TrackedVehicle (nota: é derivada da classe Vehicle ) que instanciou o método change\_direction() utilizando o método específico (concreto) denominado control\_track()
* respetivamente, a subclasse nomeada WheeledVehicle faz o mesmo truque, mas usa o método turn\_front\_wheels() para forçar o veículo a virar.

A vantagem mais importante (omitindo questões de legibilidade) é que esta forma de código permite implementar um algoritmo de viragem novinho em folha, apenas modificando o método turn() , que pode ser feito num só local, uma vez que todos os veículos o obedecerão.

É desta forma que o **polimorfismo ajuda o programador a manter o código limpo e consistente**.

A herança não é a única forma de construir classes adaptáveis. Pode-se alcançar os mesmos objetivos (nem sempre, mas muito frequentemente) usando uma técnica denominada composição.

**Composição é o processo de compor um objeto utilizando outros objetos diferentes**. Os objetos utilizados na composição fornecem um conjunto de características desejadas (propriedades e/ou métodos) para que possamos dizer que agem como blocos utilizados para construir uma estrutura mais complicada.

Pode-se dizer que:

* a **herança alarga as capacidades de uma classe**, acrescentando novos componentes e modificando os existentes; por outras palavras, a receita completa está contida dentro da própria classe e de todos os seus antepassados; o objeto toma todos os pertences da classe e faz uso deles;
* a **composição projeta uma classe como um recipiente** capaz de armazenar e utilizar outros objetos (derivados de outras classes) onde cada um dos objetos implementa uma parte do comportamento de uma classe desejada.

Vamos ilustrar a diferença utilizando os veículos previamente definidos. A abordagem anterior levou-nos a uma hierarquia de classes em que a classe mais alta estava ciente das regras gerais utilizadas na viragem do veículo, mas não sabia como controlar os componentes apropriados (rodas ou lagartas).

As subclasses implementaram esta capacidade através da introdução de mecanismos especializados. Vamos fazer (quase) a mesma coisa, mas usando a composição. A classe - como no exemplo anterior - está ciente de como virar o veículo, mas a viragem real é feita por um objeto especializado armazenado numa propriedade chamada controller. A classe controller é capaz de controlar o veículo, manipulando as peças relevantes do veículo.

Dê uma vista de olhos ao editor - é assim que poderia parecer.

import time

class Tracks:

def change\_direction(self, left, on):

print("tracks: ", left, on)

class Wheels:

def change\_direction(self, left, on):

print("wheels: ", left, on)

class Vehicle:

def \_\_init\_\_(self, controller):

self.controller = controller

def turn(self, left):

self.controller.change\_direction(left, True)

time.sleep(0.25)

self.controller.change\_direction(left, False)

wheeled = Vehicle(Wheels())

tracked = Vehicle(Tracks())

wheeled.turn(True)

tracked.turn(False)

Existem duas classes nomeadas Tracks e Wheels - elas sabem controlar a direção do veículo. Há também uma classe chamada Vehicle que pode utilizar qualquer um dos controladores disponíveis (os dois já definidos, ou qualquer outro definido no futuro) - a controller é ela própria passada para a classe durante a iniciação.

Desta forma, a capacidade de viragem do veículo é composta utilizando um objeto externo, não implementado no interior da Vehicle classe.

Por outras palavras, temos um veículo universal e podemos instalar nele tanto lagartas como rodas.

O código produz o seguinte output:

wheels: True True

wheels: True False

tracks: False True

tracks: False False

# Herança única vs. herança múltipla

Como já sabe, não existem obstáculos à utilização de herança múltipla em Python. Pode derivar qualquer nova classe a partir de mais de uma classe previamente definida.

Existe apenas um "mas". O facto de o poder fazer não significa que tenha de o fazer.

Não se esqueça disso:

* uma única classe de herança é sempre mais simples, mais segura e mais fácil de compreender e manter;

* a herança múltipla é sempre arriscada, pois tem muito mais oportunidades de cometer um erro na identificação destas partes das superclasses, que irão influenciar efetivamente a nova classe;

* herança múltipla pode tornar o overriding extremamente complicado; além disso, a utilização da função super() torna-se ambígua;

* a herança múltipla viola o **princípio da responsabilidade única** (mais detalhes aqui: <https://en.wikipedia.org/wiki/Single_responsibility_principle>) uma vez que faz uma nova classe de duas (ou mais) classes que nada sabem uma sobre a outra;

* sugerimos fortemente a herança múltipla como a última de todas as soluções possíveis - se precisar realmente das muitas funcionalidades diferentes oferecidas pelas diferentes classes, a composição pode ser uma alternativa melhor.

