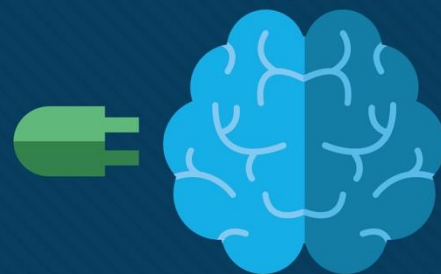




Python: Introdução à Orientação a Objetos



Professor José Fernando Lino Santiago





Introdução à Orientação a Objetos

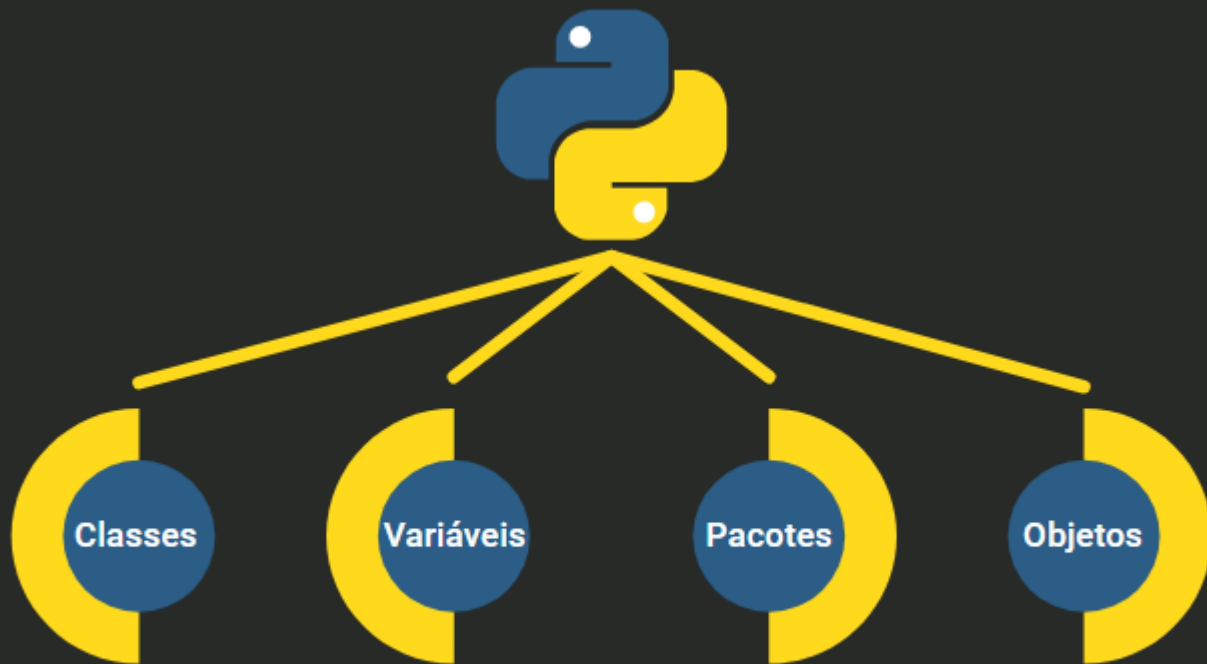
Orientação a Objetos na Linguagem Python

A linguagem Python foi criada para o desenvolvimento orientado a objetos, o que facilita muito a compreensão de alguns conceitos, torna a programação mais simples de ser compreendida e facilita o trabalho em equipe!

Projetos desenvolvidos utilizando a Orientação a Objetos são mais **estáveis**, de **fácil manutenção** e sua **reutilização** é mais simples.

Na linguagem Python, quando iniciamos um projeto, apesar dele poder ser desenvolvido utilizando os conceitos de programação procedural, a linguagem já vai pré organizá-lo para a orientação a objetos, pois ele será organizado por meio de estruturas denominadas **Classes** que vão armazenar trechos de códigos relacionados entre si.





A Comunidade Python determina **algumas convenções** para atribuir nomes a esses elementos.

Algumas convenções, caso ignoradas, não vão ocasionar erros ao código, mas manter um código organizado, seguindo as convenções, **facilita muito o desenvolvimento e a compreensão do código**, principalmente, quando o trabalho é realizado em equipe.



- | Referente a caracteres, seguir o mesmo padrão de variáveis e objetos.
- | Sempre iniciar as classes com caracteres maiúsculos, inclusive as iniciais de nomes compostos:
Exemplo: **MinhaClasse()**



- | Utilizar somente caracteres e letras minúsculas.
- | No caso de variáveis com nome composto, utilizar o *underline* para separação das palavras.
- | Não iniciar o nome com números (podemos utilizar números no nome, mas não devemos iniciar com eles).
- | Não utilizar caracteres especiais.
- | Não utilizar espaçamento em branco.
- | Evitar utilizar os caracteres I e O.



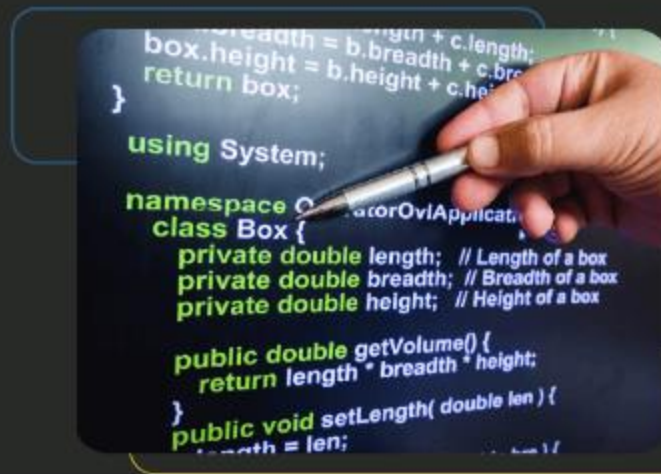
- | Utilizar nomes pequenos.
- | Utilizar sempre caracteres minúsculos.
- | Utilizar o *underline* para unir nomes compostos.

