

Linguagem de Programação II

Prof. ME Fernando Sousa

Gerente de Sistemas Pleno em Data Science e Big Data, atualmente trabalha desenvolvendo soluções para coleta, armazenamento, processamento de dados e extração de informação. Atua também como professor de graduação e pós-graduação na FIT – Faculdade Impacta de Tecnologia e como tutor de conteúdo no Curso de Especialização em Informática em Saúde da UNIFESP.

Bacharel em Informática Biomédica pela USP e Mestre em Ciências na área de Gestão e Informática em Saúde pela UNIFESP, com trabalho sobre mineração de textos e recuperação de informação em conteúdos de saúde. Atua com análise e desenvolvimento de soluções de TI desde 2009. Possui experiência em informática biomédica/ informática em saúde, reconhecimento e classificação de padrões, extração e recuperação de informação, aprendizado de máquina, mineração de textos, sistemas de informação em saúde, saúde pública, análise e desenvolvimento de sistemas (Java, C++, php, C#, Python), desenvolvimento Web, desenvolvimento para dispositivos móveis, e bancos de dados, big data e data science.

Agradecimentos:

Agradecemos os Professores Me Leonardo Takuno, Me. Renan Rodrigues, Prof. Dr. Renato Rodrigues Oliveira da Silva e Prof. Esp. Vilson Ferreira que colaboraram com a elaboração da disciplina de Linguagem de Programação 2 e com a elaboração desta apostila.



O professor



1	Introdução	6
2	Ambiente de Desenvolvimento	6
3	Testes Unitários	11
	3.1 Primeiros Testes	13
4	Dicionários	15
	4.1 Objetivos	16
	4.2 Conceitos	16
	4.3 Manipulação de dicionários	18
	4.4 Inserindo Complexidade	21
5	Modularização	22
	5.1 Importar módulos	22
	5.2 Criar módulos	25
6	Objetivos em Python	28
	6.1 Introdução:Linguagens de Programação	28
	6.2 Objetos em Python	30
7	Programação Orientada a Objetos	34
	7.1 Classes em Python	34
	7.2 Modelagem Orientada e objetos	36
	7.2.1. Modelando uma televisão em Python	38
	7.2.2. Construtores	41
	7.2.3. Atributos Privados	42
	7.2.4. Métodos com parâmetros	43
	7.2.5. Considerações	44
	7.3. Exemplo 2 - Modelando Professor, aluno e Disciplina	45
	7.3.1. Associação de classes	47
8.	Herança e Polimorfismo	50
	8.1. Encapsulamento	51
	8.2. Herança	52
	8.3. Metodologias mais utilizadas.	53
	8.4. Polimorfismo	54
	8.5. Herança e Polimorfismo em Python	55

9	Padrões de Projeto	56
	9.1 Padrões de Criação	57
	9.1.1. Factory	58
	9.1.2. Singleton	60
	9.2 Padrões de Estrutura	60
	9.2.1. Adapter	60
	9.1.2. Facade	61
	9.1.3. Decorator	62
	9.3 Padrões de Comportamento	62
	9.3.1. Observer	63
10	Arquivos	
	10.1. Objetivos	64
	10.1.1. Conceitos	64
11	Tratamento de Exceções	
	11.1. Tipos de Erros	69
	11.2. Tratamento de erros no Programa	69
	11.3. Exceções	70
	11.4. Entendendo as Exceções	71
	11.5. Tratando Exceções	72
11	Bibliografia	79

1. Introdução

A disciplina de Linguagem de Programação 2 é uma continuação do que vocês viram em Linguagem de Programação 1. Nesta disciplina vocês conhecerão mais estruturas de programação utilizadas em Python (e que podem ser aplicadas em outras linguagens de programação) para poder resolver problemas ainda mais complexos, utilizando lógica e Python. Tudo que você aprendeu em Linguagem de Programação 1 será aproveitado aqui: variáveis, strings, números, estruturas de decisão, estruturas de repetição, listas, tuplas e funções. Tudo isso é na realidade a base para todas as disciplinas de programação que terão no curso. Não será diferente para essa disciplina. Utilizaremos todos os conceitos aprendidos, mas agora aplicando principalmente em Programação Orientada a Objetos, um paradigma de desenvolvimento de software que faz um paralelo com as coisas do mundo real para os programas. Você aprenderá como criar diferentes módulos no seu sistema que fazem tarefas específicas e podem ser reutilizados, deixando seu código ainda mais modular e de fácil manutenção. Além disso, será introduzida a cultura de testes de software, uma parte importante do desenvolvimento que visa diminuir ao máximo os erros e aumentar a qualidade. Esta disciplina é fundamental para sua formação como analista e desenvolvedor de sistemas. Entender como estruturar programas modelando com orientação a objetos, entender como variáveis objetos funcionam em uma linguagem de programação, como utilizar padrões de projeto, como testar seus módulos, utilizar um banco de dados relacional integrado com sua solução e como utilizar outras estruturas importantes da linguagem Python serão um diferencial na sua carreira dentro da área de TI.

2. Ambiente de Desenvolvimento

Neste capítulo vamos fazer as configurações iniciais do ambiente de desenvolvimento que será utilizado na disciplina: Python com Visual Studio Code

✓ Configurar o ambiente com Python e Visual Studio Code (IDE);

✓ Utilizar a IDE para criar e executar um programa simples.

A disciplina de Linguagem de Programação 2 utiliza como base a linguagem de programação Python. Você deve ter instalado em seu computador o Python, versão 3.x (<https://www.python.org/downloads/>). Esta é o mesmo instalador utilizado na disciplina de Linguagem de Programação 1, e configuração de instalação será exatamente a mesmo. O que mudará será o ambiente utilizado para escrever os programas.

Um programa de computador ou um projeto pode ser desenvolvido utilizando apenas o bloco de notas. Na linguagem Python, é possível, por exemplo, escrever o programa no Notepad++ salvar o arquivo com a extensão .py e pronto, o programa está pronto para ser executado (Figura 1)

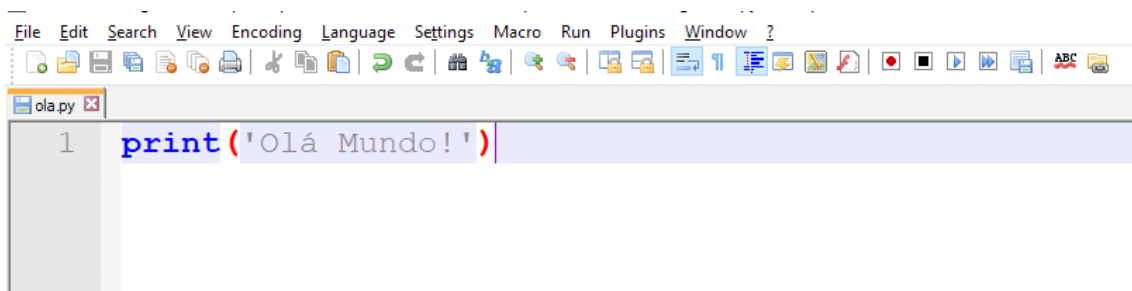


Figura 1 - Programa ola.py em Python, escrito utilizando o Notepad++

A linguagem Python possui o próprio Ambiente de Desenvolvimento Integrado (IDE – Integrated Development

Environment), chamado IDLE. O IDLE é instalado automaticamente quando o Python é instalado. A IDLE é um programa que contém o Python Shell (console para executar comandos do Python) e o próprio editor de texto, onde é possível escrever um programa e executá-lo diretamente no Python Shell, pressionando F5 (Figura 2).

