Um Guia para Iniciantes em Programação Orientada a Objetos (POO) Python

Daniel Diaz, Novembro 8, 2022

Importante: para as implementações dos fontes utilize preferencialmente o Jupyter ou Colab, pois posso no decorrer do curso para ver seus códigos.

Procure no início da célula incluir como comentário a data em que iniciou a implementação.

A programação é uma arte. E, como na arte, selecionar os pincéis e tintas adequados é essencial para produzir os melhores trabalhos. A Programação orientada a objetos Python é uma dessas habilidades.

A escolha da linguagem de programação correta é uma parte crucial de qualquer projeto, e pode levar a um desenvolvimento fluido e agradável ou a um pesadelo completo. Portanto, seria melhor se você usasse a linguagem mais adequada para seu caso de uso.

Essa é a principal razão para aprender programação orientada a objetos em Python, que é também uma das linguagens de programação mais populares.

Vamos aprender!

Tabela de Conteúdos

Um exemplo de programa Python

Antes de aprofundar no assunto, vamos fazer uma pergunta: você já escreveu um programa Python como o que está abaixo?

```
secret_number = 20

while True:
    number = input('Guess the number: ')
    try:
        number = int(number)
    except:
        print('Sorry that is not a number')
        continue
    if number != secret_number:
        if number > secret_number:
            print(number, 'is greater than the secret
number')
        elif number < secret_number:
            print(number, 'is less than the secret number')
    else:
        print('You guessed the number:', secret_number)
        break
```

Este código é um simples adivinhador de números. Ele cumpre perfeitamente seu propósito.

Mas aí vem um enorme problema: e se lhe pedíssemos para implementar um novo recurso? Poderia ser algo simples – por exemplo:

“Se a entrada for um múltiplo do número secreto, dê uma dica ao usuário”.

O programa se tornaria rapidamente complexo e pesado à medida que você aumentasse o número de características e, portanto, o número total de condicionais aninhados.

É exatamente isso que a programação orientada a objetos tenta resolver.

Programação é uma arte que requer as ferramentas adequadas para construir algo bonito 💯

Requisitos para aprender POO Python

Antes de entrar na programação orientada a objetos, é recomendável fortemente que você tenha uma compreensão firme dos conceitos básicos de Python.

Classificar tópicos considerados “básicos” pode ser difícil. Por causa disso, projetamos uma checklist com todos os principais conceitos necessários para aprender programação orientada a objetos em Python.

- **Variável:** Nome simbólico que aponta para um objeto específico (veremos o que **os objetos** significam).

- **Operadores aritméticos:** Adição (+), subtração (-), multiplicação (*), divisão (/), divisão inteira (//), modulo (%).

- **Tipos de dados incorporados:** Numéricos (inteiros, flutuadores, complexos), Seqüências (strings, listas, tuplas), Booleanos (True, False), Dicionários, e Conjuntos.

- **Expressões booleanas:** Expressões em que o resultado é **True** ou **False.**
- **Condicional:** Avalia uma expressão booleana e faz algum processo dependendo do resultado. Manipulado por declarações **if/else.**
- **Loop:** Execução repetida de blocos de código. Pode ser um loop **for** ou **while**.
- **Funções:** Bloco de código organizado e reutilizável. Você os cria com a palavra-chave **def**.
- **Argumentos:** Objetos passados para uma função. Por exemplo: `sum([1, 2, 4])`
- **Execute um script Python**: Abra um terminal ou uma linha de comando e digite "python <nome do arquivo>".
- **Abra uma shell Python**: Abra um terminal e digite `python` ou `python3`, dependendo do seu sistema.

Agora que você tem estes conceitos claros, você pode avançar com a compreensão da programação orientada a objetos.