**O que é o Method Resolution Order (MRO) e porque é que nem todas as heranças fazem sentido?**

MRO, em geral, é uma forma (pode-se chamar-lhe uma **estratégia**) em que uma determinada linguagem de programação analisa a parte superior da hierarquia de uma classe, a fim de encontrar o método de que ela necessita atualmente. Vale a pena salientar que línguas diferentes utilizam ligeiramente (ou mesmo completamente) diferentes MROs. O Python é uma criatura única a este respeito, contudo, e os seus costumes são um pouco específicos.

Vamos mostrar-lhe como funciona o MRO de Python em dois casos peculiares, que são exemplos claros de problemas que podem ocorrer quando se tenta usar a herança múltipla de forma demasiado imprudente. Vamos começar com um snippet que inicialmente pode parecer simples. Veja o que preparámos para si no editor.

class Top:

def m\_top(self):

print("top")

class Middle(Top):

def m\_middle(self):

print("middle")

class Bottom(Middle):

def m\_bottom(self):

print("bottom")

object = Bottom()

object.m\_bottom()

object.m\_middle()

object.m\_top()

Temos a certeza de que, se analisar o snippet você mesmo, não verá quaisquer anomalias no mesmo. Sim, tem toda a razão - parece claro e simples, e não suscita preocupações. Se executar o código, este produzirá o seguinte output previsível:

bottom

middle

top

**output**

Sem surpresas até agora. Vamos fazer uma pequena alteração neste código. Veja:

class Top:

def m\_top(self):

print("top")

class Middle(Top):

def m\_middle(self):

print("middle")

class Bottom(Middle, Top):

def m\_bottom(self):

print("bottom")

object = Bottom()

object.m\_bottom()

object.m\_middle()

object.m\_top()

Consegue ver a diferença? Está escondida nesta linha:

class Bottom(Middle, Top):

Desta forma exótica, transformamos um código muito simples com um claro caminho de uma única herança num misterioso enigma de múltiplas heranças. “É válido?” pode-se perguntar. Sim, é. "Como é isso possível?" deve perguntar agora, e esperamos que sinta realmente a necessidade de fazer esta pergunta.

Como pode ver, a ordem em que as duas superclasses foram listadas entre parêntesis está em conformidade com a estrutura do código: a classe Middle precede a classe Top , assim como no caminho da herança real.

Apesar da sua estranheza, a amostra está correta e funciona como esperado, mas é preciso afirmar que esta notação não traz nenhuma nova funcionalidade ou significado adicional.

Vamos modificar o código mais uma vez - agora vamos trocar os dois nomes de superclasse na definição de classe Bottom . Este é o aspeto que o snippet tem agora:

class Top:

def m\_top(self):

print("top")

class Middle(Top):

def m\_middle(self):

print("middle")

class Bottom(Top, Middle):

def m\_bottom(self):

print("bottom")

object = Bottom()

object.m\_bottom()

object.m\_middle()

object.m\_top()

Para antecipar a sua pergunta, diremos que esta emenda estragou o código, e que já não funcionará mais. Que pena. A ordem que tentamos forçar (Top, Middle) é incompatível com o caminho da herança que é derivado da estrutura do código. O Python não vai gostar. Isto é o que veremos:

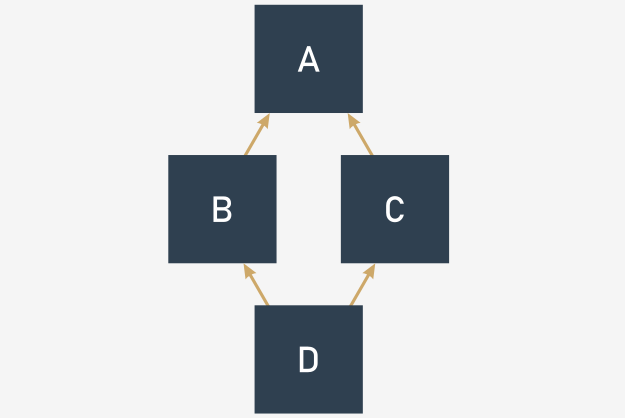
TypeError: Cannot create a consistent method resolution order (MRO) for bases Top, Middle

**output**

Achamos que a mensagem fala por si. O MRO de Python não pode ser distorcido ou violado, não só porque é assim que o Python funciona, mas também porque é uma regra a que se tem de obedecer.

# O problema do diamante

O segundo exemplo do espectro de questões que podem eventualmente surgir de heranças múltiplas é ilustrado por um problema clássico chamado o **problema do diamante**. O nome reflete a forma do diagrama da herança - dê uma vista de olhos na imagem:



* Existe a superclasse mais alta chamada A;
* Existem duas subclasses derivadas de A: B e C;
* e há também a subclasse mais baixa chamada D, derivada de B e C (ou C e B, pois estas duas variantes significam coisas diferentes em Python)

Consegue ver o diamante ali?