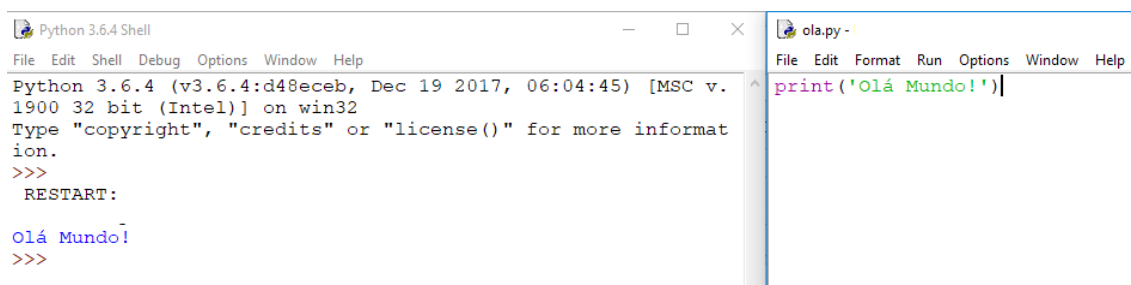


Figura 2 - Programa em Python escrito (lado direito) e executado no Python Shell (lado esquerdo) utilizando o IDLE

Um IDE, de forma geral, é um software que prove facilidade quando estamos escrevendo um programa. Por exemplo, a IDLE provê a coloração do código, para facilitar a identificação de comandos e estruturas, e a integração com o Python Shell para executar o programa escrito. Um IDE deve prover, pelo menos:

- Editor de código fonte
- Construção automática (*build*)
- Ferramenta de depuração (*debugger*)

Alguns IDEs mais modernos proveem outras funcionalidades, como complemento automático de código, integração com ferramentas de versionamento, construção de interfaces gráficas, e até mesmo ferramentas para construir diagramas.

Neste curso utilizaremos um IDE bem moderno chamado Visual Studio Code (<https://code.visualstudio.com/>), que será chamado simplesmente de VS Code nesta apostila, disponível para as plataformas Windows, Linux e macOS. O VS Code é um IDE de propósito geral, ou seja, pode ser utilizado para diversas linguagens. Ele provê a grande parte das facilidades mencionadas, além de ser leve e ter uma ótima integração com a linguagem Python. Sua instalação também é bem simples, bastando executar a instalação padrão, ou baixar o pacote .zip e executar sem a necessidade de instalar. Ao abrir o IDE, ele será parecido com o a Figura 3.

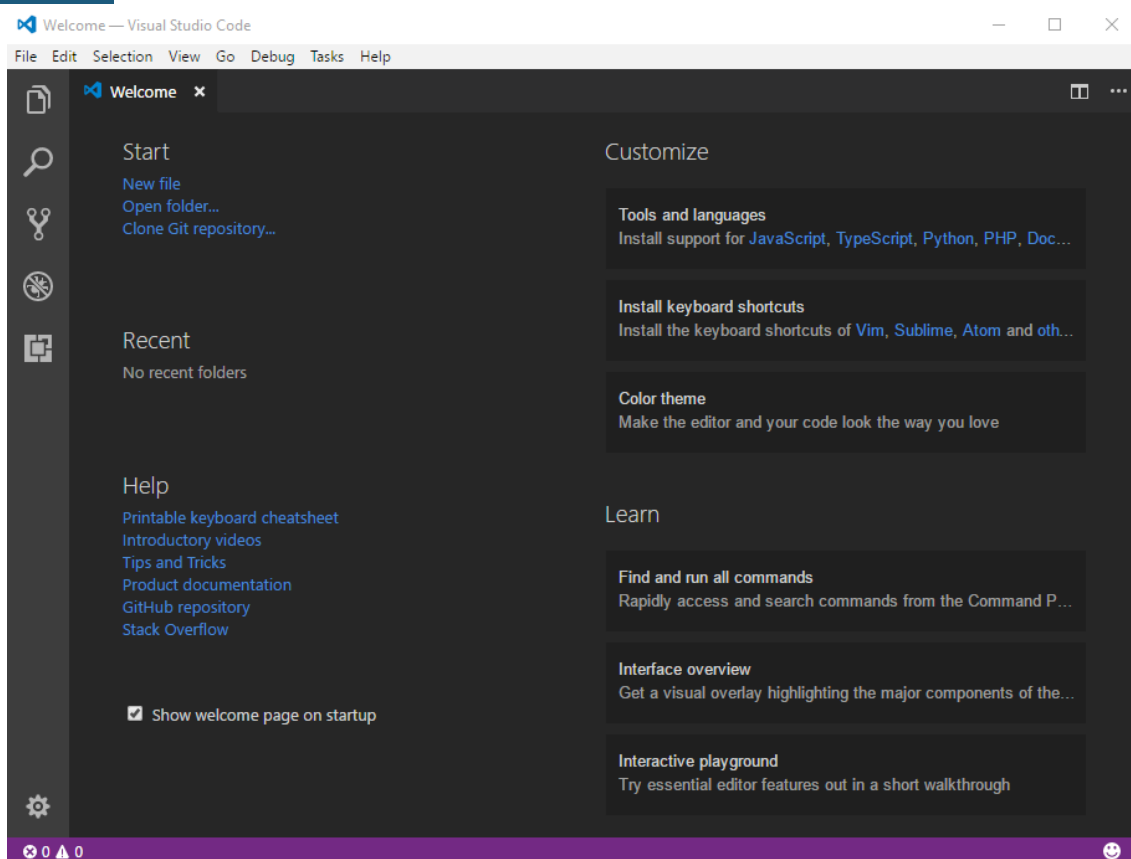


Figura 3 - Tela inicial do Visual Studio Code

Por ser um IDE de uso geral, a instalação padrão vem somente com as funcionalidades básicas. Outras funcionalidades ferramentas específicas para cada linguagem são instaladas por plug-ins, através da própria interface da IDE. Nesta disciplina será utilizado o plug-in do Python. Proceda com a instalação deste plug-in conforme mostra a Figura 4.