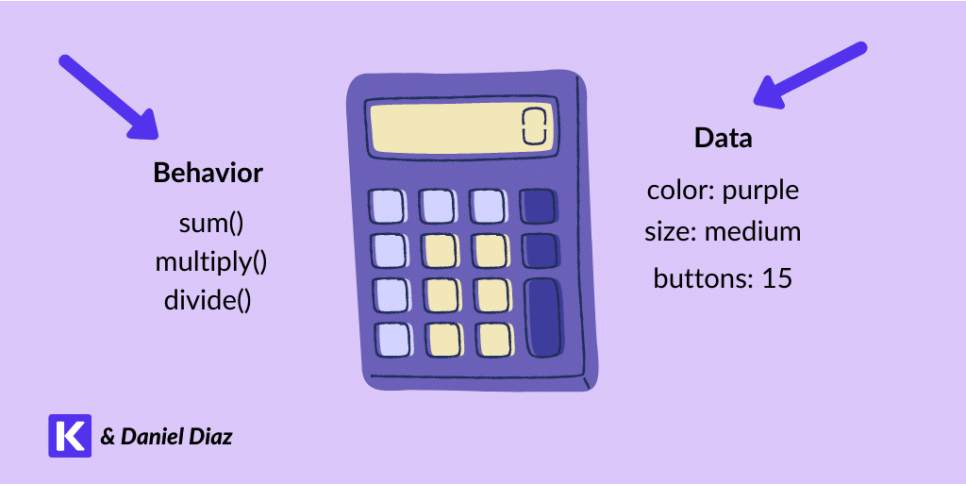
O que é Programação Orientada a Objetos em Python?

A Programação Orientada a Objetos (POO) é um paradigma de programação no qual podemos pensar em problemas complexos como objetos.

Um paradigma é uma teoria que fornece a base para a solução de problemas.

Portanto, quando falamos de POO, estamos nos referindo a um conjunto de conceitos e padrões que usamos para resolver problemas com objetos.

Um objeto em Python é uma única coleção de dados (atributos) e comportamento (métodos). Você pode pensar em objetos como coisas reais ao seu redor. Por exemplo, considere as calculadoras:



Uma calculadora pode ser um objeto.

Como você pode notar, os dados (atributos) são sempre substantivos, enquanto os comportamentos (método) são sempre verbos.

Esta compartimentação é o conceito central da Programação Orientada a Objetos. Você constrói objetos que armazenam dados e contêm tipos específicos de funcionalidade.

Por que usamos programação orientada a objetos em Python?

POO permite que você crie um software seguro e confiável. Muitas estruturas Python e bibliotecas usam este paradigma para construir sua base de código. Alguns exemplos são Django, Kivy, pandas, NumPy, e TensorFlow.

Vejamos as principais vantagens de usar a POO em Python.

Vantagens da POO Python

As seguintes razões farão com que você opte pelo uso de programação orientada a objetos em Python.

Todas as Linguagens de Programação Moderna usam POO

Este paradigma é independente do idioma. Se você aprender POO em Python, você será capaz de usá-lo no seguinte:

- Java
- PHP (certifique-se de ler a comparação entre PHP e Python)
- Ruby
- Javascript
- C#
- Kotlin

Todos essas linguagens ou são nativamente orientados a objetos ou incluem opções de funcionalidade orientada a objetos. Se você quiser aprender qualquer uma delas após Python, será mais fácil – você encontrará muitas semelhanças entre as linguagens que trabalham com objetos.

POO permite que você codifique mais rápido

Codificar mais rápido não significa escrever menos linhas de código. Significa que você pode implementar mais recursos em menos tempo, sem comprometer a estabilidade de um projeto.

A programação orientada a objetos permite a reutilização do código através da implementação da abstração. Este princípio torna seu código mais conciso e legível.

Como você deve saber, os programadores passam muito mais tempo lendo o código do que escrevendo-o. É a razão pela qual a legibilidade é sempre mais importante do que obter recursos o mais rápido possível.

A produtividade decresce com código não legível

POO ajuda você a evitar o código Spaghetti

Você se lembra do programa de adivinhação de números no início deste artigo?

Se você continuar acrescentando recursos, você terá muitas declarações **if** aninhados no futuro. Este emaranhado de infinitas linhas de código é chamado de código spaghetti, e você deve evitá-lo o máximo possível.

A POO nos dá a possibilidade de comprimir toda a lógica nos objetos, evitando assim longos trechos de **"if"** aninhados.

POO melhora sua análise de qualquer situação

Uma vez que você tenha alguma experiência com POO, você será capaz de pensar em problemas como objetos pequenos e específicos.

