Módulos, Noções de Orientação a Objetos, Tratamento de arquivos

Módulos, Pacotes e Bibliotecas

Um módulo, para python, consiste em um arquivo de código fonte (“.py”) que pode ser importado para um outro programa.

- Analogia: Trabalho em grupo, cada qual com seus conhecimentos, contribuem para efetuar o mesmo.

Cada módulo define suas próprias funções, variáveis e etc.. Os módulos são carregados pela já conhecida instrução “import”.

Já um pacote, consiste em uma coleção de módulos, indexados em um diretório. Para ser considerado um pacote válido, um diretório deve conter o arquivo “__init__.py”, ainda que vazio.

- Continuando a analogia: seria a sala, com várias equipes, cada qual com um trabalho.

Pacotes são úteis, especialmente para organização de artefatos de código.

Temos então as bibliotecas, que consistem em pacotes, módulos, arquivos “standalone” e outros tipos arquivos, todos eles agrupados e, de alguma forma, publicados para uso em tarefas de desenvolvimento de código.

- Finalizando a analogia: Seu professor corrige e aponta alguns trabalhos para serem apresentados durante a reunião de pais, por exemplo.

  Python, por ser Open Source, possui uma grande vantagem: Grande parte de suas bibliotecas se encontra disponível sem custos, para quem desejar usá-las. Isto sem falar da possibilidade de você mesmo criar e divulgar seus próprios artefatos de código.

Alguns módulos já vem “embutidos” na instalação padrão de python. Dentre eles, podemos destacar:

- math: Já citada em exemplos anteriores, serve para desenvolver programas que envolvam alguma faceta de matemática;
- os: Funcionalidades relativas ao sistema;
- time/datetime/calendar: Gerência de tempo;
- html: Manipulação de páginas web em formato HTML;

E muitos outros...

```python
import time # Importamos o modulo time (padrao na maioria das instalacoes)
#Vamos utiliza-lo para mensurar o tempo de execucao
atual = time.time() # Tempo atual

x = 0

'''Fazemos algum processamento aqui,
por exemplo, somar todo mundo de 0 a 999'''

for i in range(1000):
    x += i

print x

print time.time() - atual # Diferenca que representa o tempo decorrido
```

```python
import random

for i in range(5):
    print random.randint(0,10); # Gerar 5 aleatorios, entre 0 e 10

```

A exemplo, temos alguns módulos e bibliotecas bem populares, que comumente são englobados em programas desenvolvidos em python.

- O Numpy, que provê funcionalidades matemáticas.
- PyGame, com funcionalidades para trabalho com interfaces e modelagens n-dimensionais.
- Sqlite, para trabalhar com o SGBD homônimo.

Orientação à Objetos

A orientação à objetos (OO) é uma tecnologia que, atualmente, encontra-se difundida na maioria das linguagens de programação. Para se ter uma ideia, de acordo com o TIOBE, o paradigma OO esta presente em 9 das 10 linguagens mais populares atualmente (2/2017). Python é OO desde sua origem.

Como OO é um conteúdo relativamente extenso, iremos apenas trabalhar conceitos básicos. O ideal é que você curse a disciplina "Programação Orientada a Objetos" para realmente aprender os conceitos de forma "mais refinada" (ainda que não especificamente, sobre python). Por fim, não se preocupe se os conceitos aqui parecerem, a princípio, um tanto quanto confusos.

Talvez, como motivador, podemos citar o fato que o paradigma OO é indispensável de ser estudado, caso você busque entrar para o mercado de trabalho da programação. Do TIOBE:

Apr 2018	Apr 2017	Change	Programming Language	Ratings	Change
1	1		Java	15.777%	+0.21%
2	2		C	13.589%	+6.62%
3	3		C++	7.218%	+2.66%
4	5	^	Python	5.803%	+2.35%
5	4	v	C#	5.265%	+1.69%
6	7	^	Visual Basic .NET	4.947%	+1.70%
7	6	v	PHP	4.218%	+0.84%
8	8		JavaScript	3.492%	+0.64%
9	-	^^	SQL	2.650%	+2.65%
10	11	^	Ruby	2.018%	-0.29%

OO, na prática, significa realizar a representação de objetos do mundo real, através de código. Para tanto, existe toda uma teoria e estruturação, já bastante consolidada.

A começar, o que seria um objeto do mundo real?

- Na realidade, objeto aqui assume diversos sentidos, não só de coisas, mas também, pessoas, entidades e algumas vezes, situações. Pode ser tanto um objeto concreto, como um objeto “fusca”, “tom_cruise” ou “girafa”, quanto um objeto abstrato, como “classificacao_do_aluno” ou “data”.