Classes no Projeto Python

O desenvolvimento de uma aplicação envolve os mais diferentes tipos de informações, que são modeladas com estruturas que conhecemos como **classes**.

A partir das classes é possível **criar objetos**, ou seja, uma classe é um "molde" para a criação de objetos.

Podemos afirmar que classe é um **Tipo Abstrato de Dados (TAD)**, ou seja, o código que define e implementa um novo tipo de informação.





Criação de Objetos

Antes de continuar o projeto Python, é importante fixar os principais conceitos da orientação a objetos.

Classe

Objeto

Instância

Podemos dizer que a classe é o projeto do objeto, contendo o código de programação.



O primeiro passo para criar um objeto em Python é definir uma classe. A classe é como um modelo que define as propriedades e comportamentos do objeto. Por exemplo, se você estiver criando uma classe para representar um carro, poderá definir atributos como modelo, cor, ano e marca, e métodos como acelerar, frear e virar.



Criação de Objetos

Antes de continuar o projeto Python, é importante fixar os principais conceitos da orientação a objetos.

Classe

Objeto

Instância

É a execução do código de uma classe.

Quando executamos o código de uma classe é criado um novo objeto na memória.

Uma nova instância de um objeto é um tipo abstrato de informação de um novo tipo de dado.

Criar o objeto:

Depois de definir a classe, você pode criar um objeto dessa classe. Isso é feito usando a palavra-chave `class` seguida pelo nome da classe. Por exemplo, se você tiver uma classe `Carro`, poderá criar um objeto chamado `meu_carro` usando a seguinte sintaxe:

```
main.py
1  # Criar o objeto
2  meu_carro = Carro()
```

Atribuir valores aos atributos: Depois de criar o objeto, você pode atribuir valores aos seus atributos. Isso é feito usando a sintaxe **objeto.atributo = valor**. Por exemplo, se você quiser definir o modelo do carro como "Fusca", poderá fazer o seguinte:

 main.py

```
1  # Atribuir valores aos atributos:  
2  meu_carro.modelo = "Fusca"
```



Criação de Objetos

Antes de continuar o projeto Python, é importante fixar os principais conceitos da orientação a objetos.

Classe

Objeto

Instância

Podemos dizer que a instância é o objeto sendo executado.

Quando criamos um novo objeto, afirmamos que estamos criando uma instância dele.



Definir métodos: Os métodos são as funções que definem o comportamento do objeto. Você pode definir um método dentro da classe usando a sintaxe `def nome_do_metodo(self, argumentos):`. O argumento `self` é obrigatório e representa o próprio objeto. Por exemplo, se você quiser definir um método `acelerar` para o objeto `Carro`, poderá fazer o seguinte:

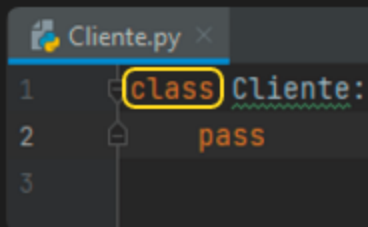
```
# Definir métodos:
class Carro:
    def acelerar(self):
        print("Acelerando...")
```

- Chamar métodos: Depois de definir um método, você pode chamá-lo usando a sintaxe `objeto.nome_do_metodo()`. Por exemplo, se você quiser chamar o método `acelerar` do objeto `meu_carro`, poderá fazer o seguinte:

```
1  # Chamar métodos:  
2  meu_carro.acelerar()
```

Após o desenvolvimento de um projeto Python, chegou a hora de entendermos a estrutura da classe.

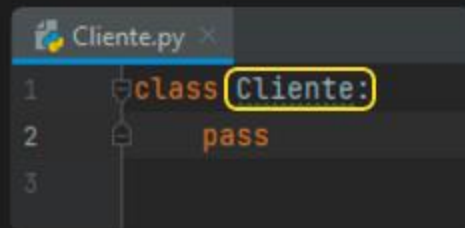
>>1. Iniciamos com a palavra **class**.



```
1 class Cliente:
2     pass
3
```

Após o desenvolvimento de um projeto Python, chegou a hora de entendermos a estrutura da classe.

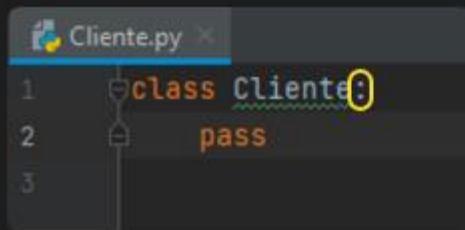
>>2. Adicionamos o nome da classe.



```
1 class Cliente:
2     pass
3
```


Após o desenvolvimento de um projeto Python, chegou a hora de entendermos a estrutura da classe.

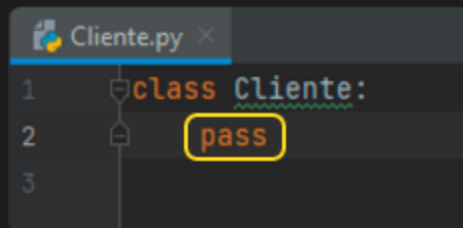
>>3. O operador **dois pontos** faz parte da sintaxe da declaração de classe.



```
1 class Cliente:
2     pass
3
```

Após o desenvolvimento de um projeto Python, chegou a hora de entendermos a estrutura da classe.

>>4. Utilizamos a palavra reservada *pass*, quando nenhuma estrutura será definida no corpo dessa classe.



```
Cliente.py x
1 class Cliente:
2     pass
3
```

The screenshot shows a code editor window titled 'Cliente.py'. It contains a Python class definition. Line 1: `class Cliente:`. Line 2: `pass`. Line 3: (empty line). The word `pass` is highlighted with a yellow box. There are line numbers 1, 2, and 3 on the left side of the editor. There are also navigation arrows on the left and right sides of the editor window.