Este entendimento leva a uma rápida inicialização do projeto.

Programação Estruturada vs Programação Orientada a Objetos

A programação estruturada é o paradigma mais utilizado pelos iniciantes porque é a maneira mais simples de construir um pequeno programa.

Envolve a execução sequencial de um programa Python. Isso significa que você está dando ao computador uma lista de tarefas e depois executando-as de cima para baixo.

Vamos ver um exemplo de programação estruturada com um programa de cafeteria.

```
small = 2
regular = 5
big = 6
user_budget = input('What is your budget? ')
try:
   user_budget = int(user_budget)
except:
   print('Please enter a number')
   exit()
if user_budget > 0:
   if user_budget >= big:
       print('You can afford the big coffee')
       if user_budget == big:
```

```
            print('It\'s complete')
        else:
            print('Your change is', user_budget - big)
    elif user_budget == regular:
        print('You can afford the regular coffee')
        print('It\'s complete')
    elif user_budget >= small:
        print('You can buy the small coffee')
        if user_budget == small:
            print('It\'s complete')
        else:
            print('Your change is', user_budget - small)
```

O código acima age como um vendedor de café. Ele lhe pedirá um orçamento e então "venderá" o maior café que você puder comprar.

Tente rodá-lo no terminal. Ele executará passo a passo, dependendo da sua entrada.

Este código funciona perfeitamente, mas temos três problemas:

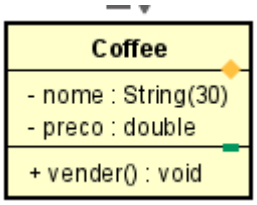
1. Tem uma lógica muito repetida.
2. Utiliza muitos condicionamentos **if**.
3. Vai ser difícil ler e modificar.

A POO foi inventada como uma solução para todos esses problemas.

Vamos ver o programa acima implementado com POO.
É apenas para comparar programação estruturada e programação orientada a objetos.

```
class Coffee:
        # Constructor
        def __init__(self, name, price):
                self.name = name
                self.price = float(price)
        def check_budget(self, budget):
                # Check if the budget is valid
                if not isinstance(budget, (int, float)):
                        print('Enter float or int')
                        exit()
                if budget < 0:
                    print('Sorry you don\'t have money')
                    exit()
        def get_change(self, budget):
                return budget - self.price
        def sell(self, budget):
                self.check_budget(budget)
                if budget >= self.price:
                        print(f'You can buy the {self.name}
coffee')
                        if budget == self.price:
                                print('It\'s complete')
```

```
                        else:
                                print(f'Here is your change
{self.get_change(budget)}$')
                        exit('Thanks for your transaction')
```



Nota: Todos os conceitos a seguir serão explicados mais profundamente através do artigo.

O código acima representa uma **classe** chamada "Coffee". Ele tem dois atributos – "nome" e "preço" – e ambos são utilizados nos métodos. O método primário é "vender", que processa toda a lógica necessária para completar o processo de venda.

Se você tentar administrar essa classe, não obterá nenhum resultado. Isso ocorre principalmente porque estamos apenas declarando o "modelo" para os cafés, não os cafés em si.

Vamos implementar essa classe com o seguinte código:

```
#Cafeteria
small = Coffee('Small', 2)
regular = Coffee('Regular', 5)
big = Coffee('Big', 6)
try:
   user_budget = float(input('What is your budget? '))
except ValueError:
   exit('Please enter a number')

for coffee in [big, regular, small]:
   coffee.sell(user_budget)
```

Aqui estamos fazendo **instâncias,** ou objetos de café, da classe "Coffee", chamando então o método "vender" de cada café até que o usuário possa pagar qualquer opção.

Obteremos o mesmo resultado com ambas as abordagens, mas podemos estender a funcionalidade do programa muito melhor com o POO.