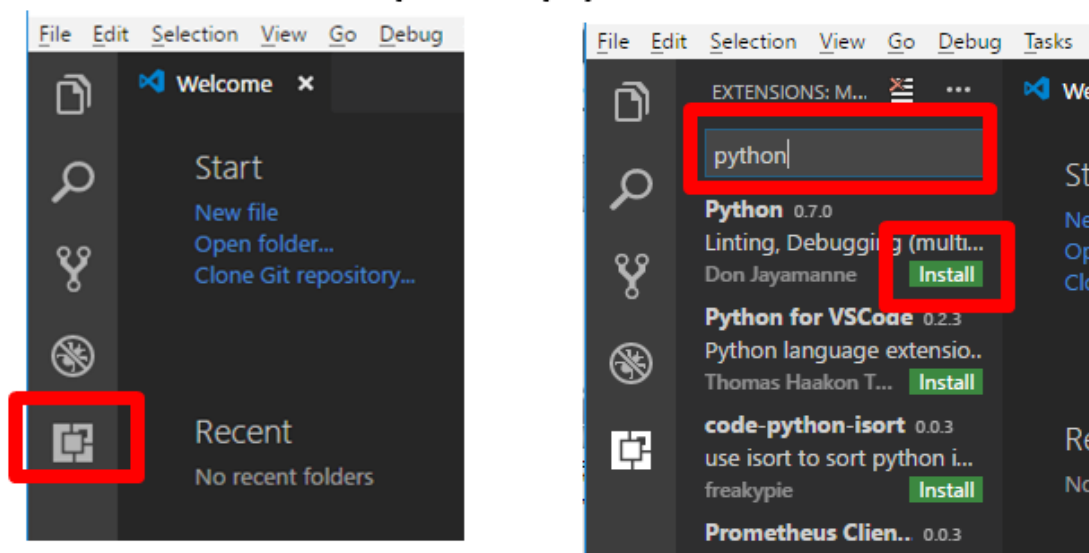


Figura 4 - Instalação do plug-in para Python no Visual Studio Code

Com este plug-in instalado o ambiente já está pronto para a criação de programas em Python. Um projeto no VS Code está associado a uma pasta do seu computador. Quando este IDE é aberto pela primeira vez, nenhuma pasta está selecionada, ou seja, não há nenhum projeto aberto. É preciso então informar ao IDE onde está seu projeto, clicando no botão Open Folder, disponível na primeira opção do menu ao lado esquerdo (Figura 5). Como exemplo foi criada a pasta OlaMundo, e ela foi selecionada para ser aberta no VS Code (o local da pasta pode ser qualquer caminho do computador) (Figura 6).

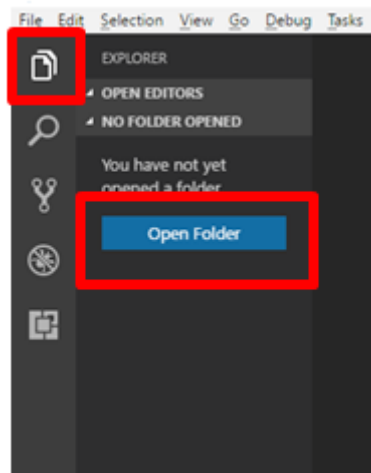


Figura 5 - Botão para selecionar uma nova pasta de projeto

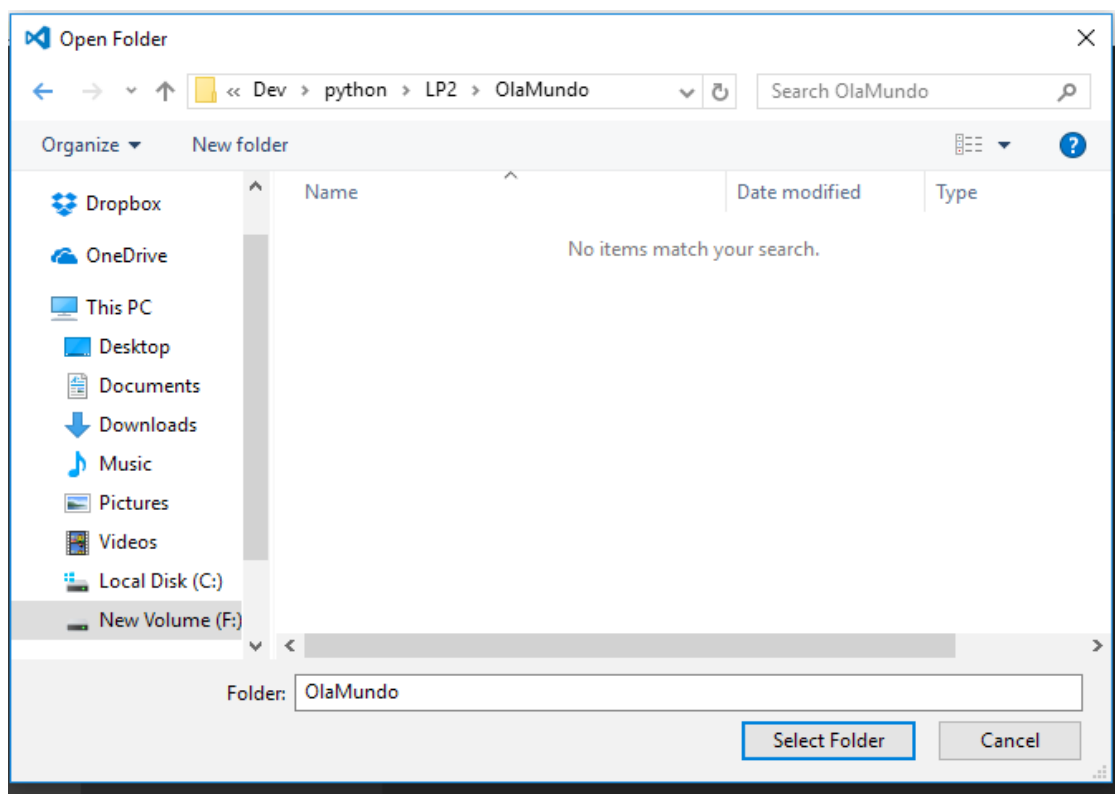


Figura 6 - Seleção da pasta do projeto no VS Code

Com a pasta selecionada (projeto aberto) é possível incluir arquivos no projeto. O mesmo programa `ola.py` feito no Notepad++ e IDLE, pode agora ser feito no VS Code. Isso é feito clicando no ícone de novo arquivo (Figura 7). A simplicidade do IDE faz com que o arquivo apareça logo abaixo, sendo necessário nomeá-lo (no exemplo, `ola.py`). Ao clicar no arquivo, ele aparecerá no editor de texto ao lado direito (Figura 8).