Após o desenvolvimento de um projeto Python, chegou a hora de entendermos a estrutura da classe.

>>5. Em seguida, podemos definir o corpo da classe.

A screenshot of a code editor window titled "Cliente.py". It shows a Python class definition. Line 1: `class Cliente:`. Line 2: `pass`. The word "pass" is highlighted with a yellow box. Line 3 is empty. There are line numbers 1, 2, and 3 on the left side of the code block.

```
1 class Cliente:
2     pass
3
```

Declaração dos Membros da Classe

Utilizamos os membros da classe para manter uma mesma estrutura de tudo que pertence a determinada informação da classe.

Conforme já estudamos, **classes são um tipo abstrato de dados**, sendo assim, haverá valores e esses necessitam de funções específicas para serem manipulados.

Temos dois tipos básicos de membros que compõe uma classe:

Atributos (Propriedades)

Os atributos armazenam as características de uma classe.
Os atributos são as declarações de variáveis da classe.

Métodos

São ações da classe, suas funções.
Representam os estados e ações dos objetos quando instanciados.

Definindo o Método Construtor

O **Método Construtor** é definido de forma **implícita ou explícita** por todas as classes e, como o próprio nome já cita, é utilizado para **construir o objeto**.

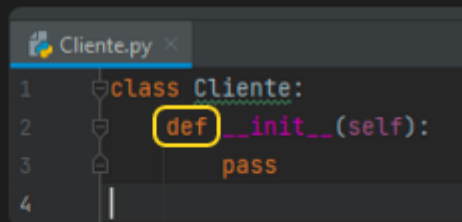
Todas as vezes que um objeto estiver sendo criado (instanciado) é por meio do **Construtor** que ele será inicializado.

Este método é invocado, automaticamente, pela máquina virtual do Python todas as vezes que um objeto é criado.



Inserindo o Método Construtor na Classe Cliente

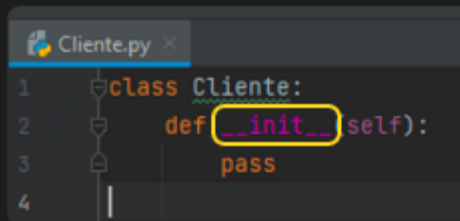
>>1. A palavra **def** é utilizada para a declaração de método.



```
Cliente.py x
1 class Cliente:
2     def __init__(self):
3         pass
4
```

Inserindo o Método Construtor na Classe Cliente

>>2. Para definir o construtor adicionamos: underline, underline, a palavra reservada init, underline, underline. O `init()` é um método especial que será chamado sempre que criarmos um objeto da classe.



```
1 class Cliente:
2     def __init__(self):
3         pass
4
```

Inserindo o Método Construtor na Classe Cliente

>>3. Incluímos o parâmetro obrigatório **self**, que está presente em todos os métodos. Resumidamente, podemos afirmar que o parâmetro **self**, neste momento, exporta as características do objeto.

```

1 class Cliente:
2     def __init__(self):
3         pass
4

```



Importante

Todo método Python tem **self** como primeiro parâmetro.

A palavra reservada representa o objeto em si, portanto, sempre que quisermos especificar atributos de objetos, devemos associá-lo à palavra reservada **self**.

Inserindo o Método Construtor na Classe Cliente

>>4. Para finalizar, adicionamos dois pontos (:).



```
Cliente.py
1 class Cliente:
2     def __init__(self):
3         pass
4
```



Vamos criar um exemplo de classe Carro

```
main.py
1  # Vamos criar um exemplo de classe Carro
2  class Carro:
3      def __init__(self, marca, modelo, ano, cor, combustivel,
4          velocidade=0):
5          self.marca = marca
6          self.modelo = modelo
7          self.ano = ano
8          self.cor = cor
9          self.combustivel = combustivel
10         self.velocidade = velocidade
11     def acelerar(self, incremento):
12         self.velocidade += incremento
13         print(f"{self.modelo} acelerou para {self.velocidade}
14         km/h")
15     def frear(self, decremento):
16         if self.velocidade - decremento < 0:
17             self.velocidade = 0
18         else:
19             self.velocidade -= decremento
20         print(f"{self.modelo} freou para {self.velocidade}
21         km/h")
22     def ligar(self):
23         print(f"{self.modelo} ligou")
24
25     def desligar(self):
26         print(f"{self.modelo} desligou")
```

Segue o programa completo utilizando a classe Carro como exemplo:

main.py

```
1  # Vamos criar um exemplo de classe Carro
2  class Carro:
3      def __init__(self, marca, modelo, ano, cor, preco):
4          self.marca = marca
5          self.modelo = modelo
6          self.ano = ano
7          self.cor = cor
8          self.preco = preco
9          self.velocidade = 0
10
11     def acelerar(self, valor):
12         self.velocidade += valor
13
14     def frear(self, valor):
15         self.velocidade -= valor
16         if self.velocidade < 0:
17             self.velocidade = 0
18
19     def mostrar_velocidade(self):
20         print(f"A velocidade atual é de {self.velocidade}
21         km/h.")
22
23     carro1 = Carro("Chevrolet", "Camaro", 2021, "Amarelo", 280000)
24     carro2 = Carro("Ferrari", "458 Spider", 2020, "Vermelho",
25     1250000)
26
27     print("Carro 1:")
28     print(f"Marca: {carro1.marca}")
29     print(f"Modelo: {carro1.modelo}")
```

```
27     print(f"Modelo: {carro1.modelo}")
28     print(f"Ano: {carro1.ano}")
29     print(f"Cor: {carro1.cor}")
30     print(f"Preço: {carro1.preco}")
31     carro1.acelerar(80)
32     carro1.mostrar_velocidade()
33
34     print("Carro 2:")
35     print(f"Marca: {carro2.marca}")
36     print(f"Modelo: {carro2.modelo}")
37     print(f"Ano: {carro2.ano}")
38     print(f"Cor: {carro2.cor}")
39     print(f"Preço: {carro2.preco}")
40     carro2.frear(50)
41     carro2.mostrar_velocidade()
```

> Console x Shell x +

```
Carro 1:
Marca: Chevrolet
Modelo: Camaro
Ano: 2021
Cor: Amarelo
Preço: 280000
A velocidade atual é de 80 km/h.
Carro 2:
Marca: Ferrari
Modelo: 458 Spider
Ano: 2020
Cor: Vermelho
Preço: 1250000
A velocidade atual é de 0 km/h.
> □
```



Vamos trabalhar !!!