Abaixo está uma tabela comparando a programação orientada a objetos e a programação estruturada:

POO	**Structured Programming**
Mais fácil de manter	Difícil de manter
Não se repita (DRY)	Código repetido em muitos lugares

Pequenos trechos de código reutilizados em muitos lugares	Uma grande quantidade de código em poucos lugares
Abordagem por objetos	Abordagem por código de bloco
Mais fácil de depurar	Mais difícil de depurar
Grande curva de aprendizagem	Curva de aprendizagem mais simples
Utilizado em grandes projetos	Otimizado para programas simples

Para concluir a comparação de paradigmas:

- Nenhum dos dois paradigmas é perfeito (o POO pode ser esmagador para ser usado em projetos simples).
- Estas são apenas duas maneiras de resolver um problema; existem outras por aí.
- A POO é usada em grandes bases de código, enquanto a programação estruturada é principalmente para projetos simples.

Tudo é um objeto em Python

Você tem usado o POO o tempo todo sem se dar conta.

Mesmo quando se usa outros paradigmas em Python, ainda se usa objetos para fazer quase tudo.

Isso porque, em Python, *tudo* é um objeto.

Lembre-se da definição de objeto: Um objeto em Python é uma única coleção de dados (atributos) e comportamento (métodos).

Isso corresponde a qualquer tipo de dado em Python.

Uma string é uma coleção de dados (caracteres) e comportamentos (**upper()**, **lower()**, etc.). O mesmo se aplica a **inteiros**, flutuante, **booleanos**, **listas** e dicionários.

Antes de continuar, vamos rever o significado dos atributos e métodos.

Atributos e métodos

Os atributos são **variáveis** internas dentro dos objetos, enquanto os métodos são **funções** que produzem algum comportamento.

Vamos fazer um exercício simples na shell Python. Você pode abri-la digitando `python` ou `python3` em seu terminal.

```
~
❯ python
Python 3.9.5 (default, May 24 2021, 12:50:35)
[GCC 11.1.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

shell Python

Agora, vamos trabalhar com a shell Python para descobrir métodos e tipos.

```
>>> kinsta = 'Kinsta, Premium Application, Database, and Managed WordPress hosting'
>>> kinsta.upper()
'KINSTA, PREMIUM APPLICATION, DATABASE, AND MANAGED WORDPRESS HOSTING'
```

Na segunda linha, estamos chamando um método de string, **upper()**. Ele retorna o conteúdo da cadeia tudo em maiúsculas. Entretanto, ele não muda a variável original.

```
>>> kinsta
'Kinsta, Premium Application, Database, and Managed WordPress hosting'
```

Vamos mergulhar em funções valiosas ao trabalhar com objetos.

A função **tipo()** permite obter o tipo de um objeto. O "tipo" é a classe à qual o objeto pertence.

```
>>> type(kinsta)
# class 'str'
```

A função **dir()** retorna todos os atributos e métodos de um objeto. Vamos testá-lo com a variável **kinsta.**

```
>>> dir(kinsta)
['__add__', '__class__',  ..... 'upper', 'zfill']
```

Agora, tente imprimir alguns dos atributos ocultos deste objeto.

```
 >>> kinsta.__class__ # class 'str' e>
```

Isto produzirá a classe à qual o objeto **kinsta** pertence. Portanto, podemos dizer que a única coisa que a função do **tipo** retorna é o atributo **__classe__** de um objeto.

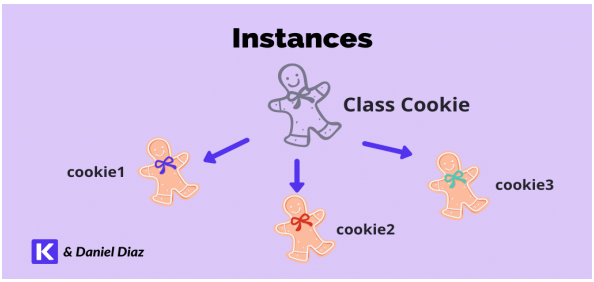
Você pode experimentar todos os tipos de dados, descobrindo todos os seus atributos e métodos diretamente no terminal. Você pode saber mais sobre os tipos de dados incorporados na documentação oficial.

Seu primeiro objeto em Python

Uma **classe** é como um modelo**.** Ela permite criar objetos personalizados com base nos atributos e métodos que você define.

Nome da classe(Sub)	NomeDaClasse
Atributos (Adjetivos)	TipoDeAcesso atributo1: tipo TipoDeAcesso atributo2: tipo TipoDeAcesso atributo3: tipo TipoDeAcesso atributo : tipo
Verbos (Infinitivo)	TipoDeAcesso Método1() TipoDeAcesso Método2() TipoDeAcesso MétodoN()

Você pode pensar nisso como um **cortador de cookies** que você modifica para assar os cookies perfeitos (objetos, não cookies de rastreamento), com características definidas: Forma, tamanho, e muito mais.



Instâncias em Python

Por outro lado, temos **instâncias.** Uma instância é um objeto individual de uma classe, que tem um endereço de memória único.

Agora que você sabe o que são classes e instâncias, vamos definir algumas!

Para definir uma classe em Python, você usa a palavra-chave **class**, seguida de seu nome. Neste caso, você criará uma classe chamada **Cookie**.

Nota: Em Python, usamos a convenção de nomes em maiúsculas para as classes..