Dê uma vista de olhos ao código no editor. A mesma estrutura, mas expressa em Python.

class A:

pass

class B(A):

pass

class C(A):

print('c')

pass

class D(B, C):

pass

d = D()

Algumas linguagens de programação proíbem a herança múltipla, e como consequência, não o deixam construir um diamante - esta é a rota que Java e C# escolheram seguir desde as suas origens.

O Python, contudo, escolheu uma rota diferente - permite herança múltipla, e não se importa se escrever e executar um código como o que está no editor. Mas não se esqueça do MRO - é sempre ele que manda.

Vamos reconstruir o nosso exemplo da página anterior para o tornar mais parecido com um diamante, tal como abaixo:

class Top:

def m\_top(self):

print("top")

class Middle\_Left(Top):

def m\_middle(self):

print("middle\_left")

class Middle\_Right(Top):

def m\_middle(self):

print("middle\_right")

class Bottom(Middle\_Left, Middle\_Right):

def m\_bottom(self):

print("bottom")

object = Bottom()

object.m\_bottom()

object.m\_middle()

object.m\_top()

Nota: ambas as classes Middle definem **um método com o mesmo nome**: m\_middle().

Introduz uma pequena incerteza na nossa amostra, embora estejamos absolutamente certos de que pode responder à seguinte pergunta-chave: qual dos dois métodos m\_middle() será realmente invocado quando a seguinte linha for executada?

Object.m\_middle()

Por outras palavras, o que verá no ecrã: middle\_left ou middle\_right?

Não precisa de se apressar - pense duas vezes e tenha em mente o MRO do Python!

Está preparado?

Sim, tem razão. A invocação ativará o método m\_middle() , que vem da classe Middle\_Left . A explicação é simples: a classe está listada antes Middle\_Right na lista de herança da classe Bottom . Se quiser ter a certeza de que não há dúvidas, tente trocar estas duas classes na lista e verifique os resultados.

Se quiser experimentar algumas impressões mais profundas sobre a herança múltipla e pedras preciosas, tente modificar o nosso snippet e equipar a classe Upper com outro espécime do método m\_middle() , e investigue cuidadosamente o seu comportamento.

Como pode ver, os diamantes podem trazer alguns problemas à sua vida - tanto os verdadeiros como os oferecidos pelo Python.

# Key takeaways

1. Um método chamado \_\_str\_\_() é responsável pela **conversão do conteúdo de um objeto numa string (mais ou menos) legível**. Pode redefini-la se quiser que o seu objeto se possa apresentar de uma forma mais elegante. Por exemplo:

class Mouse:

def \_\_init\_\_(self, name):

self.my\_name = name

def \_\_str\_\_(self):

return self.my\_name

the\_mouse = Mouse('mickey')

print(the\_mouse) # Prints "mickey".

2. Uma função chamada issubclass(Class\_1, Class\_2) é capaz de determinar se Class\_1 é uma **subclasse** de Class\_2. Por exemplo:

class Mouse:

pass

class LabMouse(Mouse):

pass

print(issubclass(Mouse, LabMouse), issubclass(LabMouse, Mouse)) # Prints "False True"

3. Uma função chamada isinstance(Object, Class) verifica se um objeto **vem de uma classe indicada**. Por exemplo:

class Mouse:

pass

class LabMouse(Mouse):

pass

mickey = Mouse()

print(isinstance(mickey, Mouse), isinstance(mickey, LabMouse)) # Prints "True False".

4. Um operador chamado is verifica se duas variáveis se referem **ao mesmo objeto**. Por exemplo:

class Mouse:

pass

mickey = Mouse()

minnie = Mouse()

cloned\_mickey = mickey

print(mickey is minnie, mickey is cloned\_mickey) # Prints "False True".

5. Uma função sem parâmetros chamada super() devolve uma **referência para a superclasse mais próxima da classe**. Por exemplo:

class Mouse:

def \_\_str\_\_(self):

return "Mouse"

class LabMouse(Mouse):

def \_\_str\_\_(self):

return "Laboratory " + super().\_\_str\_\_()

doctor\_mouse = LabMouse();

print(doctor\_mouse) # Prints "Laboratory Mouse".