Exercícios

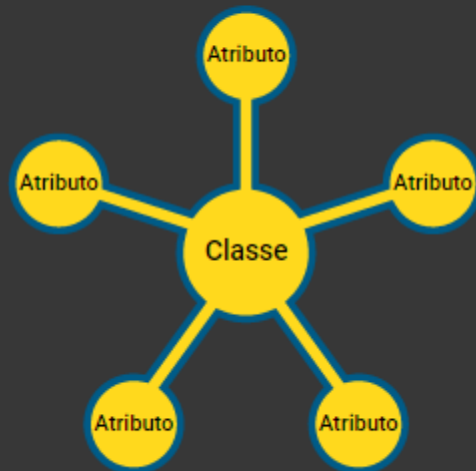


1. Adicione um atributo "placa" à classe Carro e exiba-o no programa.
2. Crie um método "ligar" na classe Carro que imprime a mensagem "O carro está ligado".
3. Adicione um atributo "combustível" à classe Carro e um método "abastecer" que recebe um valor de combustível e acrescenta esse valor ao atributo "combustível".
4. Crie um método "consumir_combustivel" que recebe um valor de combustível e reduz o atributo "combustivel" do objeto em questão pelo valor informado.
5. Modifique o método "frear" para imprimir a mensagem "O carro parou" quando a velocidade for igual a zero.
6. Crie uma nova classe chamada "Moto" com atributos semelhantes à classe Carro e um método "empinar" que imprime a mensagem "A moto está empinando".
7. Crie uma nova instância de um objeto moto e chame o método "empinar".
8. Crie um método estático "calcular_media" que recebe uma lista de valores e retorna a média desses valores.
9. Crie uma lista de objetos Carro e calcule a média de preços desses carros utilizando o método estático "calcular_media".
10. Crie um método de classe "total_objetos" que retorna o número total de objetos instanciados da classe Carro.

Espero que esses exercícios ajudem a praticar a programação orientada a objetos em Python!

Atributos da Classe

Lembre-se de que os **atributos** de uma **classe** representam as **características** que ela possui.



Agora, vamos ver como **adicionar atributos**, **modificar uma classe** e **personalizar o Método Construtor**.

Neste exemplo, temos a classe Carro com seus atributos (marca, modelo, ano, cor, preco) e métodos (acelerar, frear, mostrar_velocidade), além da instância de dois objetos carro1 e carro2, e a chamada de alguns métodos dessas instâncias. No final, o programa exibe informações dos carros e suas velocidades.

Como Adicionar Atributos à Classe?

Para adicionar atributos a uma classe, **basta definir o nome do atributo** acompanhado da palavra reservada **self**, no método especial denominado **__init__** do Método Construtor.

```
Cliente.py x
1 class Cliente:
2     def __init__(self):
3
4         self.nome
5         self.telefone
6
```

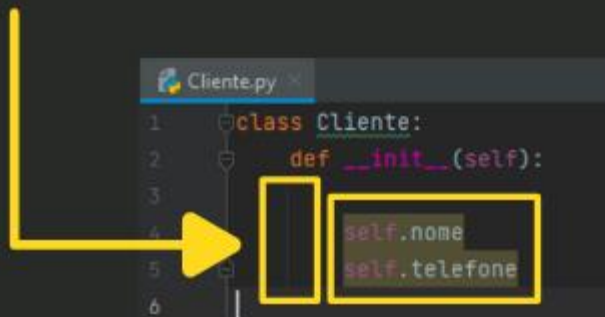

Como Modificar a Classe?

Vamos adicionar dois atributos à classe Cliente.

Primeiramente, **remova a palavra reservada pass.**

Em seguida, **insira o código correspondente aos atributos.**

Observe que o recuo indica que os atributos pertencem ao Método Construtor da classe.



```

1 class Cliente:
2     def __init__(self):
3
4         self.nome
5         self.telefone
6

```

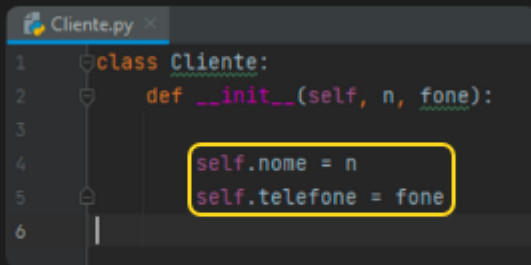
Como Personalizar o Método Construtor?

Como já vimos, o **Método Construtor** tem a finalidade de **estabelecer os valores obrigatórios para construção de um novo objeto**.

O Método Construtor da classe **pode conter um conjunto de parâmetros**. Com isso, podemos determinar os valores para inicialização dos atributos.

Isso garante um melhor funcionamento de toda a estrutura do objeto, **obrigando ao programador** determinar **valores default** no momento da inicialização do objeto.

>>2. Inicializamos os atributos com os valores passados por parâmetro.



```
1 class Cliente:
2     def __init__(self, n, fone):
3
4         self.nome = n
5         self.telefone = fone
6
```

Como Personalizar o Método Construtor?

Como já vimos, o **Método Construtor** tem a finalidade de **estabelecer os valores obrigatórios para construção de um novo objeto**.

O Método Construtor da classe **pode conter um conjunto de parâmetros**. Com isso, podemos determinar os valores para inicialização dos atributos.