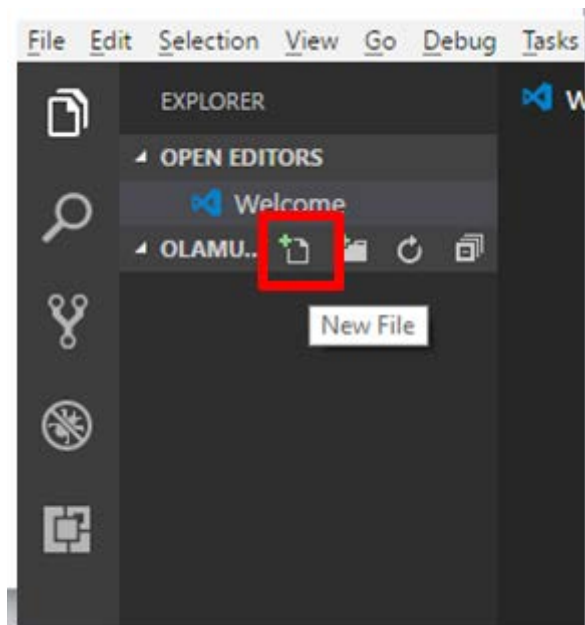


Figura 7 - Opção de inserir novo arquivo no Projeto do VS Code

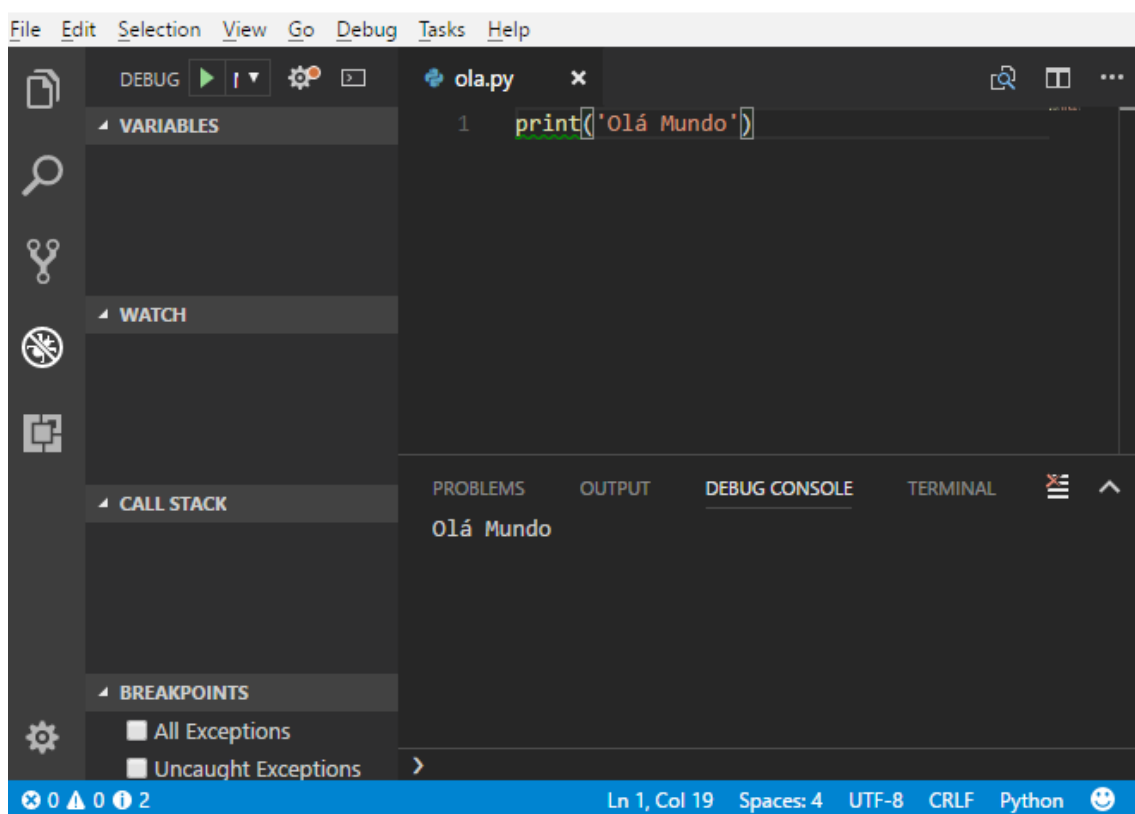


Figura 8 - Programa em Python aberto no editor de texto do VS Code

Para executar o programa, clique com o botão direito no código-fonte e selecione a opção "Run Python File in Terminal". Seu programa será executado no terminal integrado do VS Code, que aparece logo abaixo do código. Esta é a configuração básica do ambiente para a disciplina. Outras configurações necessárias serão realizadas ao longo dos capítulos.

3. Testes Unitários

Testar é importante em qualquer área, com qualquer coisa que fazemos. Em desenvolvimento de software não é diferente. É de suma importância realizar testes de software para garantir sua qualidade. Existem técnicas e bibliotecas que ajudam a fazer testes unitários automatizados, diminuindo o tempo gasto com testes manuais.

- ✓ Elucidar a importância de testes de software;
- ✓ Apresentar a biblioteca pytest;
- ✓ Criar e executar testes unitários utilizando pytest.

Iniciaremos este curso falando de testes unitários. A ideia é eu daqui para frente você inclua a cultura de testes de software durante o desenvolvimento dos seus programas. Python, e outras linguagens de programação proveem ferramentas que auxiliam na realização de testes do seu programa. Mas porque devemos testar?

Pessoas tem habilidade que são difíceis de serem implementadas por computadores, como por exemplo, interpretar a expressão de outras pessoas. São capacidades que fazemos muito bem, e que um computador não se sai tão bem. Mas se há uma coisa em que nós, seres humanos, não somos tão bons é em se ater a detalhes. O detalhe é algo muito importante em programação de computadores. Dependendo do problema, programar pode ser desafiador. E erros podem acontecer, afinal, seres humanos erram. Os erros usualmente são difusos, persistentes e inevitáveis. Ou seja, os erros em programas vão acontecer.

Por isso, é importante testar os softwares, de diversas maneiras e com várias situações possíveis. Testes de software fazem parte do processo de desenvolvimento. Testamos durante o desenvolvimento, após o software estar pronto e quando precisa de atualizações.

Os erros em programas podem ser basicamente de dois tipos: erros de sintaxe ou erro de semântica.

Os erros de sintaxe são aqueles que ocorrem na estrutura do código. Ou seja, escrevemos algum comando errado, algum bloco mal indentado, a falta de um parêntesis... e quando chega à linha deste comando, o programa lança um erro. As IDEs mais modernas contém ferramentas para diminuir estes erros antes de executar o programa, marcando onde foi identificado o erro de sintaxe.

Já os erros de lógica, ou de semântica, acontecem por uma lógica errada ou mal implementada. Às vezes estes tipos erros não causam a parada do software, mas podem gerar situações catastróficas. Imagine se quando você depositasse um dinheiro na sua conta, o programa do banco diminuísse o valor do seu saldo. Isso é um erro de lógica. O programa executou sem erros aparentes, mas a operação que ele realizou estava errada. Lembre-se que o programa só faz o que a gente manda. Se há algo errado, fomos nós que programamos.

Softwares robustos devem conter o mínimo de erros possíveis. Bugs no programa podem causar catástrofes e prejuízos financeiros. Uma coordenada errada lançada no plano de voo de um avião, um arredondamento fora do lugar numa operação financeira, uma instância de um objeto mal alocada, causando confusão nos dados de um paciente em um prontuário... enfim, inúmeros exemplos podem ser mostrados.

Dado que os erros podem acontecer, como fazemos para testar? A primeira tentativa seria fazer testes manuais. Entretanto, esta é uma abordagem cansativa e você vai testar apenas alguns casos e algumas vezes.

Por isso existem os testes automatizados, onde definimos uma bateria de testes que cobre o maior número de casos possíveis e pode ser executada rotineiramente várias vezes ao dia. Nesses testes automatizados definimos uma série de entradas que sabemos qual será a saída e então comparamos com a saída do nosso programa.