Classes

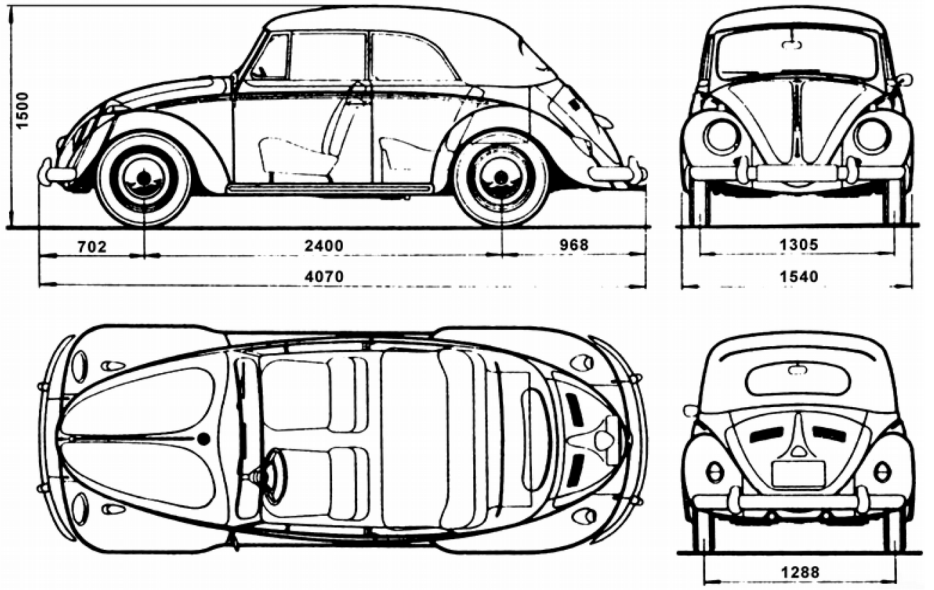
Faremos uso aqui, de um exemplo cunhado por Deitel em seu livro sobre programação em Java.

É muito comum que pessoas possuam carros. No entanto, para que seu carro pudesse ser construído, o mesmo teve de ser projetado. Algum engenheiro teve de pensar: “Quantas portas terá? ” ou “O que deve ocorrer quando pisa-se no acelerador?”.

Agora pensando em outro ponto: Este “projeto” pode servir não somente para um carro, mas, quanto mais generalista for, mais carros o mesmo poderá representar.

Isto pode ser considerado como uma analogia à classes.

Uma classe é uma entidade que representa um conjunto de objetos com características em comum. É um modelo, que serve para criar objetos.



Declarando uma classe:

```
1
2   class nome_da_classe (): # Definicao de uma classe
3
4       #Atributos e Metodos aqui no corpo
5
6       pass # Como nao ha nada no exemplo...
7
8   # Um exemplo mais comum
9
10  class carro ():
11
12      def __init__(self): # Este eh um metodo que possui uma peculiaridade
13          pass
```

Objetos

Como dito anteriormente, temos que um objeto é uma representação virtual de alguma entidade. Porém, é muito mais que isso. Os objetos também são instâncias de uma classe. Objetos são a base de toda a teoria de OO.

- A saber, instância significa uma “ocorrência concreta”.

A descrição de um objeto se dá por sua classe e pelas informações à ele atribuídas.

Instanciação

Retomando o exemplo do carro, temos um projeto pronto (classe). No entanto, para sairmos dirigindo nosso carro, primeiro ele deve ser “construído” (atenção com essa palavra!).

Lógica: O que você pilota, o projeto do carro ou o carro em si?

Para podermos efetivamente usufruir de tudo que foi definido na classe (seus métodos e etc.), devemos instanciá-la, ou seja, criar um objeto através dela.