6. Os métodos, bem como as variáveis de instância e classe definidas numa superclasse, são **automaticamente herdados** pelas suas subclasses. Por exemplo:

class Mouse:

Population = 0

def \_\_init\_\_(self, name):

Mouse.Population += 1

self.name = name

def \_\_str\_\_(self):

return "Hi, my name is " + self.name

class LabMouse(Mouse):

pass

professor\_mouse = LabMouse("Professor Mouser")

print(professor\_mouse, Mouse.Population) # Prints "Hi, my name is Professor Mouser 1"

7. Para encontrar qualquer propriedade objeto/classe, o Python procura-a no interior:

* do objeto em si;
* de todas as classes envolvidas na linha de herança do objeto, de baixo para cima;
* se houver mais do que uma classe num determinado caminho de herança, o Python digitaliza-as da esquerda para a direita;
* se ambos acima falharem, a exceção AttributeError é levantada.

8. Se qualquer uma das subclasses definir uma variável de método/classe variável/instância com o mesmo nome que a existente na superclasse, o novo nome **sobrepõe-se** a qualquer uma das instâncias anteriores do nome. Por exemplo:

class Mouse:

def \_\_init\_\_(self, name):

self.name = name

def \_\_str\_\_(self):

return "My name is " + self.name

class AncientMouse(Mouse):

def \_\_str\_\_(self):

return "Meum nomen est " + self.name

mus = AncientMouse("Caesar") # Prints "Meum nomen est Caesar"

print(mus)

**Exercícios**

**Cenário**

Suponha que o seguinte snippet de código foi executado com sucesso:

class Dog:

kennel = 0

def \_\_init\_\_(self, breed):

self.breed = breed

Dog.kennel += 1

def \_\_str\_\_(self):

return self.breed + " says: Woof!"

class SheepDog(Dog):

def \_\_str\_\_(self):

return super().\_\_str\_\_() + " Don't run away, Little Lamb!"

class GuardDog(Dog):

def \_\_str\_\_(self):

return super().\_\_str\_\_() + " Stay where you are, Mister Intruder!"

rocky = SheepDog("Collie")

luna = GuardDog("Dobermann")

Agora responda às perguntas dos exercícios 1-4.

**Exercício 1**

Qual é o output esperado do seguinte snippet de código?  
  
print(rocky)

print(luna)

Verifique

**Exercício 2**

Qual é o output esperado do seguinte snippet de código?  
  
print(issubclass(SheepDog, Dog), issubclass(SheepDog, GuardDog))

print(isinstance(rocky, GuardDog), isinstance(luna, GuardDog))

Verifique

**Exercício 3**

Qual é o output esperado do seguinte snippet de código?  
  
print(luna is luna, rocky is luna)

print(rocky.kennel)

Verifique

**Exercício 4**

Defina uma subclasse SheepDogchamada LowlandDog, e equipe-a com um método \_\_str\_\_() que substitui um método herdado com o mesmo nome. O novo método dog's \_\_str\_\_() deve devolver a string “Woof! I don't like mountains!" .

**Mais sobre exceções**

Discutir a programação de objetos oferece uma excelente oportunidade para regressar às exceções. A natureza objetiva das exceções de Python torna-as uma ferramenta muito flexível, capaz de se adaptar a necessidades específicas, mesmo aquelas que ainda não conhece.

Antes de mergulharmos na**face objetiva das exceções**, queremos mostrar-lhe alguns aspetos sintáticos e semânticos da forma como o Python trata o bloco try-except, uma vez que oferece um pouco mais do que aquilo que temos apresentado até agora.

A primeira característica que queremos discutir aqui é um possível ramo adicional, que pode ser colocado dentro (ou melhor, diretamente atrás) do bloco try-except - é a parte do código que começa com else - tal como no exemplo no editor.

def reciprocal(n):

try:

n = 1 / n

except ZeroDivisionError:

print("Division failed")

return None

else:

print("Everything went fine")

return n

print(reciprocal(2))

print(reciprocal(0))

Um código assim rotulado é executado quando (e apenas quando) nenhuma exceção tiver sido levantada dentro da parte try: . Podemos dizer que exatamente um ramo pode ser executado após try: - quer o que começa com except (não se esqueça de que pode haver mais do que um ramo deste tipo) ou o que começa com else.

Nota: o ramo else: tem de ser localizado após o último ramo except .

O código de exemplo produz o seguinte output:

Everything went fine

0.5

Division failed

None

O bloco try-except pode ser prolongado de mais uma forma - adicionando uma parte encabeçada pela keyword finally (deve ser o último ramo do código concebido para lidar com exceções).

Nota: estas duas variantes (else e finally) não são de modo algum dependentes, e podem coexistir ou ocorrer independentemente.

O bloco finally é sempre executado (finaliza a execução do bloco try-except, daí o seu nome), independentemente do que aconteceu anteriormente, mesmo quando se levanta uma exceção, independentemente de esta ter sido tratada ou não.