Isso garante um melhor funcionamento de toda a estrutura do objeto, **obrigando ao programador** determinar **valores default** no momento da inicialização do objeto.

>>1. Com o parâmetro **self** são passados os parâmetros que serão utilizados para inicialização dos atributos.

```
Cliente.py x
1 class Cliente:
2     def __init__(self, n, fone):
3
4         self.nome = n
5         self.telefone = fone
6
```



Importante

Na linguagem Python não é recomendável criar mais de um Método Construtor para a classe.

Instanciando Objetos

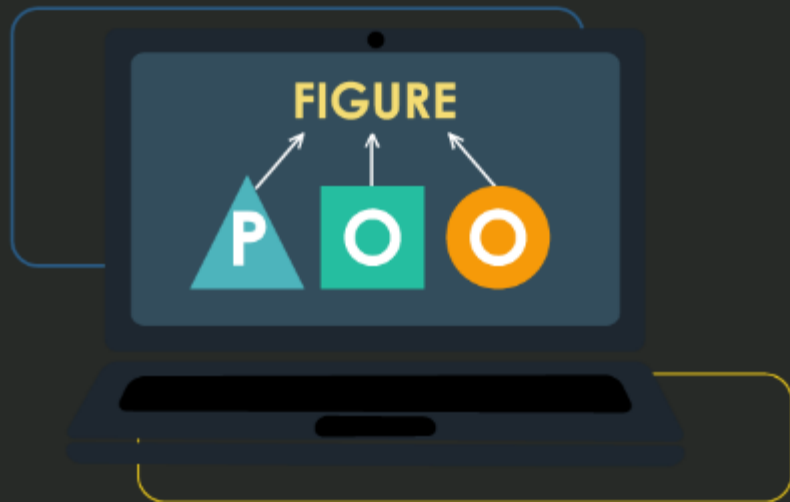
Na linguagem Python, as classes são utilizadas para definição dos objetos.

Podemos afirmar que a classe é o nosso código e, para que esse código seja utilizado, precisamos criar os objetos, assim, criamos **instâncias do objeto**.

Na orientação a objetos, **instância e objetos são sinônimos**.

```
>>> class Learning:
...     def __init__(self, name, age, gender):
...         self.title = learn
...         self.subtitle = python
...         self.paragraph = everyday
...
>>> Programmer = Learning("learn", python, "everyday")
>>> print Sue
<__main__.Programmer instance at 0x32111320>
>>> print Programmer.subtitle
python
```

Objetos no Python



Na linguagem Python, todo objeto criado possui um código de identificação **composto por um número inteiro não negativo**, conhecido como **ID**.

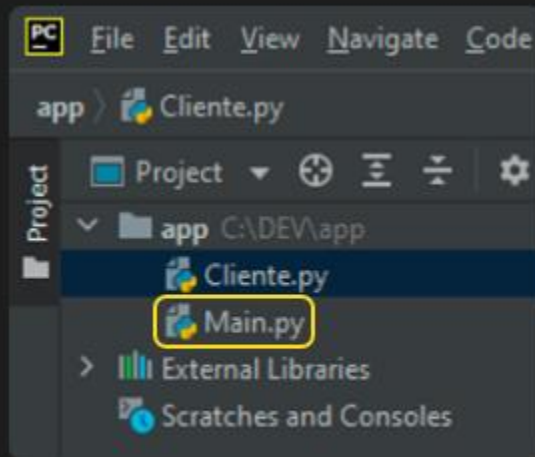
Assim, as instâncias de objetos são diferenciadas.

Esse **ID** diferencia objeto e atributos deste objeto.

Criando Referência de Classes

Para instanciar o objeto de uma classe para outra, devemos **criar a referência da classe** que será instanciada.

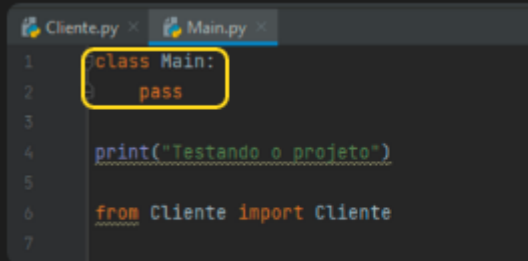
>>1. Primeiramente, abra a classe **Main.py** criada no início do projeto e disponível no menu.



Criando Referência de Classes

Para instanciar o objeto de uma classe para outra, devemos **criar a referência da classe** que será instanciada.

>>2. A classe **Main** não possui atributos.
Utilizamos apenas a palavra reservada **pass**.

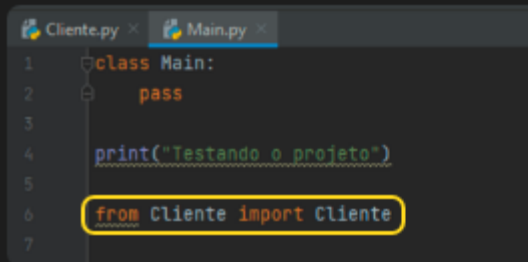


```
1 class Main:
2     pass
3
4     print("Testando o projeto")
5
6     from Cliente import Cliente
7
```

Criando Referência de Classes

Para instanciar o objeto de uma classe para outra, devemos **criar a referência da classe** que será instanciada.

>>3. Adicionamos a linha de código correspondente à importação da classe.



```
1 class Main:
2     pass
3
4     print("Testando o projeto")
5
6     from Cliente import Cliente
7
```


Criando Referência de Classes

Para instanciar o objeto de uma classe para outra, devemos **criar a referência da classe** que será instanciada.

>>4. Agora, vamos interpretar o código de importação da classe?
A palavra reservada **from** é utilizada para indicar qual classe será importada...

```
5  
6 from Cliente import Cliente  
7
```

Criando Referência de Classes

Para instanciar o objeto de uma classe para outra, devemos **criar a referência da classe** que será instanciada.