Em Python existem algumas ferramentas que ajudam a realizar testes unitários vamos utilizar aqui o pytest (docs.pytest.org), uma biblioteca do Python que facilita bastante a criação de testes, encapsulando muitas das configurações.

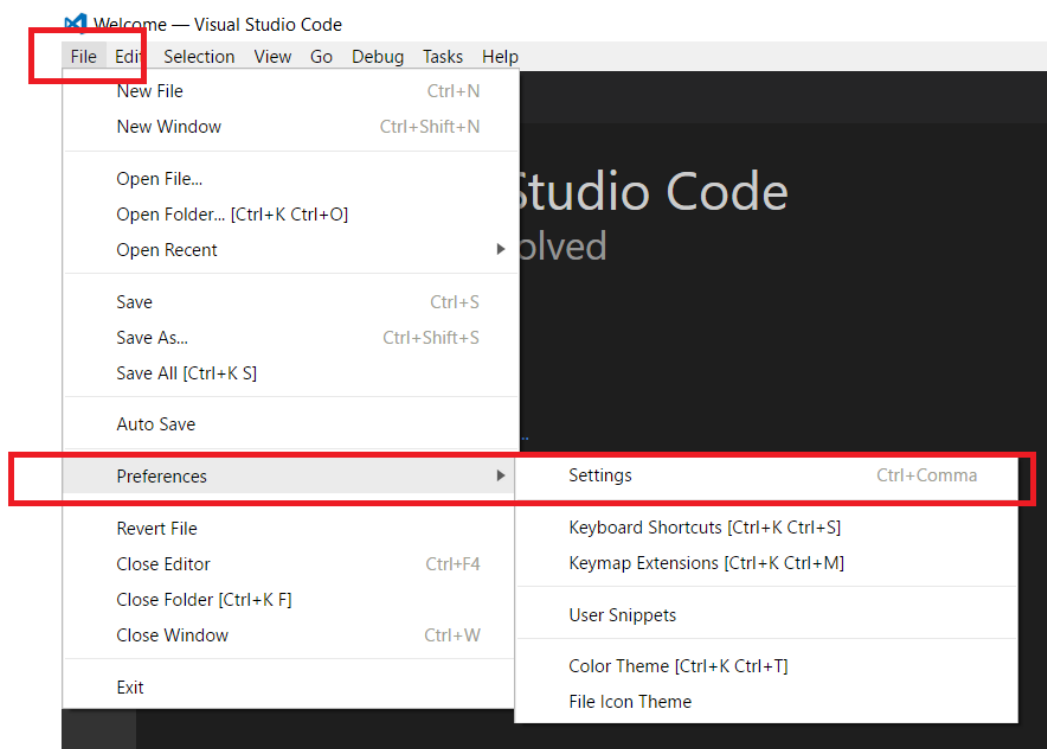
Primeiro é preciso instalar o pytest dentro do seu ambiente Python. Qualquer biblioteca do Python pode ser insta

lada utilizando o gerenciador de pacotes pip. Para instalar o pytest, abra uma janela do CMD do Windows e digite o seguinte comando:

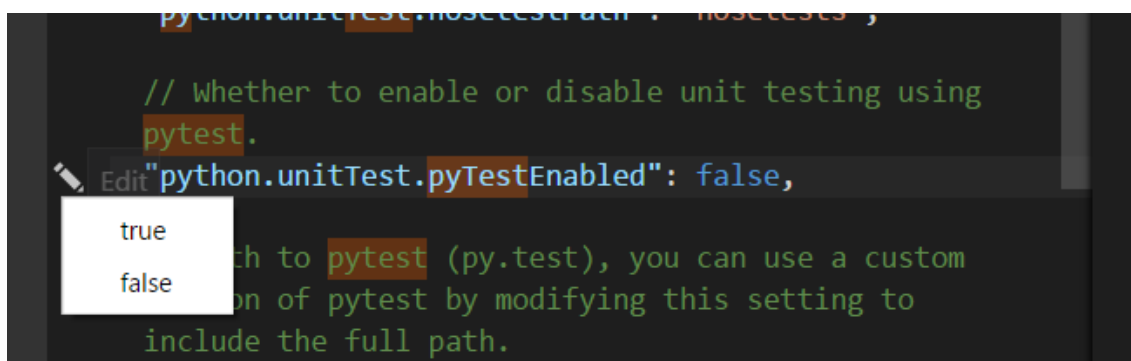
```
pip install pytest
```

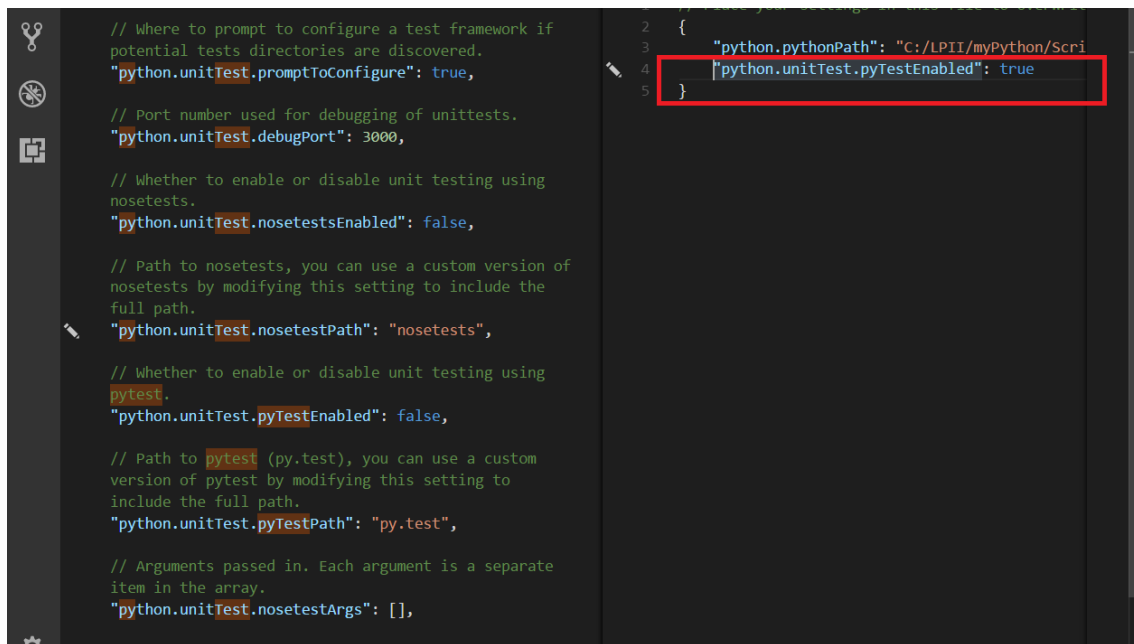
Se nenhum erro ocorrer, você já terá o pytest instalado no seu ambiente. Agora basta criar um programa de testes e utilizar a biblioteca.