BRIDGESTONE
Marlboro


vodafone
BRIDGESTONE
Marlboro

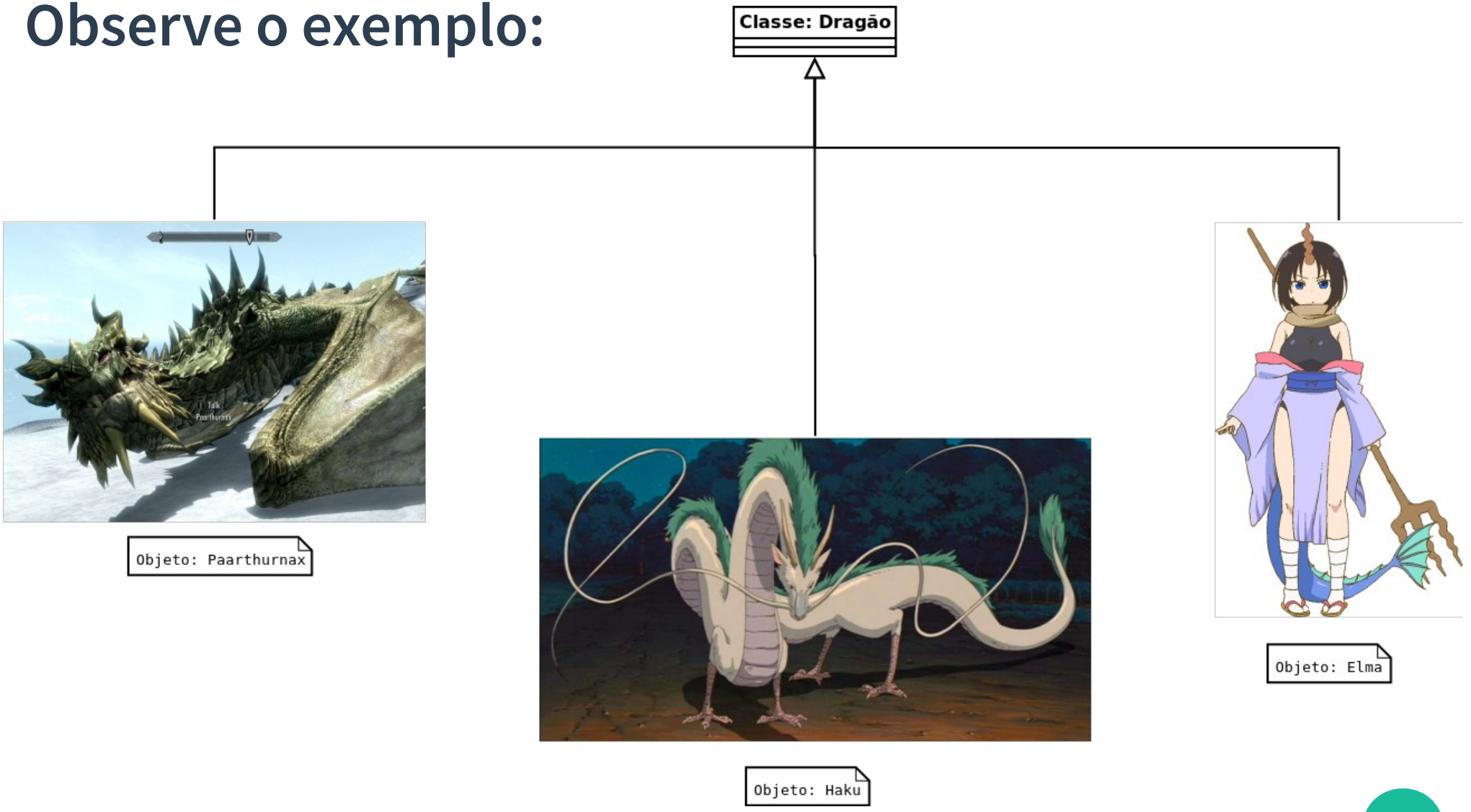

565

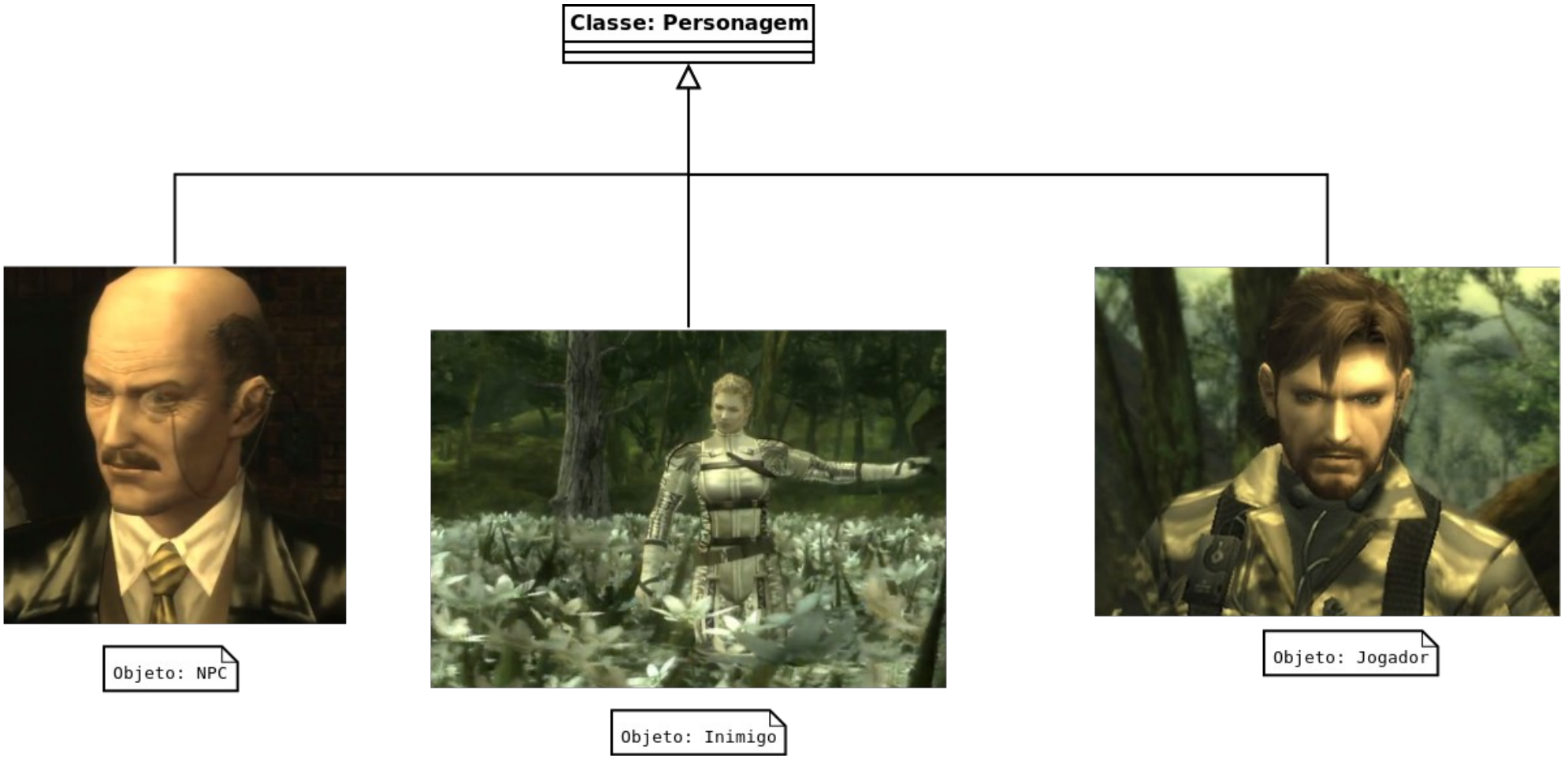
Repetindo, objetos são instâncias de uma classe.
Em python, podemos criar um objeto fazendo:

```
1  class Empregado(): # Declaracao da classe "Empregado"
2      pass
3
4  objeto_empregado = Empregado() # Chamada e criacao de um objeto
5
6  #Note, no entanto, que este exemplo nao ira realizar absolutamente nada.
7
```