>>5. ... e a palavra reservada **import** é a referência de qual classe será utilizada para a criação de objetos em um arquivo à parte.

```
5  
6 from Cliente import Cliente  
7
```

Instanciando um Novo Objeto

>>1. Após criarmos a referência da classe, adicionamos, na linha seguinte, o novo objeto, passando por parâmetro os valores para inicialização dos atributos.

```
Cliente.py x Main.py x
1 class Main:
2     pass
3
4     print("Testando o projeto")
5
6     from Cliente import Cliente
7
8     c1= Cliente("João", "114444-2222")
9
```

Instanciando um Novo Objeto

>>2. A declaração de um novo objeto funciona como declarar uma nova variável.

```
1  
2  
3 c1 Cliente("João", "116444-2222")  
4
```

Instanciando um Novo Objeto

>>3. Passamos o nome da classe que será instanciada.

```
c1= Cliente("João", "116444-2222")
```

Instanciando um Novo Objeto

>>4. Passamos todos os atributos criados no Método Construtor da classe Cliente.

```
c1= Cliente("João", "114444-2222")
```

```
Cliente.py
1 class Cliente:
2     def __init__(self):
3
4         self.nome
5         self.telefone
6
```

Testando o Projeto

Para verificarmos a prática de instanciar objetos, vamos acrescentar os comandos para impressão e, em seguida, compilar o nosso código.

>>1. A primeira linha imprime a referência do objeto (id).
Já a segunda linha imprime o conteúdo adicionado.

```

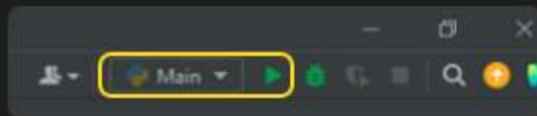
1 class Main:
2     pass
3
4     print("Testando o projeto")
5
6     from Cliente import Cliente
7
8     c1= Cliente("João", "114444-2222")
9
10    print(c1)
11    print(c1.nome, " e " + c1.telefone)
12

```

Testando o Projeto

Para verificarmos a prática de instanciar objetos, vamos acrescentar os comandos para impressão e, em seguida, compilar o nosso código.

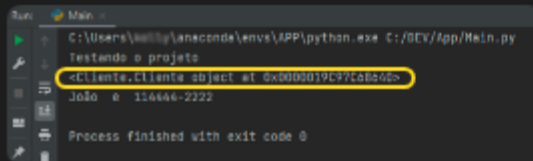
>>2. Para compilar, selecione a classe Main.
Em seguida, pressione o botão para compilar o código.



Testando o Projeto

Para verificarmos a prática de instanciar objetos, vamos acrescentar os comandos para impressão e, em seguida, compilar o nosso código.

>>3. Será apresentado o ID do objeto.



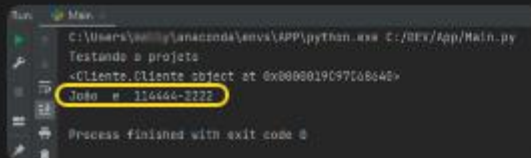
```

C:\Users\Welly\Anaconda\envs\APP\python.exe C:/DEV/app/Main.py
Testando o projeto
<cliente.cliente object at 0x0000019C371a5500>
João e 116444-2222
Process finished with exit code 0
  
```

Testando o Projeto

Para verificarmos a prática de instanciar objetos, vamos acrescentar os comandos para impressão e, em seguida, compilar o nosso código.

>>4. E será exibido o conteúdo do objeto.



```

C:\Users\joel\anaconda\envs\APP\python.exe C:/Dev/App/Main.py
Testando o projeto
<Cliente.Cliente object at 0x0000019C97C08840>
Nome = 116444-1222
Process finished with exit code 0
  
```

Projeto Controle Bancário

Vamos iniciar um projeto de controle bancário!

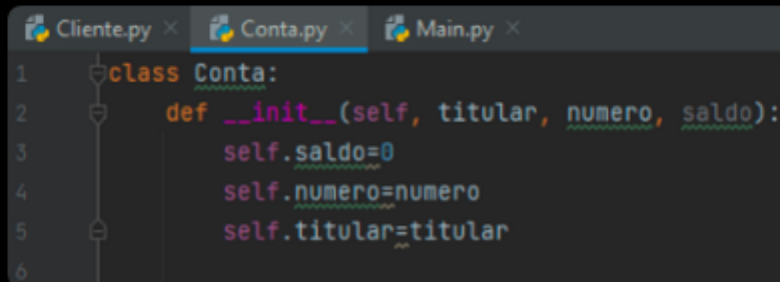
O objetivo do projeto é o desenvolvimento orientado a objetos para a execução de tarefas do cotidiano bancário, como saque, consulta de saldo e depósito.

Durante esta aula, já desenvolvemos a classe **Cliente** e seus atributos. Agora, vamos desenvolver a classe **Conta**, que será definida recebendo o objeto **Cliente**, além dos atributos "número" e "saldo".

Classe Conta

Para desenvolver a classe **Conta**, crie um novo arquivo Python, por meio do menu **File-New**. Na caixa de texto **New**, escolha a opção **Python File**. Digite o nome **Conta** e pressione a tecla **Enter**, para finalizar.