Como estamos utilizando o VSCode, vamos configurá-lo para utilizar o pytest. Abri o VSCode e selecione o menu File>Preference>Settings



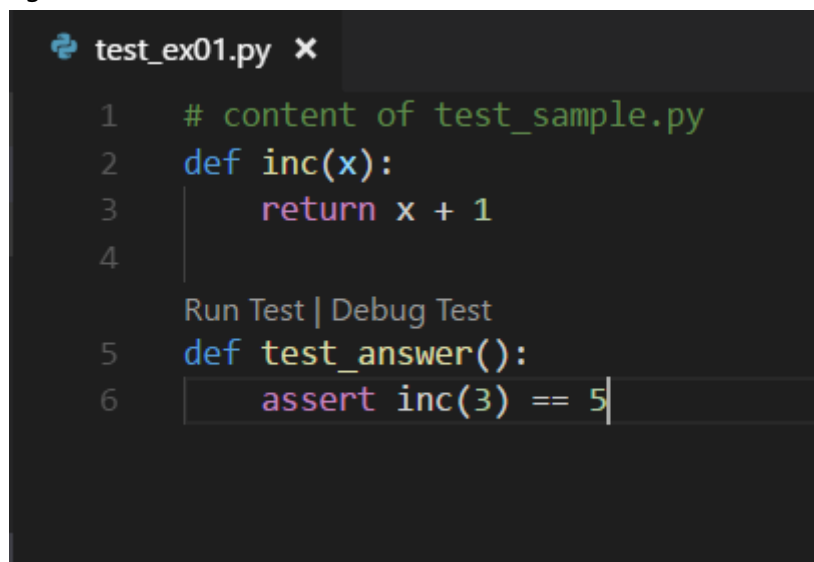
Em Search Settings digite pytest e selecione python.unitTest.pytestEnabled. Altere para True



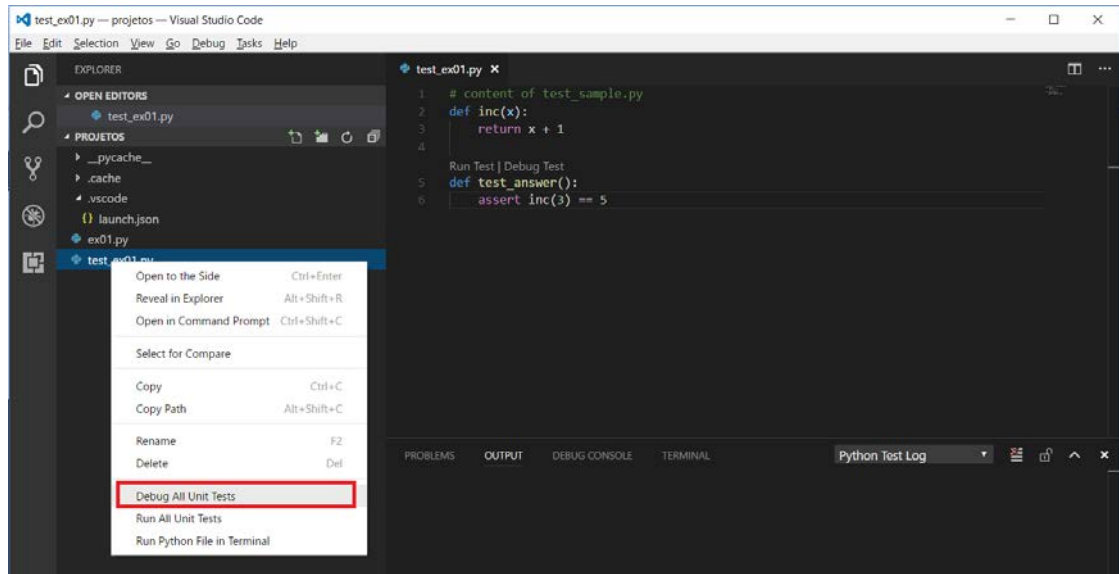


3.1. Primeiros Testes

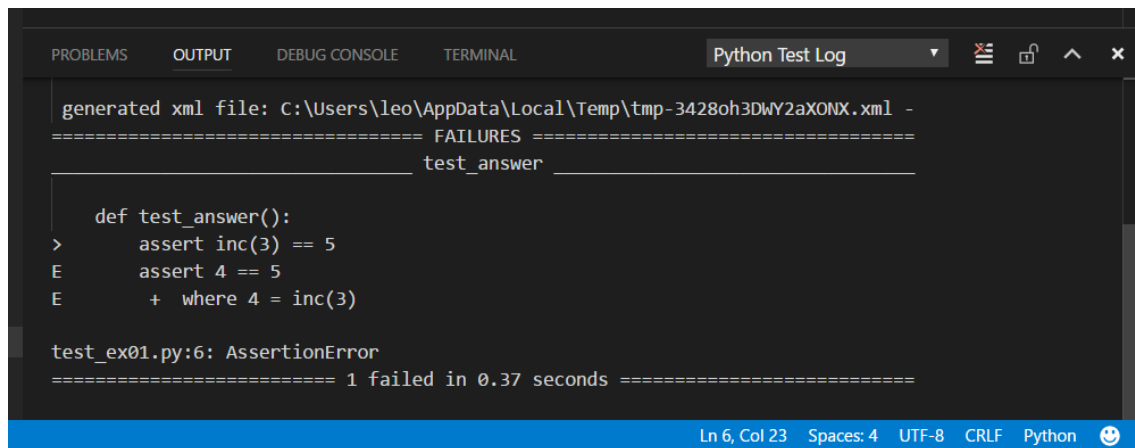
Para realizar testes unitários com pytest, existem algumas regras que devemos seguir. Primeiro, o arquivo que tem os testes deve começar com o nome `test_`. Por exemplo, `test_ex01.py` seria um nome válido para um arquivo de teste utilizando pytest. Sabendo disso, Crie um arquivo de teste chamado `test_ex01.py` em um novo projeto no VS Code e escreva o seguinte código de teste.



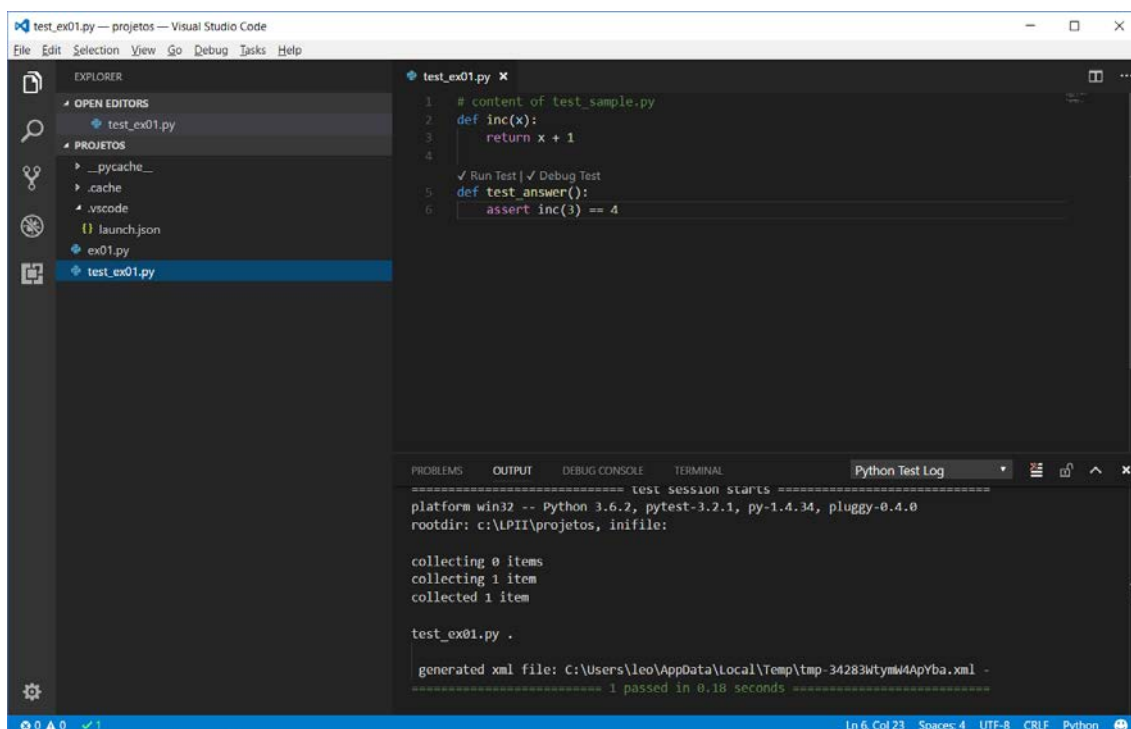
Salve o arquivo. Na janela Explorer, clique com o botão direito sobre o arquivo `test_ex01.py` e selecione a opção "Debug All Unit Tests".