Vamos ver algumas classes e instâncias para “clarear” nossa visão:

Observe o exemplo:




Classe: Personagem
Objeto: NPC
Objeto: Inimigo
Objeto: Jogador

Atributos

Um objeto pode ter características específicas (suas qualidades). A isto, damos o nome de atributos.

```
1   class Pessoa(): # Classe
2       def __init__(self, nome, idade, hobby): # Chamada do metodo
3           # Abaixo, teremos a definicao dos atributos
4           self.nome = nome # Atributo nome
5           self.idade = int(idade) # Atributo idade
6           self.hobby = hobby # Atributo hobby
7           #Talvez esteja curioso com o porque deste "self", nao?
8
9           #Todos atributos, representando alguma caracteristica de pessoa
10
11      def apresenta(self): # Nao se preocupe com isto ainda
12          print "Sou ", self.nome, ", tenho ", self.idade
```

- No código, atributos são representados de forma bem similar à variáveis (Porém não devemos confundir os conceitos!).

```
1   class Carro():
2
3       # Este __init__ sera chamado ao criarmos o objeto. Detalhes a frente.
4       def __init__(self, marca, modelo, motor, proprietario):
5           self.marca = marca
6           self.modelo = modelo
7           self.motor = motor
8           self.proprietario = proprietario
9
10  # Criando o objeto a partir da classe "Carro"
11  fuscao = Carro("Volkswagen", "80 - Off-Road", "V8", "Popretariu")
12
13  print fuscao.marca # Acesso ao atributo "marca". Imprime "Volkswagen"
14  print fuscao.modelo
15  print fuscao.motor
16  print fuscao.proprietario
17
```

Não há, praticamente, restrição quanto ao tipo assumido por um atributo. Isto implica que atributos podem ser até mesmo, objetos gerados por outras classes.

```
1  class Pessoa():
2      def __init__(self, nome):
3          self.nome = nome
4
5  class Filme():
6
7      def __init__(self):
8          self.protagonista = Pessoa("Leonardo Dicaprio") # Chamando a outra classe
9          self.protagonista2 = Pessoa("Tom Hanks")
10
11 filme_top = filme()
12
13 print filme_top.protagonista.nome # Observe como fazemos o acesso aqui
14
```

Atributos não devem ser confundidos com “variáveis de escopo classe” (também chamadas Static Class Variables, SCV). Diferente de um atributo, as SCVs são compartilhadas por todas as instancias de uma classe (objetos):

```
1  class Aluno():
2      quantidade_alunos = 0 #Variavel estatica, inicializada com 0
3      def __init__(self, nome):
4          self.nome = nome # Um atributo "comum"
5          Aluno.quantidade_alunos += 1 #Observe que o acesso eh diferenciado
6
7  aluno1 = Aluno("Ada")
8  print "De aluno1: ", aluno1.quantidade_alunos #Altera para todos os membros da classe
9  aluno2 = Aluno("Alan")
10 print "De aluno2: ", aluno2.quantidade_alunos
11
12 print "De aluno1: ", aluno1.quantidade_alunos
```

Métodos

Como métodos, entendemos “tudo aquilo que um objeto é capaz de fazer”. Note que isto se diferencia bastante das características de um objeto (atributos).

São úteis pois permitem descrevermos instruções para atuarem com, sobre ou para nossos objetos.

```
1  class Aluno ():
2      def __init__(self, nome): # Isto eh um metodo
3          self.nome = nome
4          self.escola = "Escola Python"
5
6      def estuda(self): # Isto eh outro metodo. Observe a sintaxe, similar a funcoes
7          print self.nome, " esta estudando no momento"
8
9
```

- São bastante similares à funções, porém novamente, não confunda os conceitos!

```
1  class Aluno ():
2      def __init__(self, nome): # Isto eh um metodo
3          self.nome = nome
4          self.escola = "Escola Python"
5
6      def estuda(self): # Isto eh outro metodo. Observe a sintaxe, similar a funcoes
7          print self.nome, " esta estudando no momento"
8
9
10
11 aluno1 = Aluno("Beatrix Kiddo")
12 aluno1.estuda() # Chamada de metodo, novamente, olhe a sintaxe
```

Exercício #18 (5 min.)

(Analítico) Dadas as classes a seguir, relate pelo menos 3 objetos que podem ser declarados para cada:

- Classes: Peixe; Moto; Cenário;

Dados os objetos abaixo, agrupe os mesmos (sem exclusões) em no máximo, 2 classes distintas. A seguir, ilustre para cada classe, 2 atributos e 2 métodos.

- Objetos: machado; marceneiro; cozinheiro; canivete; martelo; motorista; professor; faxineiro; advogado; alicate; policial;

Construtor

Lembram-se de nosso “Método Especial”? Há um motivo para termos mencionado isto. Ele é denominado Construtor, e ele é o método que é invocado toda vez que um novo objeto pertencente àquela classe for criado.

Alguns autores também o chamam método inicializador ou instanciador. Por padrão, toda classe em Python possui um “construtor abstrato”.

Por padrão, todo Construtor em Python deve ser nomeado “__init__()”, e, opcionalmente, pode conter quaisquer parâmetros o programador desejar (incluindo o nosso amigo “self”).