```
class Cookie:
      pass
```

De preferência, execute o Jupyter, Colab(Google) em outros casos outro compilador Python e digite o código acima. Para criar uma instância de uma classe, basta digitar seu nome e parênteses após ela. É o mesmo processo que a invocação de uma função.

```
cookie1 = Cookie()
```

Parabéns – você acaba de criar seu primeiro objeto em Python! Você pode verificar sua identificação e digitar com o seguinte código:

```
id(cookie1)
140130610977040 # Unique identifier of the object

type(cookie1)
<class '__main__.Cookie'>
```

Como você pode ver, este cookie tem um identificador único na memória, e seu tipo é **Cookie**.

Você também pode verificar se um objeto é uma instância de uma classe com a função **isinstance().**

```
isinstance(cookie1, Cookie)
# True
isinstance(cookie1, int)
# False
isinstance('a string', Cookie)
# False
```

Método construtor

O método **__init__( )** também é chamado de "construtor". É chamado de Python cada vez que instanciamos um objeto.

O construtor cria o estado inicial do objeto com o conjunto mínimo de parâmetros que ele precisa para existir. Vamos modificar a classe **Cookie**, para que ela aceite parâmetros em seu construtor.

```
class Cookie:
	# Constructor
	def __init__(self, name, shape, chips='Chocolate'):
		# Instance attributes
		self.name = name
		self.shape = shape
		self.chips = chips
```

Na classe **Cookie**, cada cookie deve ter um nome, forma e lascas. Nós definimos o último como "Chocolate".

Por outro lado, **self** refere-se à instância da classe (o próprio objeto).

Atualize a classe anterior crie uma instância do cookie.

```
cookie2 = Cookie()
# TypeError
```

Você vai perceber um erro. Isso porque você deve fornecer o conjunto mínimo de dados que o objeto precisa para viver – neste caso, **nome** e **forma,** uma vez que já definimos **chips** para "Chocolate".

```
cookie2 = Cookie('Awesome cookie', 'Star')
```

Para acessar os atributos de uma instância, você deve usar a notação de pontos.

```
cookie2.name
# 'Awesome cookie'
cookie2.shape
# 'Star'
cookie2.chips
# 'Chocolate'
```

Por enquanto, a classe **Cookie** não tem nada muito suculento. Vamos adicionar um método de **cozimento de amostra()** para tornar as coisas mais interessantes.

```
class Cookie:
	# Constructor
	def __init__(self, name, shape, chips='Chocolate'):
		# Instance attributes
		self.name = name
```

```
            self.shape = shape
            self.chips = chips

      # The object is passing itself as a parameter
      def bake(self):
            print(f'This {self.name}, is being baked with
the shape {self.shape} and chips of {self.chips}')
            print('Enjoy your cookie!')
```

Para chamar um método, use a notação de ponto e invoque-o como uma função.