Logo após, adicione a codificação inicial para a classe:



```
1 class Conta:
2     def __init__(self, titular, numero, saldo):
3         self.saldo=0
4         self.numero=numero
5         self.titular=titular
6
```

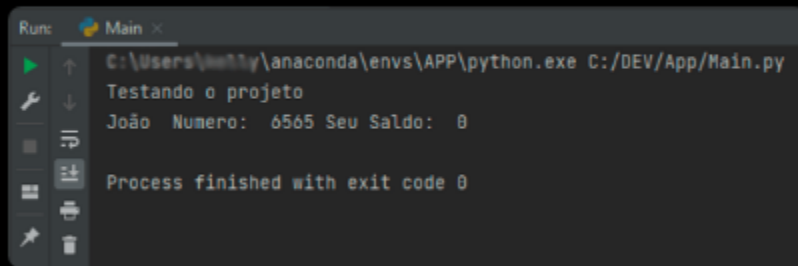
Testando o Projeto

Para verificar o funcionamento do projeto até o momento, modifique a classe **Main**, trazendo os dois objetos criados:

```
Cliente.py  Conta.py  Main.py
1 class Main:
2     pass
3
4     print("Testando o projeto")
5
6     from Cliente import Cliente
7
8     from Conta import Conta
9
10    c1 = Cliente("João", "114444-2222")
11    conta = Conta(c1.nome, 556$ 0)
12
13    print(conta.titular, " Número: ", conta.numero, " Seu Saldo: ", conta.saldo)
14
```

Analise o Resultado

Observe que, ao compilar o projeto (por meio da classe Main), as informações são impressas e a informação relacionada ao titular é trazida do objeto **Cliente**:



```
Run: Main x
C:\Users\melly\anaconda\envs\APP\python.exe C:/DEV/App/Main.py
Testando o projeto
João Numero: 6565 Seu Saldo: 0
Process finished with exit code 0
```



Encapsulamento de Dados

Encapsulamento de Dados

Uma das principais vantagens do conceito de orientação a objetos é a **utilização de estruturas sem a necessidade de conhecer como elas foram implementadas**.

Para isso, o conceito de **encapsulamento de dados torna-se essencial**, pois envolve a **proteção dos atributos ou métodos de uma classe**.

A ideia de encapsular o código vem com a premissa de proteger atributos e métodos de uma classe (tornando-os privados), de forma que somente a classe onde as declarações foram feitas tenham acesso.

Esse conceito garante a integridade das informações e também facilita a utilização das implementações.



O conceito de encapsulamento traz o **isolamento do código**, ou seja, variáveis e funções que são utilizadas internamente **não devem estar disponíveis externamente**.



Importante

Diferente da maioria das linguagens, como Java, PHP e C#, o Python (independente da nomenclatura), mantém todos os atributos e métodos públicos.

Isso não significa que todas as funções de uma classe podem ser chamadas por outras ou, principalmente, que todos os atributos podem ser lidos e alterados sem cuidados.

Para isso, na linguagem Python temos o que chamamos de **convenção** para aplicação destes conceitos de orientação a objetos.

É muito importante ressaltar que a maioria das IDEs (inclusive o PyCharm) oculta, automaticamente, atributos ou classes quando utilizamos a convenção de forma correta.

Modificadores de Acesso

De forma geral, todas as linguagens de programação que utilizam orientação a objetos usam modificadores de acesso para alterar a visibilidade de classes, atributos e métodos.

Para a implementação do encapsulamento é fundamental **alterarmos a visibilidade dos atributos de uma classe**. Para isso, utilizamos os **modificadores de acesso**.

Diferentemente de outras linguagens, como o Java e o C#, que utilizam palavras reservadas, a linguagem Python utiliza o símbolo *underscore* "_".

Dentro da orientação a objetos temos os modificadores **Public, Protected e Private**.

A seguir, vamos conhecer as principais características de cada um deles.

Public



Protected



Private





É o mais comum entre os modificadores.

Ele permite acesso tanto de dentro, quanto de fora de uma classe.

Sua implementação se dá por meio do uso do *underline* `"_"` na frente do nome.

Utilizando o modificador protegido, somente suas classes e subclasses terão acesso ao atributo ou método.

Para sua implementação adicione um *underline* `"_"` antes do nome.

É o modificador mais restrito do desenvolvimento orientado a objetos.

Ele permite que somente a sua classe (onde foi definido) tenha acesso a um determinado atributo ou método.

Para definir o método *private* adicionamos *underline* duplo `"__"` na frente do nome.

Visibilidade dos Membros



Um dos recursos mais importantes da orientação a objetos é o de **restringir o acesso às variáveis de um objeto e a alguns métodos**.

O objetivo principal desta ação é **evitar que variáveis internas sejam acessadas** e recebam valores diretamente ou, ainda, que métodos internos sejam invocados externamente, garantindo, assim, a integridade das informações.

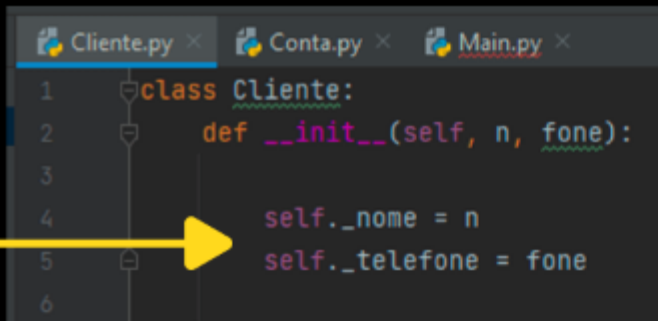
Para modificar a visualização de um membro dentro das IDEs utilizamos as convenções apresentadas para a linguagem.

Na prática

Modificando os Atributos

Para iniciar o processo de encapsulamento, vamos modificar os atributos das classes já criadas de forma que fiquem privados. Vamos iniciar pela classe **Cliente**.

Observe que foi adicionado o símbolo *underline* () antes da definição do nome do atributo.



```

1 class Cliente:
2     def __init__(self, n, fone):
3
4         self._nome = n
5         self._telefone = fone
6
  
```

Conferindo a Restrição de Acesso

Após a alteração do nome do atributo, podemos observar que a classe **Main** não visualiza mais o atributo diretamente.

```

10 c = Cliente("João", "114444-2222")
11 conta=Conta(c1, 6505, 0)
12
13 print(conta)
14
15

```

Seu Saldo: 0

exit code 0

Method	Class
__init__(self, n, fone)	Cliente
not	not expr
par	(expr)
__eq__(self, o)	object
__ne__(self, o)	object
__annotations__	object
__class__	object
__delattr__(self, name)	object
__dict__	object
__dir__(self)	object
__format__(self, format_spec)	object
__getattr__(self, name)	object

Press Enter to insert, Ctrl to replace

Métodos de Acesso (Get e Set)



Para permitir o acesso aos atributos de forma controlada, a prática mais comum é a utilização de dois métodos de acesso: um **retornando valor** e outro que **muda valor**.