- Lembre-se bem disto: O Construtor não cria o objeto, mas é chamado quando o objeto será criado.

```
1  class Personagem():
2      def __init__(self, nome, classe, hp): # Construtor
3          self.nome = nome
4          self.classe = classe
5          self.hp = int(hp)
6      def acao (self, narracao):
7          print self.nome, " realizou: ", narracao
8
9  p1 = Personagem("Bill", "Espadachim", 100) # O construtor eh chamado neste momento!
10 p2 = Personagem("Beholder", "Monstro", 200)
11
12 p1.acao("Retira espada da bainha")
```

self, para que?

Pode parecer um pouco estranho esta palavra “self”, como atributo do método, principalmente se você já possui algum entendimento sobre outra linguagem OO, como o Java.

Esta variável “self” representa o acesso ao objeto, já instanciado, que está sendo “usado” no momento. Este acesso pode ser usado, por exemplo, para operar sobre o próprio objeto em algum método de sua classe.

Na realidade, o nome “self” é apenas uma convenção. Qualquer que seja o nome, desde que seja passado como o primeiro, no caso de um método, será aceito, Mas, por questão de padronização, a maioria dos programadores prefere usar a palavra “self”. Nós faremos o mesmo.

```
1  class Pessoa():
2      def __init__(self, nome, idade): # Observe que eh o primeiro parametro
3          self.nome = nome # self representa a declaracao de nome para a instancia em questao no momento
4          self.idade = idade
5      def apresenta (self):
6          print "Eu sou", self.nome # Acesso ao atributo nome da instancia que chamar este metodo
7
8  p1 = Pessoa("Von Neumann", 33)
9
10 p1.apresenta()# Nao especificamos o self ao chamar o metodo (ja sabiamos disso...)
```

- Em Java, por exemplo, temos uma palavra reservada com uma funcionalidade parecida (não a mesma): “this”. Para aqueles com algum conhecimento em Java, podem pensar no uso do self como o do this.

```java
package teste;

public class Pessoa {

    private String nome;
    private int idade;

    public Pessoa(String nome, int idade) { // Este é um construtor em Java
        this.nome = nome; // Observe o "this"
        this.idade = idade;
    }

    public void apresentacao() {
        System.out.println("Eu sou " + this.nome);
    }



    public static void main(String[] args) {

        Pessoa p1 = new Pessoa("James Gosling", 62);
        p1.apresentacao();


    }

}
```

Dissecação de uma classe

Exercício #19 (10 min.)

Vamos agora declarar a classe “Veículo”. A mesma deve servir para criar qualquer tipo de veículo, para qualquer ambiente. Para isto, será necessário que você pense num veículo da forma mais genérica possível.

- A mesma deve ter, no mínimo, 3 métodos e 3 atributos.

A seguir, declare 3 objetos diferentes da classe “Veículo” e invoque métodos de cada um deles.

Os pilares da Programação O.O.

A orientação a objetos vai muito além dos conceitos ilustrados. É, por si, uma forma de percepção e abstração do mundo ao nosso redor.

OO toma por base alguns conceitos específicos, empregados por definição. Tais conceitos constituem os chamados “pilares de OO”.

São eles:

- Herança;
- Encapsulamento (não será trabalhado em detalhe no momento);
- Polimorfismo (não será trabalhado em detalhe no momento);

Herança

Tal qual a palavra diz, herança é um artifício pelo qual classes irão ser capazes de “passar recursos a suas classes filhas”. O objetivo disso é prover reaproveitamento de artefatos de código.

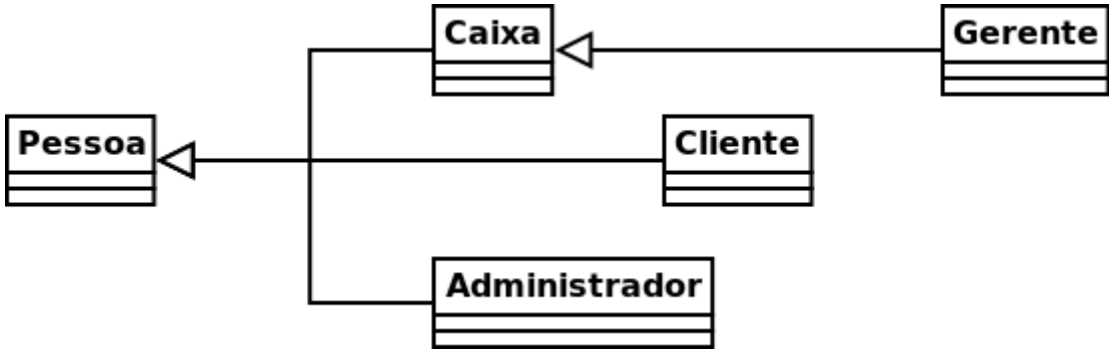
Como terminologias novas, temos:

- Classe Pai: Classe que está diretamente acima na hierarquia.
- Classe Filha: Aquela que herda de uma classe pai. Bem simples, não?

Os recursos passados por herança podem ser métodos ou atributos.

A herança é um recurso que permite a especialização de classes. A ideia aqui é manter alguma funcionalidade “comum” implementada em uma classe pai, sendo que suas filhas também possuirão tal funcionalidade.