```
cookie3 = Cookie('Baked cookie', 'Tree')
cookie3.bake()
# This Baked cookie, is being baked with the shape Tree and
chips of Chocolate
Enjoy your cookie!
```

Os 4 pilares do POO em Python

A Programação Orientada a Objetos inclui quatro pilares principais:

1. Abstração

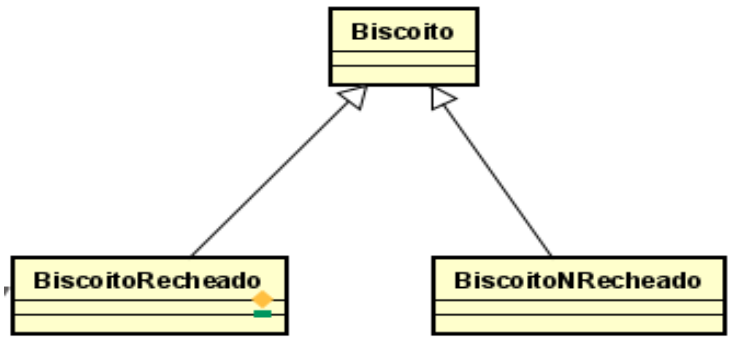
A abstração esconde e protege do usuário a funcionalidade interna de uma aplicação. O usuário pode ser o cliente final ou outros desenvolvedores.

Podemos encontrar **abstração** em nossa vida diária. Por exemplo, você sabe como usar seu telefone, mas provavelmente não sabe exatamente o que está acontecendo dentro dele cada vez que você abre um aplicativo.

Outro exemplo é o próprio Python. Você sabe como usá-lo para construir software funcional, e pode fazê-lo mesmo que não entenda o funcionamento interno do Python.

Aplicar o mesmo ao código permite coletar todos os objetos em um problema e **abstrair** a funcionalidade padrão em classes.

2. Herança



A herança nos permite definir várias **subclasses** a partir de uma classe já definida.

O objetivo principal é seguir o princípio DRY(Don't repeat yourself). **Você poderá reutilizar muito código, implementando todos os componentes de compartilhamento em** superclasses**.**

Você pode pensar nisso como o conceito de **herança genética da** vida real. As crianças (subclasse) são o resultado da herança entre dois pais (superclasses). Eles herdam todas as características físicas (atributos) e alguns comportamentos (métodos) comuns.

3. Polimorfismo

O polimorfismo nos permite modificar ligeiramente os métodos e atributos das **subclasses** previamente definidas na **superclasse**.

O significado literal é "**muitas formas**". Isso porque construímos métodos com o mesmo nome, mas com funcionalidades diferentes.

Voltando à ideia anterior, as crianças também são um exemplo perfeito de polimorfismo. Elas podem herdar um comportamento definido de **ficar_cansadas()** mas de uma maneira ligeiramente diferente, por exemplo, ficar com fome a cada 4 horas em vez de a cada 6 horas.

4. Encapsulamento

Encapsulamento é o processo no qual protegemos a integridade interna dos dados em uma classe.

Embora não haja uma declaração **privada** em Python, você pode aplicar o encapsulamento usando o mangling em Python. Há métodos especiais chamados **getters ou métodos assessores** e **setters ou métodos mutadores** em outras palavras, nos permitem alterar ou mudar dados em uma variável.

Imaginemos uma classe **humana** que tem um atributo único chamado **_altura**. Você só pode modificar este atributo dentro de certas restrições (é quase impossível ser superior a 3 metros).

Construindo uma área com a calculadora de resolução de shapes

Uma das melhores coisas sobre Python é que ele nos permite criar uma grande variedade de software, desde um programa CLI (interface de linha de comando) até um aplicativo web complexo.

Agora que você aprendeu os conceitos de pilares do POO, é hora de aplicá-los a um projeto real.

Nota: Todos os códigos a seguir estarão disponíveis dentro deste repositório GitHub. Uma ferramenta de revisão de código que nos ajuda a gerenciar as versões de código com Git.

Sua tarefa é criar uma calculadora de área com as seguintes shapes:

- Quadrado
- Rectângulo
- Triângulo
- Círculo
- Hexágono

Formato class shape

Primeiramente, criar um arquivo **calculator.py** e abri-lo. Como já temos os objetos para trabalhar, será fácil de **abstract** em uma classe.

Você pode analisar as características comuns e descobrir que todas estas são **formatos 2D**. Portanto, a melhor opção é criar uma shape class com um método **get_area()** do qual cada forma herdará.

Nota: Todos os métodos devem ser verbos. Isso porque este método é chamado **get_area()** e não **area()**.