Verifique pelo output que ocorreu falhas



Corrija os erros e execute até conseguir resolver todos os problemas.



O pytest:

- Considera que arquivos do tipo `test_*.py` e `*_test.py` são arquivos de teste
- Executa funções do tipo `test_*`
- Considera classes do tipo `Test*`

Agora que você já sabe da importância dos testes, e como utilizar o pytest, segue um resumo:

- Pense cuidadosamente nos casos em que seu programa pode falhar.
- Pense nos diferentes tipos de entrada que exercitam caminhos diferentes no seu programa.
- Pense nos casos diferentes no seu código.
- Escreva testes automatizados para todos os casos!

Sugestão de vídeo complementar: Assista e realize o coding dojo Happy numbers: <https://www.youtube.com/watch?v=9gokU36gZTY>

4. Dicionários

Linguagens de programação contêm estruturas que facilitam armazenar e manipular dados dentro de um programa, como por exemplo, vetores e listas. Python disponibiliza uma estrutura de dados não sequencial chamada dicionário. Neste capítulo veremos como utilizar dicionário e manipular os dados inseridos nele.

- ✓ Compreender o uso de dicionários do Python para armazenar coleções não sequenciais;
- ✓ Familiarizar-se com as funções e os métodos disponíveis para manipulação de dicionários em Python.

4.1. Objetivos

Os objetivos deste capítulo são:

- Compreender o uso de dicionários do Python para armazenar coleções não sequenciais.
- Familiarizar-se com as funções e os métodos disponíveis para manipulação de dicionários em Python.

4.2. Conceitos

Em Linguagem de Programação 1 foi introduzido o conceito de listas. Listas possibilitam que elementos sejam armazenados e removidos de coleções sequenciais. Quando desejamos acessar um elemento da lista, procuramos pelo seu índice – sua posição na coleção. Python provê outra estrutura para armazenar coleções de dados: Dicionários. Algumas aplicações requerem uma estrutura mais flexível para armazenar dados. Por exemplo, armazenar o nome de um aluno vinculado ao seu número de matrícula. Em programação, isso é conhecido como um par chave-valor. O número da matrícula do aluno é a chave, enquanto que o nome do aluno é o valor. O valor (nome do aluno) é acessado associando-se a uma chave (RA).

Uma coleção que possibilita encontrar um dado associado a uma chave é chamado de mapeamento, ou seja, mapear uma chave a um valor. Dicionários em Python são uma espécie de mapa. Em outras linguagens, estruturas similares são chamadas de Hash ou arrays associativos. Cada par chave-valor de um dicionário é chamado de item. Os dicionários são armazenados em variáveis como qualquer outro tipo de dado em Python. Um dicionário é criado e atribuído a uma variável colocando os itens separados por vírgulas dentro de chaves, da seguinte forma:

```
dic = {item1, item2, item3, ..., itemN}
```

Cada item começa com o nome da chave, seguido por dois pontos (:) e finalizado com o valor:

```
dic = {chave1: valor1, ..., chaveN: valorN}
```

Por exemplo, armazenar o número de matrícula e nomes dos alunos em dicionários, onde a matrícula é a chave e o nome o valor:

```
alunos = {11111:"Alberto", 22222: "Bruna"}
```

É importante salientar que o valor (depois dos :) pode ser qualquer tipo de dado em Python: int, float, string, listas, e até outros dicionários. Já a chave deve ser de um tipo imutável, como inteiro, string e tuplas. Se a chave for uma string, deve vir entre aspas (simples ou dupla), como qualquer string em Python:

```
aluno = {"nome":"Alberto", "idade": 26}
```

O acesso a um valor associado a uma chave é feito da mesma forma que em listas, utilizando a notação de índices, entre colchetes. O índice do dicionário é a chave utilizada na criação do item. De forma geral, o seguinte comando retorna o valor associado à chave informada:

```
<dicionario>[<chave>]
```

Veja no código abaixo alguns exemplos de dicionários, junto com o comando print() para mostrar seu conteúdo:

#dicionario vazio

```
dic_vazio = {}
```

#dicionario com chave int

```
alunos = {11111:"Alberto", 22222:"Bruna"}
```

#dicionario com chave string

```
aluno = {"nome":"Alberto","idade": 26}
```

#dicionario para armazenar uma agenda, associando #pelo nome

```
agenda = {"Igor":"555-555","Jorge":"555-666"}
```

```
print(dic_vazio)
```

```
print(alunos[11111])
```

```
print(aluno['idade'])
```

#acessar o telephone do Igor (chave)

```
fone_igor = agenda["Igor"]
```

```
print("Disque", fone_igor, "para ligar para Igor")
```

Assim como listas, dicionários são mutáveis. O valor associado a uma chave pode ser alterado por atribuição, assim como nas listas. O código a seguir mostra a atribuição de um novo valor a chave informada, de forma geral:

<dicionario>[<chave>] = <novoValor>

Veja no código abaixo alguns exemplos de alteração do valor de itens de dicionários, junto com o comando print() para mostrar seu conteúdo:

```
alunos = {11111:"Alberto", 22222:"Bruna"}
```

```
aluno = {"nome":"Alberto","idade": 26}
```

```
agenda = {"Igor":"555-555","Jorge":"555-666"}
```

#troca o valor da chave 11111

```
alunos[11111] = 'Amanda'
```

#troca o valor da c

have idade

```
aluno['idade'] = 20
```

#troca o valor da chave Jorge

```
agenda["Jorge"] = '555-000'
```



```
#retorna o novotelefone do Jorge
```

```
fone_jorge = agenda['Jorge']
print(alunos)
print(aluno)
print("Novo telefone de Jorge:", fone_jorge)
```

Em resumo:

- Dicionários são coleções mutáveis que implementam um mapeamento de chaves para valores correspondentes
- Chaves podem ser de qualquer tipo imutável
- Valores podem ser qualquer tipo, incluindo listas, objetos e até mesmo outros dicionários.