Um exemplo de herança:

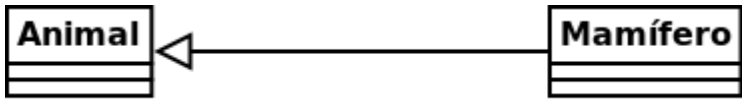


Demonstração:

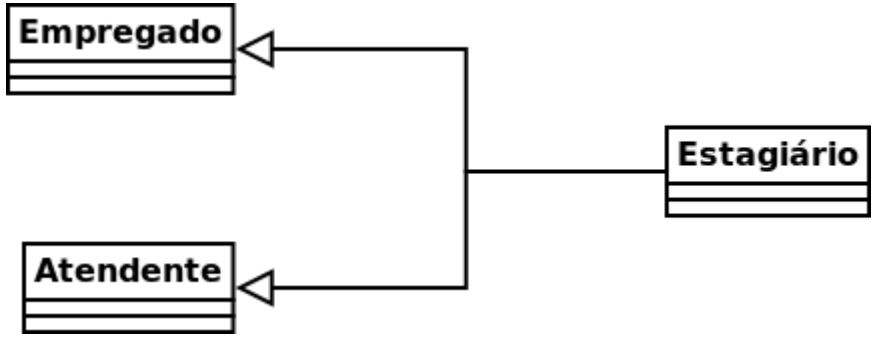
```
1  class Conta(): # Classe Conta: operacoes bancarias genericas...
2      def __init__(self, nome, dinheiro):
3          self.nome = nome
4          self.dinheiro = dinheiro
5      def deposito(self, valor):
6          self.dinheiro += valor
7      def saque(self, valor):
8          self.dinheiro -= valor
9
10 class ContaCorrente(Conta): # ContaCorrente "herda" de Conta. Heranca possui esta sintaxe.
11     def __init__(self, nome, dinheiro, nome_conjunto):
12         Conta.__init__(self, nome, dinheiro) # "Super" interessante, nao eh?
13         # A linha acima invoca o construtor de "Conta", sobre este objeto (vide self)
14         self.nome_conjunto = nome_conjunto
15     def responsaveis(self):
16         print self.nome_conjunto, " - ", self.nome
17
18 #Neste exemplo, "Conta" eh a classe pai e "ContaCorrente" eh a classe filha
19
20 c1 = ContaCorrente("Milionario", 1000, "Ze Rico")
21
22 c1.responsaveis() # Metodo de ContaCorrente
23
24 print c1.dinheiro # Este atributo eh da classe Conta
25 c1.deposito(333) # Este metodo tambem eh da classe Conta
26 print c1.dinheiro
```

Simples X Múltipla

- Herança Simples:



- Herança Múltipla:



Sobrescrevendo métodos

Se uma classe filha herda todas as funcionalidades da classe pai, o que acontece quando temos esta situação:

```
1  class Conta(): # Classe Conta: operacoes bancarias genericas...
2      def __init__(self, nome, dinheiro):
3          self.nome = nome
4          self.dinheiro = dinheiro
5      def deposito(self, valor):
6          self.dinheiro += valor
7      def saque(self, valor):
8          self.dinheiro -= valor
9
10 class ContaCorrente(Conta): # ContaCorrente "herda" de Conta. Heranca possui esta sintaxe.
11     def __init__(self, nome, dinheiro, nome_conjunto):
12         Conta.__init__(self, nome, dinheiro) # "Super" interessante, nao eh?
13         # A linha acima invoca o construtor de "Conta", sobre este objeto (vide self)
14         self.nome_conjunto = nome_conjunto
15     def responsaveis(self):
16         print self.nome_conjunto, " - ", self.nome
17     def saque(self, valor):
18         print "Saque de Conta Corrente"
19         print "Aviso: eh necessario aprovacao de ambos os correntistas" #Este eh diferente
20         self.dinheiro -= valor
21
22 #Neste exemplo, "Conta" eh a classe pai e "ContaCorrente" eh a classe filha
23
24 c1 = ContaCorrente("Milionario", 1000, "Ze Rico")
25
26 c1.responsaveis() # Metodo de ContaCorrente
27
28 print c1.dinheiro # Este atributo eh da classe Conta
29 c1.deposito(333) # Este metodo tambem eh da classe Conta
30 print c1.dinheiro
31
32 c1.saque(100)
33 print c1.dinheiro # E ai, o que sera que vai acontecer? Spoiler: Isto funciona, e eh bem util...
```

A classe Object

Existe uma classe considerada “elementar” na OO em python. Ironicamente (ou não) ela se denomina classe “object”.

A partir do Python 3, toda classe herda implicitamente (“por padrão”) de object. Em versões anteriores, no entanto, isto precisa ser declarado explicitamente.

Uma boa prática em Python e que vários programadores adotam é fazer com que as classes mais elementares (os que servem de base para outras e que não herdam de ninguém) herdem da classe “Object”.

Ou seja: a partir de agora, em toda classe elementar que tivermos, lembre-se de object.

Exercício #20 (15 min.)

Crie a classe “Animal” com os seguintes métodos e atributos:

- Atributos: nome (String); idade (Inteiro); eh_amigavel (Booleano);
- Métodos: fugir (); atacar ();

Ambos os métodos devem imprimir textos na tela.

A seguir, crie as classes “Mamífero” e “Ave”, que herdarão de “Animal”. Elas deverão possuir pelo menos 1 método próprio. Seja criativo!

Finalmente, declare um objeto de cada classe, “Mamífero” e “Ave”, e invoque os métodos fugir, atacar e o seu próprio método criado.