Veja o código no editor.

def reciprocal(n):

try:

n = 1 / n

except ZeroDivisionError:

print("Division failed")

n = None

else:

print("Everything went fine")

finally:

print("It's time to say goodbye")

return n

print(reciprocal(2))

print(reciprocal(0))

O seu output é:

Everything went fine

It's time to say good bye

0.5

Division failed

It's time to say good bye

None

# Exceções são classes

Todos os exemplos anteriores se contentavam em detectar um tipo específico de exceção e responder-lhe de uma forma adequada. Agora vamos aprofundar, e olhar para dentro da própria exceção.

Provavelmente não ficará surpreendido ao saber que **exceções são classes**. Além disso, quando é levantada uma exceção, um objeto da classe é instanciado, e passa por todos os níveis de execução do programa, procurando o ramo exceto o que está preparado para lidar com ele.

Tal objeto traz alguma informação útil que pode ajudá-lo a identificar com precisão todos os aspetos da situação pendente. Para atingir esse objetivo, o Python oferece uma variante especial da cláusula de exceção - pode encontrá-la no editor.

try:

i = int("Hello!")

except Exception as e:

print(e)

print(e.\_\_str\_\_())

Como pode ver, a declaração except é prolongada, e contém uma frase adicional que começa com a keyword as , seguida por um identificador. O identificador é concebido para apanhar o objeto de exceção, para que se possa analisar a sua natureza e tirar as devidas conclusões.

Nota: o scope do identificador abrange o seu ramo except , e não vai mais longe.

O exemplo apresenta uma forma muito simples de utilizar o objeto recebido - basta imprimi-lo (como pode ver, o output é produzido pelo método do objeto \_\_str\_\_() ) e contém uma breve mensagem descrevendo a razão.

A mesma mensagem será impressa se não houver nenhum bloco de encaixe except no código, e o Python é forçado a lidar com ela sozinho.

Todas as exceções Python integradas formam uma hierarquia de classes. Não há qualquer obstáculo à sua extensão se a considerar razoável.

Veja o código no editor.

def print\_exception\_tree(thisclass, nest = 0):

if nest > 1:

print(" |" \* (nest - 1), end="")

if nest > 0:

print(" +---", end="")

print(thisclass.\_\_name\_\_)

for subclass in thisclass.\_\_subclasses\_\_():

print\_exception\_tree(subclass, nest + 1)

print\_exception\_tree(BaseException)

Este programa descarta todas as classes de exceção pré-definidas sob a forma de uma impressão em forma de árvore.

Como **uma árvore é um exemplo perfeito de uma estrutura de dados recursiva**, uma recursividade parece ser a melhor ferramenta para a atravessar. A função print\_exception\_tree() toma dois argumentos:

* um ponto dentro da árvore a partir do qual começamos a atravessar a árvore;
* um nível de nesting (vamos utilizá-lo para construir um desenho simplificado dos ramos da árvore)

Comecemos pela raiz da árvore - a raiz das classes de exceção de Python é a classe BaseException (é uma superclasse de todas as outras exceções).

Para cada uma das classes encontradas, realize o mesmo conjunto de operações:

* imprimir o seu nome, retirado da propriedade \_\_name\_\_ ;
* iterar através da lista de subclasses entregue pelo método \_\_subclasses\_\_() , e invocar recursivamente a função print\_exception\_tree() , incrementando o nível de nesting respetivamente.

Note-se como desenhámos os ramos e as bifurcações. A impressão não é classificada de forma alguma - pode tentar classificá-la você mesmo, se quiser um desafio. Além disso, existem algumas imprecisões subtis na forma como alguns ramos são apresentados. Isso também pode ser corrigido, se assim o desejar.

É este o seu aspeto:

BaseException

+---Exception

| +---TypeError

| +---StopAsyncIteration

| +---StopIteration

| +---ImportError

| | +---ModuleNotFoundError

| | +---ZipImportError

| +---OSError

| | +---ConnectionError

| | | +---BrokenPipeError

| | | +---ConnectionAbortedError

| | | +---ConnectionRefusedError

| | | +---ConnectionResetError

| | +---BlockingIOError

| | +---ChildProcessError

| | +---FileExistsError

| | +---FileNotFoundError

| | +---IsADirectoryError

| | +---NotADirectoryError

| | +---InterruptedError

| | +---PermissionError

| | +---ProcessLookupError

| | +---TimeoutError

| | +---UnsupportedOperation

| | +---herror

| | +---gaierror

| | +---timeout

| | +---Error

| | | +---SameFileError

| | +---SpecialFileError

| | +---ExecError

| | +---ReadError

| +---EOFError

| +---RuntimeError

| | +---RecursionError

| | +---NotImplementedError

| | +---\_DeadlockError

| | +---BrokenBarrierError

| +---NameError

| | +---UnboundLocalError

| +---AttributeError

| +---SyntaxError

| | +---IndentationError

| | | +---TabError

| +---LookupError

| | +---IndexError

| | +---KeyError

| | +---CodecRegistryError

| +---ValueError

| | +---UnicodeError

| | | +---UnicodeEncodeError

| | | +---UnicodeDecodeError

| | | +---UnicodeTranslateError

| | +---UnsupportedOperation

| +---AssertionError

| +---ArithmeticError

| | +---FloatingPointError

| | +---OverflowError

| | +---ZeroDivisionError

| +---SystemError

| | +---CodecRegistryError

| +---ReferenceError

| +---BufferError

| +---MemoryError

| +---Warning

| | +---UserWarning

| | +---DeprecationWarning

| | +---PendingDeprecationWarning

| | +---SyntaxWarning

| | +---RuntimeWarning

| | +---FutureWarning

| | +---ImportWarning

| | +---UnicodeWarning

| | +---BytesWarning

| | +---ResourceWarning

| +---error

| +---Verbose

| +---Error

| +---TokenError

| +---StopTokenizing

| +---Empty

| +---Full

| +---\_OptionError

| +---TclError

| +---SubprocessError

| | +---CalledProcessError

| | +---TimeoutExpired

| +---Error

| | +---NoSectionError

| | +---DuplicateSectionError

| | +---DuplicateOptionError

| | +---NoOptionError

| | +---InterpolationError

| | | +---InterpolationMissingOptionError

| | | +---InterpolationSyntaxError

| | | +---InterpolationDepthError

| | +---ParsingError

| | | +---MissingSectionHeaderError

| +---InvalidConfigType

| +---InvalidConfigSet

| +---InvalidFgBg

| +---InvalidTheme

| +---EndOfBlock

| +---BdbQuit

| +---error

| +---\_Stop

| +---PickleError

| | +---PicklingError

| | +---UnpicklingError

| +---\_GiveupOnSendfile

| +---error

| +---LZMAError

| +---RegistryError

| +---ErrorDuringImport

+---GeneratorExit

+---SystemExit

+---KeyboardInterrupt

# Anatomia detalhada das exceções

Vamos analisar mais de perto o objeto da exceção, uma vez que há aqui alguns elementos realmente interessantes (voltaremos à questão em breve quando considerarmos as técnicas de base de input/output de Python, uma vez que o seu subsistema de exceção estende um pouco estes objetos).

A classe BaseException introduz uma propriedade chamada args. É um **tuple concebido para reunir todos os argumentos passados ao construtor da classe**. Está vazio se a construção foi invocada sem quaisquer argumentos, ou contém apenas um elemento quando o construtor recebe um argumento (não contamos o argumento self aqui), e assim por diante.

Preparámos uma função simples para imprimir a propriedade args de uma forma elegante. Pode ver a função no editor.

def print\_args(args):

lng = len(args)

if lng == 0:

print("")

elif lng == 1:

print(args[0])

else:

print(str(args))

try:

raise Exception

except Exception as e:

print(e, e.\_\_str\_\_(), sep=' : ' ,end=' : ')

print\_args(e.args)

try:

raise Exception("my exception")

except Exception as e:

print(e, e.\_\_str\_\_(), sep=' : ', end=' : ')

print\_args(e.args)

try:

raise Exception("my", "exception")

except Exception as e:

print(e, e.\_\_str\_\_(), sep=' : ', end=' : ')

print\_args(e.args)

Utilizámos a função para imprimir o conteúdo da propriedade args em três casos diferentes, em que a exceção da classe Exception é levantada de três maneiras diferentes. Para o tornar mais espetacular, imprimimos também o próprio objeto, juntamente com o resultado da invocação \_\_str\_\_() .

O primeiro caso parece rotineiro - há apenas o nome Exception após a keyword raise . Isto significa que o objeto desta classe foi criado de uma forma muito rotineira.

O segundo e terceiro casos podem parecer um pouco estranhos à primeira vista, mas não há nada de estranho aqui - estas são apenas as invocações do construtor. Na segunda declaração raise , o construtor é invocado com um argumento, e no terceiro com dois.

Como pode ver, o output do programa reflete isto, mostrando o conteúdo apropriado da propriedade args :

: :

my exception : my exception : my exception

('my', 'exception') : ('my', 'exception') : ('my', 'exception')

**Como criar a sua própria exceção**

A hierarquia de exceções não está fechada nem terminada, e pode sempre alargá-la se quiser ou precisar de criar o seu próprio mundo povoado com as suas próprias exceções.