```
class Shape:
	def __init__(self):
		pass

	def get_area(self):
		pass
```

O código acima define a classe; no entanto, ainda não há nada de interessante nela.

Vamos implementar a funcionalidade padrão da maioria dessas formas.

```
class Shape:
	def __init__(self, side1, side2):
		self.side1 = side1
		self.side2 = side2

	def get_area(self):
		return self.side1 * self.side2

	def __str__(self):
		return	f'The	area	of	this
{self.__class__.__name__} is: {self.get_area()}'
```

Vamos analisar o que estamos fazendo com este código:

- No método **__init__**, estamos solicitando dois parâmetros, **side1** e **side2**. Estes permanecerão como **atributos de instância**.
- A função **get_area()** retorna a área da forma. Neste caso, ela está usando a fórmula de área de um retângulo, uma vez que será mais fácil de implementar com outras formas.
- O método **__str__()** é um "método mágico", assim como **__init__().** Ele permite modificar a forma como uma instância irá imprimir.

- O atributo oculto **auto.__classe__.__name__** se refere ao nome da classe. Se você estivesse trabalhando com uma classe **Triângulo**, esse atributo seria "Triângulo".

Rectangle Class

Como implementamos a fórmula de área do Retângulo, poderíamos criar uma classe simples de **Rectangle** que não faz nada além de herdar da classe **shape.**

Para aplicar **inheritance** em Python, você criará uma classe como de costume e cercará a **superclasse** da qual você quer herdar entre parênteses.

```
# Folded base class
class Shape: ...

class Rectangle(Shape): # Superclass in Parenthesis
      pass
```

Square Class

Podemos ter uma excelente abordagem do **polimorfismo** com a classe **Square.**

Lembre-se de que um quadrado é apenas um retângulo cujos quatro lados são todos iguais. Isso significa que podemos usar a mesma fórmula para obter a área.

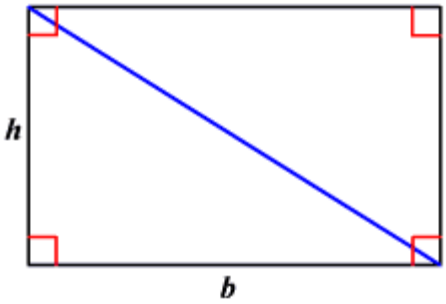
Podemos fazer isso modificando o método **init**, aceitando apenas um **side** como parâmetro e passando esse valor de lado para o construtor da classe **Rectangle.**

```
# Folded classes
class Shape: ...
class Rectangle(Shape): ...

class Square(Rectangle):
      def __init__(self, side):
            super().__init__(side, side)
```

Como você pode ver, a superfunção passa duas vezes o parâmetro **side** para a **superclasse**. Em outras palavras, ela passa o **side** tanto como **side1** como **side2** para o construtor previamente definido.

Triangle Class

Um triângulo é metade do tamanho do retângulo que o cerca.



Relação entre triângulos e retângulos (Fonte de imagem: Varsity tutors).

Portanto, podemos herdar da classe **Rectangle** e modificar o método **get_area** para corresponder à fórmula da área triangular, que é a metade da base multiplicada pela altura.

```
# Folded classes
class Shape: ...
class Rectangle(Shape): ...
class Square(Rectangle): ...

class Triangle(Rectangle):
      def __init__(self, base, height):
            super().__init__(base, height)

      def get_area(self):
            area = super().get_area()
            return area / 2
```

Outro caso de uso da função **super()** é chamar um método definido na **superclasse** e armazenar o resultado como uma variável. É isso que está acontecendo dentro do método **get_area().**

Classe Circle

Você pode encontrar a área do círculo com a fórmula πr^2, onde **r** é o raio do círculo. Isso significa que temos que modificar o método **get_area()** para implementar essa fórmula.

Nota: Podemos importar o valor aproximado de π a partir do módulo de matemática

```
# Folded classes
class Shape: ...
class Rectangle(Shape): ...
class Square(Rectangle): ...
class Triangle(Rectangle): …