Encapsulamento

A encapsulamento, atribuímos a definição do “ato de encapsular” (esconder) coisas. Em uma analogia bem simples, temos as cápsulas de remédios.

Encapsulamento trabalha com o conceito de níveis de acesso em código. Na prática, tais níveis podem ser:

- Público (Public): Todos vêem;
- Privado (Private): Somente aqueles “com algo em comum” vêem;
- Protegido (Protected): Intermediário entre os dois;

Uma “confusão” comum: Tais níveis representam a questão do acesso para o desenvolvedor, em teoria, não influindo na questão da segurança do sistema.

Em Python, o encapsulamento é implementado de uma forma um tanto quanto subjetiva. Utilizamos de algumas convenções para o mesmo.

A principal delas, sendo o uso dos “Double underscores” (__);

Uma outra forma na qual o encapsulamento é implementado em python, esta considerada uma melhor prática, é através dos chamados “Métodos Descritores”.

Não iremos entrar mais à fundo neste assunto, para evitarmos exceder a complexidade deste curso. Em um próximo momento de estudo iremos trabalhar isto.

Polimorfismo

Polimorfismo pode ser entendido como a propriedade que algo tem de assumir várias formas.

Na apresentação da “Herança”, vimos um exemplo de comportamento polimórfico.

```
1  class Conta(): # Classe Conta: operacoes bancarias genericas...
2      def __init__(self, nome, dinheiro):
3          self.nome = nome
4          self.dinheiro = dinheiro
5      def deposito(self, valor):
6          self.dinheiro += valor
7      def saque(self, valor):
8          self.dinheiro -= valor
9
10 class ContaCorrente(Conta): # ContaCorrente "herda" de Conta. Heranca possui esta sintaxe.
11     def __init__(self, nome, dinheiro, nome_conjunto):
12         Conta.__init__(self, nome, dinheiro) # "Super" interessante, nao eh?
13         # A linha acima invoca o construtor de "Conta", sobre este objeto (vide self)
14         self.nome_conjunto = nome_conjunto
15     def responsaveis(self):
16         print self.nome_conjunto, " - ", self.nome
17     def saque(self, valor):
18         print "Saque de Conta Corrente"
19         print "Aviso: eh necessario aprovacao de ambos os correntistas" #Este eh diferente
20         self.dinheiro -= valor
21
22 #Neste exemplo, "Conta" eh a classe pai e "ContaCorrente" eh a classe filha
23
24 c1 = ContaCorrente("Milionario", 1000, "Ze Rico")
25
26 c1.responsaveis() # Metodo de ContaCorrente
27
28 print c1.dinheiro # Este atributo eh da classe Conta
29 c1.deposito(333) # Este metodo tambem eh da classe Conta
30 print c1.dinheiro
31
32 c1.saque(100)
33 print c1.dinheiro # E ai, o que sera que vai acontecer? Spoiler: Isto funciona, e eh bem util...
```

Como você vê o mundo?

Nome estranho para um tópico de programação, não acha?

Na verdade, com este tópico pretendemos finalizar nossos estudos sobre O.O. em python. Faremos isso estudando como deve ser feita a análise.

- A começar, não existe uma “fórmula mágica” para programação/desenvolvimento/etc., isto são coisas que dependem da pessoa.

Duas formas de visão são consideradas mais comuns para a modelagem de um problema no paradigma O.O..

- Top-Down;

- Bottom-Up;

Top-Down:

- Começar com foco no todo, dividir para conquistar;
- Decomposição;
- Hacker Way;
- Dificuldade: Mudanças na visão podem surgir (e desaparecer) ao longo do desenvolvimento;

Bottom-Up:

- Dividir em partes (tão elementares quanto possível), começar pelas partes, passo a passo;
- Composição;
- Creator Way;
- Dificuldade: Combinar pode tender ao caos;

I/O e Arquivos

Até agora, todo programa que desenvolvemos ao longo de nosso estudo sobre python trabalhou apenas com dados existentes em seu código (hardcoded) e com informações de input do usuário. Agora, iremos aprender a como trabalhar com arquivos externos ao programa.

Em python, podemos evocar arquivos através de objetos do tipo “file”, que comportam diversas operações sobre arquivos. Para que um arquivo possa ser trabalhado, o mesmo deve ser “aberto” dentro do programa. Isto pode ser feito através do método “open(“nome”, “forma”)” Arquivos podem ser abertos como:

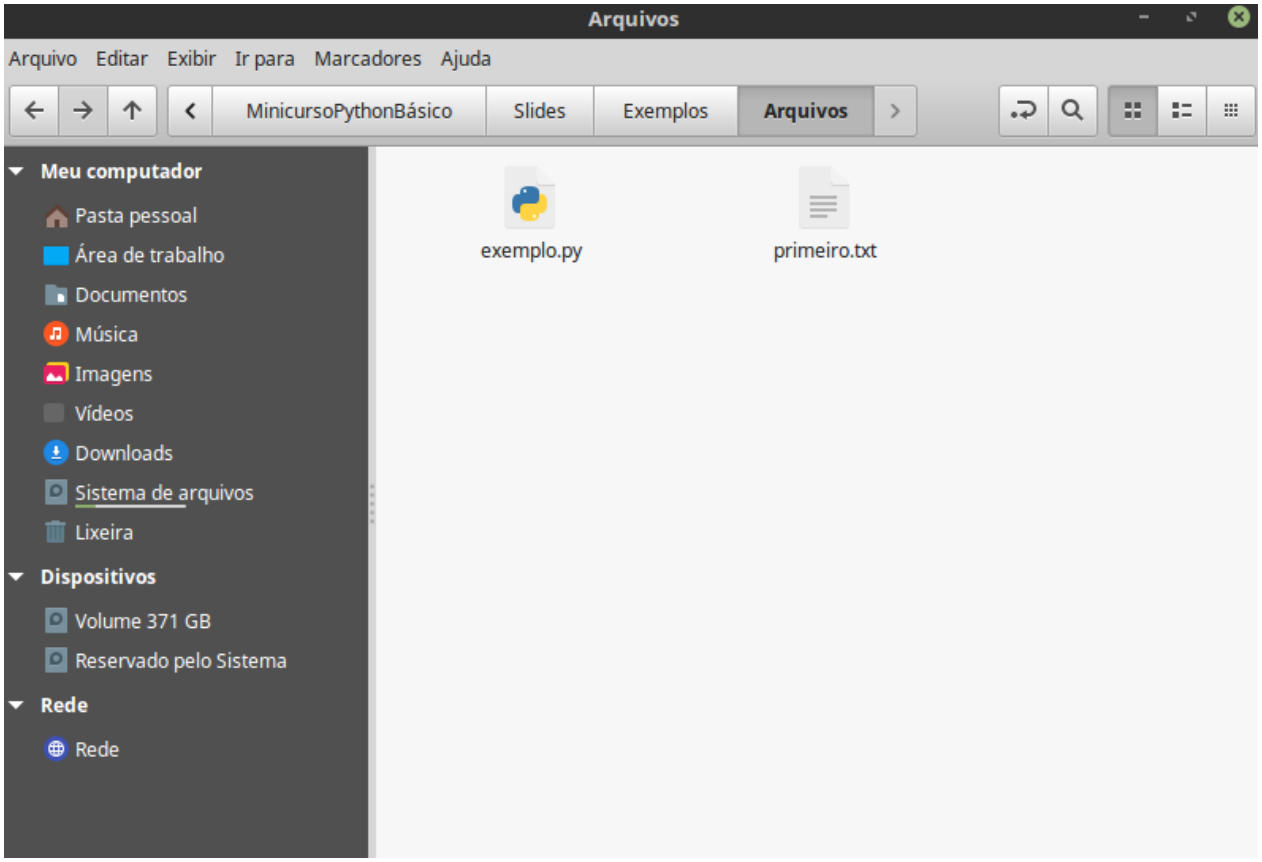
- Leitura (“r”, read);
- Gravação (“w”, write);
- Adição (“a”, append);
- Combinações;

Um exemplo simples :

```
1  primeiro_arquivo = open("primeiro.txt", "w") # Abre o arquivo para escrita (Write)
2  primeiro_arquivo.write("Este texto sera escrito dentro do arquivo!") #Escreve dentro do arquivo
3  primeiro_arquivo.close() #Importante!
4
```

Ainda, sobre o exemplo:

- Método “open()” trata de incorporar o arquivo no fluxo do programa.
- Por padrão, os arquivos são abertos em modo “somente leitura” (r). Em nosso caso, no entanto, especificamos que desejávamos escrita.
- Ao executar, será criado o arquivo “primeiro.txt”. Por padrão, será criado na mesma pasta onde o script está sendo localizado.
- O método “close()” sempre deve ser executado ao terminar alguma manipulação de arquivo, para podermos liberar memória. Não fazer isto é uma péssima prática de programação.




```
Este texto sera escrito dentro do arquivo!
```

- Um exemplo mais prático: Gerador de Cartão de Visita ASCII.

```
1  cartao = open("cartao.txt", "w")
2  cartao.write("#####################################" + "\n")
3  cartao.write("Nome: " + raw_input("Digite seu nome: ") + "\n")
4  cartao.write("Profissao: " + raw_input("Digite sua profissao: ")+ "\n")
5  cartao.write("Lema: " + raw_input("Digite seu lema: ")+ "\n")
6  cartao.write("#####################################"+ "\n")
7  cartao.close()
```

```
antonio@natsuki ~/Documentos/Oficina/Educacional/Minicursos/MinicursoPythonBasico/Slides/Exemplos/Arquivos $ python cartao.py
Digite seu nome: Irineu Prainha
Digite sua profissao: Salva-Vidas
Digite seu lema: Meu escritorio eh na praia, eu to sempre na area...
```

```
1  ######################################
2  Nome: Irineu Prainha
3  Profissao: Salva-Vidas
4  Lema: Meu escritorio eh na praia, eu to sempre na area...
5  ######################################
```

Integração com o sistema operacional

Em nosso dia-a-dia profissional, muitas vezes nos depararemos com problemas que irão requerer que trabalhemos com apoio do sistema operacional. Para isto, python nos provê com o módulo “os”.

- Um possível exemplo de uso para funções do S.O.: Gerenciamento de arquivos em massa.

Algumas das funcionalidades que este módulo nos provê:

- Execução de comandos do S.O.;
- Operações sobre arquivos e diretórios;
- Informações do sistema;
- Operações sobre processos;

```
1  import os
2
3  def detalhesRede():
4      if(os.name == "posix"): #os.name, retorna detalhes quanto ao S.O. usado
5          os.system("ifconfig")#os.system, executa um comando
6      else:
7          os.system("ipconfig")
8
9  detalhesRede()#Voce conseque definir qual a funcionalidade deste programa?
```

Dúvidas?
Sugestões?
Críticas?