Getters e **Setters** são usados na maioria das linguagens de programação orientada a objetos com o objetivo de garantir o princípio de encapsulamento de dados.

Os métodos são utilizados para implementações que alteram os valores internos da classe ou que retornam valores dela.

Get



Set



Sempre retornam valores.

O método **Get** é utilizado para ler os valores internos do objeto e enviá-los como valor de retorno da função.

Recebem valores por parâmetros.

Os métodos **Set** recebem argumentos que serão atribuídos a membros internos do objeto.

Sintaxe dos Métodos de Acesso

Get

`get_nome do atributo()`

Exemplo:

```
get_idade(self):return self._idade
```

Set

`set_nome do atributo
(valor por parâmetro)`

Exemplo:

```
def set_idade(self, valor):  
    self.idade=valor
```


Criando os Métodos de Acesso

Dando continuidade ao processo de encapsulamento, vamos desenvolver os métodos de acesso nas classes já criadas. Assim, vamos acessar os atributos privados. Vamos iniciar pela classe **Cliente**.

```
1 class Cliente:
2     def __init__(self, n, fone):
3
4         self._nome = n
5         self._telefone = fone
6
7     # método get
8     def get_nome(self):
9         return self._nome
10
11    # método set
12    def set_nome(self, nome):
13        self._nome = nome
14
```


Na prática, em Python (diferentemente da linguagem Java), o "_" (*underline*) antes do atributo não impede o acesso dele em outra classe, ou seja, **ele não fica privado**.

Essa forma é somente um indicativo de que os métodos nos quais os nomes iniciam com "_" (*underline*) não devem (mas podem) ser acessados.

Isso pode trazer problemas?



Em alguns casos pode sim! Por exemplo, na manipulação da classe **Conta**, o atributo **saldo** deve ser manipulado somente pelos métodos **Saque()** e **Depósito()** evitando, assim, que seja inserido um valor negativo no saldo.

Em alguns atributos é muito importante preservar o valor iniciado na classe, **não sendo possível realizar a inserção de qualquer valor no atributo, a não ser por meio de métodos**.

No caso do atributo **saldo**, por exemplo, ele não deve ficar negativo.

Alterando a Classe Conta

Para o atributo **saldo** não ser negativo, a utilização do método **setter** é justificável, ficando do seguinte modo:

```
Cliente.py x Conta.py x Main.py x
1 class Conta:
2     def __init__(self, titular, numero, saldo):
3         self.saldo=0
4         self.numero = numero
5         self.titular = titular
6
7     def get_saldo(self):
8         return self._saldo
9
10    def set_saldo(self, saldo):
11        if (saldo<0):
12            print("O saldo não pode ser negativo")
13        else:
14            self._saldo = saldo
15
```

Tradicionalmente, as linguagens de programação orientada a objetos relatam que atributos e métodos tem de ser separados basicamente em "público" e "privado".

A linguagem Java, em particular, sugere, por meio da sua sintaxe e práticas, que a maior parte dos atributos seja privada e, para sua manipulação, sejam criados os métodos, neste caso, os **getters** e **setters**.

Em Python, este conceito de "público e privado" não existe na sintaxe da linguagem.

O que temos em Python é a **convenção de estilo** que diz que nomes de atributos, métodos e funções iniciados com "_" (*underscore*) não devem ser usados por usuários de uma classe, só pelos próprios implementadores e que o funcionamento desses métodos e funções pode mudar sem aviso prévio.



Portanto, não é considerado errado, no Python, **deixar os atributos simplesmente como atributos de instância de forma simples**, onde qualquer usuário da classe pode ler ou alterar, sem depender de nenhum outro mecanismo.

No entanto, os métodos **getter** e **setter** podem ser utilizados com funcionalidades adicionais, conforme colocado no atributo **saldo** do nosso exemplo.

Protocolo de Descritores - Decorator

Um **decorator** é um padrão de projeto de software que permite adicionar comportamento a um objeto já existente, em tempo de execução, ou seja, agrega, de forma dinâmica, responsabilidades adicionais a um objeto.

Na prática, o decorator permite que atributos de uma classe tenham **responsabilidades**.

Um decorator **é um objeto invocável, uma função que aceita outra função como parâmetro (a função decorada)**.

O decorator pode realizar algum processamento com a função decorada e devolvê-la ou substituí-la por outra função.



@Property



A linguagem Python traz uma outra solução para manter os atributos privados, conhecida como **Property**.

A função Property é um Decorator e é utilizada para obter um valor de um atributo.

Basicamente, a função **Property** permite que você declare uma função para obter o valor de um atributo.

Podemos alterar a classe **Conta** utilizando **Property** da seguinte forma:

```
Cliente.py x Conta.py x Main.py x
1 class Conta:
2     def __init__(self, titular, numero):
3         self._saldo=0.0
4         self._numero = numero
5         self._titular = titular
6
7     @property
8     def saldo(self):
9         return self._saldo
10
11    @saldo.setter
12    def saldo(self, saldo):
13        if (saldo < 0):
14            print("saldo não pode ser negativo")
15        else:
16            self._saldo = saldo
17
```




Importante

Em Python, não é considerada uma boa prática criar uma classe e, logo em seguida, adicionar propriedades (property) para todos os atributos.

A função Property deve ser utilizada somente se você precisar da funcionalidade de transformar ou verificar um atributo quando ele é atribuído ou lido.



Strings

Obrigado