# At the start of the file
from math import pi
```

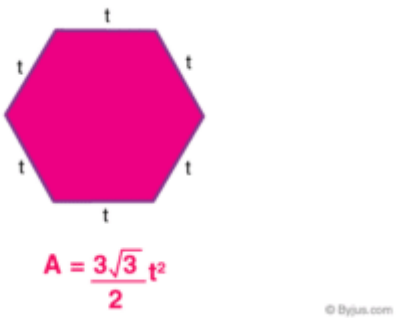
```
class Circle(Shape):
	def __init__(self, radius):
		self.radius = radius

	def get_area(self):
		return pi * (self.radius ** 2)
```

O código acima define a classe **Circle**, que usa um construtor diferente e métodos **get_area().**

Embora **o Circle** herde da classe **Shape**, você pode redefinir cada um dos métodos e atribuí-los a seu gosto.

Regular Hexagon Class

Precisamos apenas do comprimento de um lado de um hexágono regular para calcular sua área. É semelhante à classe **Square**, onde apenas passamos uma discussão para o construtor.



Fórmula da área do hexágono (Fonte de imagem: BYJU'S)

Entretanto, a fórmula é bem diferente, e implica no uso de uma raiz quadrada. É por isso que você usará a função **sqrt()** do módulo de matemática.

```
# Folded classes
class Shape: ...
class Rectangle(Shape): ...
class Square(Rectangle): ...
class Triangle(Rectangle): …
class Circle(Shape): …

# Import square root
from math import sqrt

class Hexagon(Rectangle):

	def get_area(self):
		return (3 * sqrt(3) * self.side1 ** 2) / 2
```

Testando nossas aulas

Você pode entrar em um modo interativo ao executar um arquivo Python usando um depurador. A maneira mais simples de fazer isso é usando a função de breakpoint integrada.

Nota: Esta função está disponível apenas em Python 3.7 ou mais recente.

```
from math import pi, sqrt
# Folded classes
class Shape: ...
class Rectangle(Shape): ...
class Square(Rectangle): ...
class Triangle(Rectangle): …
class Circle(Shape): …
class Hexagon(Rectangle): …

breakpoint()
```

Agora, execute o arquivo Python e brinque com as classes que você criou.

```
$ python calculator.py

(Pdb) rec = Rectangle(1, 2)(Pdb) print(rec)
The area of this Rectangle is: 2
(Pdb) sqr = Square(4)
(Pdb) print(sqr)
The area of this Square is: 16
(Pdb) tri = Triangle(2, 3)
(Pdb) print(tri)
The area of this Triangle is: 3.0
(Pdb) cir = Circle(4)
(Pdb) print(cir)
The area of this Circle is: 50.26548245743669
(Pdb) hex = Hexagon(3)
(Pdb) print(hex)
The area of this Hexagon is: 23.382685902179844
```

Desafio

Criar uma classe com um método de **run** onde o usuário pode escolher uma forma e calcular sua área.

Quando tiver completado o desafio, você pode enviar um pedido de retirada para o GitHub repo ou publicar sua solução na seção de comentários.

Pronto para começar a aprender programação orientada a objectos em Python? □ Veio ao lugar certo ☺Clique para Tweetar

Resumo

A programação orientada a objetos é um paradigma no qual resolvemos problemas pensando neles como **objetos**. Se você entende Python OOP, você também pode aplicá-lo facilmente em linguagens como Java, PHP, Javascript, e C#.

Neste artigo, você tomou conhecimento:

- O conceito de orientação a objetos em Python
- Vantagens da programação orientada a objetos sobre a programação estruturada
- Noções básicas de programação orientada a objetos em Python
- Conceito de **classes** e como utilizá-las no Python
- O **construtor** de uma classe em Python
- **Métodos** e **atributos** em Python
- Os quatro pilares do OOP
- Implementar **abstraction**, **inheritance** e **polimorfismo** em um projeto

FONTES:

1) Disponível em: KINSTA https://kinsta.com/pt/blog/programacao-orientada-objetos-python/ acesso em 06mar2023
2) NETO MOREIRA, OZIEL – Entendendo e Dominando o Java, 3ª. edição, São Paulo, Digerati Books, 2009.