Pode ser útil quando se cria um módulo complexo que detecta erros e levanta exceções, e se pretende que as exceções sejam facilmente distinguíveis de quaisquer outras trazidas pelo Python.

Isto é feito **definindo as suas próprias, novas exceções como subclasses derivadas de subclasses predefinidas**.

Nota: se quiser criar uma exceção que será utilizada como um caso especializado de qualquer exceção incorporada, derive-a apenas desta. Se quer construir a sua própria hierarquia, e não quer que ela esteja intimamente ligada à árvore de exceção de Python, derive-a de qualquer uma das classes de exceção de topo, como Exception.

Imagine que criou uma aritmética completamente nova, regida pelas suas próprias leis e teoremas. É evidente que a divisão também foi redefinida, e tem de se comportar de uma forma diferente da divisão de rotina. É também claro que esta nova divisão deve levantar a sua própria exceção, diferente da ZeroDivisionError, mas é razoável assumir que em algumas circunstâncias, você (ou o utilizador da sua aritmética) pode querer tratar todas as divisões zero da mesma maneira.

Exigências como estas podem ser satisfeitas da forma apresentada no editor. Veja o código, e vamos analisá-lo:

class MyZeroDivisionError(ZeroDivisionError):

pass

def do\_the\_division(mine):

if mine:

raise MyZeroDivisionError("some worse news")

else:

raise ZeroDivisionError("some bad news")

for mode in [False, True]:

try:

do\_the\_division(mode)

except ZeroDivisionError:

print('Division by zero')

for mode in [False, True]:

try:

do\_the\_division(mode)

except MyZeroDivisionError:

print('My division by zero')

except ZeroDivisionError:

print('Original division by zero')

* Definimos a nossa própria exceção, denominada MyZeroDivisionError, derivado a partir da incorporada ZeroDivisionError. Como pode ver, decidimos não acrescentar novos componentes à classe.  
    
  Com efeito, uma exceção a esta classe pode ser - dependendo do ponto de vista desejado - tratada como uma simples ZeroDivisionError, ou considerada separadamente.

* A função do\_the\_division() levanta ou uma exceção MyZeroDivisionError ou ZeroDivisionError , dependendo do valor do argumento.  
    
  A função é invocada quatro vezes no total, enquanto as duas primeiras invocações são tratadas utilizando apenas um ramo except (o mais geral) e os dois últimos com dois ramos diferentes, capazes de distinguir as exceções (não se esqueça: a ordem dos ramos faz uma diferença fundamental!)

Quando se vai construir um universo completamente novo cheio de criaturas completamente novas que não têm nada em comum com todas as coisas familiares, pode querer **construir a sua própria estrutura de exceção**.

Por exemplo, se trabalhar num grande sistema de simulação destinado a modelar as atividades de um restaurante de pizza, pode ser desejável formar uma hierarquia separada de exceções.

Pode começar a construí-la **definindo uma exceção geral como uma nova classe base** para qualquer outra exceção especializada. Fizemo-lo da seguinte forma:

class PizzaError(Exception):

def \_\_init\_\_(self, pizza, message):

Exception.\_\_init\_\_(self, message)

self.pizza = pizza

Nota: vamos recolher aqui informações mais específicas do que uma regular Exception faz, pelo que o nosso construtor aceitará dois argumentos:

* um que especifique uma pizza como um assunto do processo,
* e um que contenha uma descrição mais ou menos precisa do problema.

Como pode ver, passamos o segundo parâmetro ao construtor de superclasse, e guardamos o primeiro dentro da nossa própria propriedade.

Um problema mais específico (como um excesso de queijo) pode exigir uma exceção mais específica. É possível derivar a nova classe a partir da classe já definida PizzaError , como fizemos aqui:

class TooMuchCheeseError(PizzaError):

def \_\_init\_\_(self, pizza, cheese, message):

PizzaError.\_init\_\_(self, pizza, message)

self.cheese = cheese

A exceção TooMuchCheeseError precisa de mais informações do que a exceção regular PizzaError , por isso adicionamo-la ao construtor - o nome cheese é então armazenado para processamento posterior.

Veja o código no editor. Juntámos as duas exceções previamente definidas e aproveitámo-las para trabalhar num pequeno trecho de exemplo.

class PizzaError(Exception):

def \_\_init\_\_(self, pizza, message):

Exception.\_\_init\_\_(self, message)

self.pizza = pizza

class TooMuchCheeseError(PizzaError):

def \_\_init\_\_(self, pizza, cheese, message):

PizzaError.\_\_init\_\_(self, pizza, message)

self.cheese = cheese

def make\_pizza(pizza, cheese):

if pizza not in ['margherita', 'capricciosa', 'calzone']:

raise PizzaError(pizza, "no such pizza on the menu")

if cheese > 100:

raise TooMuchCheeseError(pizza, cheese, "too much cheese")

print("Pizza ready!")

for (pz, ch) in [('calzone', 0), ('margherita', 110), ('mafia', 20)]:

try:

make\_pizza(pz, ch)

except TooMuchCheeseError as tmce:

print(tmce, ':', tmce.cheese)

except PizzaError as pe:

print(pe, ':', pe.pizza)

Um destes é levantado dentro do make\_pizza() funcionam quando qualquer uma destas duas situações erradas é descoberta: um pedido de pizza errado, ou um pedido de demasiado queijo.

Nota:

* remoção do ramo a começar por except TooMuchCheeseError fará com que todas as exceções que aparecem sejam classificadas como PizzaError;
* remoção do ramo a começar por except PizzaErrorwill causar as exceções TooMuchCheeseError a permanecer sem manuseio, e fará com que o programa termine.

A solução anterior, embora elegante e eficiente, tem uma fraqueza importante. Devido à forma algo descontraída de declarar os construtores, as novas exceções não podem ser usadas tal como estão sem uma lista completa dos argumentos requeridos.

Vamos remover esta fraqueza ao **definir os valores predefinidos para todos os parâmetros do construtor**. Dê uma vista de olhos:

class PizzaError(Exception):

def \_\_init\_\_(self, pizza='uknown', message=''):

Exception.\_\_init\_\_(self, message)

self.pizza = pizza

class TooMuchCheeseError(PizzaError):

def \_\_init\_\_(self, pizza='uknown', cheese='>100', message=''):

PizzaError.\_\_init\_\_(self, pizza, message)

self.cheese = cheese

def make\_pizza(pizza, cheese):

if pizza not in ['margherita', 'capricciosa', 'calzone']:

raise PizzaError

if cheese > 100:

raise TooMuchCheeseError

print("Pizza ready!")

for (pz, ch) in [('calzone', 0), ('margherita', 110), ('mafia', 20)]:

try:

make\_pizza(pz, ch)

except TooMuchCheeseError as tmce:

print(tmce, ':', tmce.cheese)

except PizzaError as pe:

print(pe, ':', pe.pizza)

Agora, se as circunstâncias o permitirem, é possível utilizar os nomes das classes sozinhos.

# Key takeaways

1. A função else: da declaração try é executado quando não houve nenhuma exceção durante a execução do bloco try: .

2. O método finally: da declaração try é **sempre** executado.

3. A sintaxe except *Exception\_Name* as an *exception\_object*: permite-lhe interceptar um objeto portador de informação sobre uma exceção pendente. A propriedade do objeto chamada args (um tuple) armazena todos os argumentos passados para o construtor do objeto.

4. As classes de exceção podem ser alargadas para as enriquecer com novas capacidades, ou para adotar as suas características a novas exceções definidas.

Por exemplo:

try:

assert \_\_name\_\_ == "\_\_main\_\_"

except:

print("fail", end=' ')

else:

print("success", end=' ')

finally:

print("done")

O código faz o output: success done.

**Exercício 1**

Qual é o output esperado do seguinte código?

import math

try:

print(math.sqrt(9))

except ValueError:

print("inf")

else:

print("fine")

Verifique

**Exercício 2**

Qual é o output esperado do seguinte código?

import math

try:

print(math.sqrt(-9))

except ValueError:

print("inf")

else:

print("fine")

finally:

print("the end")

Verifique

**Exercício 3**

Qual é o output esperado do seguinte código?

import math

class NewValueError(ValueError):

def \_\_init\_\_(self, name, color, state):

self.data = (name, color, state)

try:

raise NewValueError("Enemy warning", "Red alert", "High readiness")

except NewValueError as nve:

for arg in nve.args:

print(arg, end='! ')

**Parabéns! Completou o *PE2: Módulo 3*.**

Muito bem! Chegou ao fim do Módulo 3 e completou um marco importante na sua educação em programação Python. Aqui está um breve resumo dos objetivos que abordou e com os quais se familiarizou no Módulo 3:

* Os fundamentos e conceitos básicos da programação orientada a objetos;
* As diferenças entre as abordagens processuais e de objeto no exemplo da stack;
* propriedades (variáveis de instância e de classe, atributos)
* métodos (métodos de classe e de objeto, o construtor, parâmetros e propriedades)
* o conceito de herança (funções, métodos, hierarquias de classe, polimorfismo, composição, herança única vs. múltipla)
* a natureza objetiva das exceções em Python.

Está agora pronto para fazer o quiz do módulo e tentar o desafio final: Teste do módulo 3, que o ajudará a avaliar o que aprendeu até agora.



issub