4.3. Manipulação de dicionários

Para adicionar um novo item ao dicionário basta fazer uma nova atribuição, utilizando uma nova chave. De forma geral, o comando abaixo adiciona um novo item nova_chave: novo_valor ao dicionário:

```
<dicionario>[<nova_chave>] = <novo_valor>
```

Veja no código abaixo alguns exemplos para adicionar itens em dicionários, junto com o comando print() para mostrar seu conteúdo:

```
alunos = {11111:"Alberto", 22222:"Bruna"}
aluno = {"nome":"Alberto","idade": 26}
agenda = {"Igor":"555-555","Jorge":"555-666"}
#adiciona o valor Camila a nova chave 33333
alunos[33333] = 'Camila'
#adiciona o valor M a nova chave sexo
aluno['sexo'] = 'M'
#adiciona o valor 555-333 a nova chave Luciana
agenda["Luciana"] = '555-333'
print(alunos)
print(aluno)
print(agenda)
```

Assim como listas, dicionários utiliza a função len() para retornar a quantidade de itens de um dicionário. De forma geral, utiliza-se len(dicionario). Veja o exemplo abaixo:

```
alunos = {11111:"Alberto", 22222:"Bruna"}
```

```
#tamanho do dicionário: 2
```

```
print(len(alunos))
```

Para remover um item de um dicionário a partir de uma chave utiliza-se o operador `del dicionario[chave]` (também como utilizado em listas). Por exemplo:

```
alunos = {11111:"Alberto", 22222:"Bruna"}
```

```
#remove o item com chave 11111
```

```
del alunos[11111]
```

```
print(alunos)
```

Também é importante verificar se existe uma chave no dicionário, antes de acessar o remover. Para isso utiliza-se o operador `in`. O resultado será um booleano:

- True: chave existe
- False: chave não existe

```
alunos = {11111:"Alberto", 22222:"Bruna"}
```

```
agenda = {"Igor":"555-555", "Jorge":"555-666"}
```

```
#verifica se chave 11111 existe em alunos
```

```
existe11111 = 11111 in alunos
```

```
#verifica se chave Luciana existe em agenda
```

```
existeLuciana = 'Luciana' in agenda
```

```
print(existe11111)
```

```
print(existeLuciana)
```

Todas as chaves de um dicionário podem ser retornadas em uma lista. Para tanto se utiliza o método `keys()`, aplicado sobre o dicionário que se deseja retornar as chaves. Exemplos:

```
alunos = {11111:"Alberto", 22222:"Bruna"}
```

```
agenda = {"Igor":"555-555", "Jorge":"555-666"}
```

```
#chaves de alunos
```

```
chavesAlunos = alunos.keys()
```

```
#chaves de agenda
```

```
chavesAgenda = agenda.keys()
```

```
print(chavesAlunos)
```

```
print(chavesAgenda)
```

De maneira similar, todos os valores de um dicionário podem ser retornados em uma lista. Para tanto se utiliza o

método `values()`, aplicado sobre o dicionário que se deseja retornar os valores. Exemplos:

```
alunos = {11111:"Alberto", 22222:"Bruna"}
agenda = {"Igor":"555-555", "Jorge":"555-666"}
#chaves de alunos
chavesAlunos = alunos.values()
#chaves de agenda
chavesAgenda = agenda.values()
print(chavesAlunos)
print(chavesAgenda)
```

Também é possível retornar as chaves e valores de um dicionário em uma lista de tuplas (chave, valor). Isso é feito utilizando o método `items()`, similar aos métodos `keys()` e `values()`. Veja no exemplo a seguir:

```
alunos = {11111:"Alberto", 22222:"Bruna"}
agenda = {"Igor":"555-555", "Jorge":"555-666"}
#itens de alunos
itensAlunos = alunos.items()
#itens de agenda
itensAgenda = agenda.items()
print(itensAlunos)
print(itensAgenda)
```

Outro método importante é o `get(chave [, padrão])` (`[, padrao]` não é uma lista, apenas informa que este parâmetro é opcional). Ele é utilizado para retornar o valor de uma chave de um dicionário, similar ao acesso direto com os colchetes (`dicionario[chave]`). A diferença é que, se a chave não existir, não será lançado um erro, mas retornado `None` ou então o valor especificado no segundo parâmetro do método. Veja os exemplos:

```
alunos = {11111:"Alberto", 22222:"Bruna"}
agenda = {"Igor":"555-555", "Jorge":"555-666"}
#reotrna o valor de 11111 ou None, caso não
#exista a chave
valorAluno = alunos.get(11111)
#reotrna o valor de Luciana ou 'não existe', caso não
#exista a chave
valorAgenda = agenda.get('Luciana','não existe')
print(valorAluno)
print(valorAgenda)
```

Para limpar todos os elementos de um dicionário, utiliza-se o método `clear()`, da seguinte forma:

```
alunos = {11111:"Alberto", 22222:"Bruna"}
alunos.clear()
print(alunos)
```

Dicionários são coleções, portanto estruturas iteráveis no Python. Um dicionário pode ser percorrido utilizando um loop `for` sobre o dicionário. O comando `for chave in dicionario` retorna na variável `chave` cada uma das chaves do dicionário. Também é possível iterar no par `chave:valor`. Para tanto deve-se utilizar o método `items()`, visto anteriormente. O comando `for` ficará então da seguinte forma: `for chave, valor in dicionario.items()`. Como o método `items()` retorna uma lista de tuplas, cada elemento da lista (uma tupla) será automaticamente convertido em duas variáveis no loop `for`: `chave` e `valor`. Veja os exemplos a seguir:

```
agenda = {"Igor":"555-555","Jorge":"555-666"}
#percorre as chaves do dicionário
# e mostrar o valor de cada item da agenda
for chave in agenda:
    print(agenda[chave])
#mostrar a chave e o valor de cada item da agenda
for chave, valor in agenda.items():
    print(chave, ":", valor )
```

4.4. Inserindo complexidade

Anteriormente, foi comentado que um dicionário em Python pode armazenar qualquer tipo de dado válido em seus valores:

- `int`;
- `float`;
- `string`;
- `listas`;
- `objetos`;
- outros dicionários;

Suponha que no dicionário de agenda dos exemplos anteriores, queremos guardar 2 telefones para um mesmo nome. Podemos colocar os dois telefones em uma lista e associar essa lista (valor) ao nome da pessoa (chave). Por exemplo:

```
agenda = {
    "Igor": ["555-555", "666-666"],
    "Jorge": "555-666",
    "Luciana": ["555-444"]
}
print(agenda)
```

Repare na flexibilidade de dicionários: as chaves podem armazenar tipos de dados diferentes dentro de um mesmo dicionário. No exemplo, um mesmo dicionário tem chaves que armazenam listas e strings.

Imagine outra situação em que deseja incluir outros dados do contato, por exemplo, telefone, e-mail e endereço, todos eles relacionado ao mesmo nome. Podem-se utilizar dicionários aninhados, ou seja, um dicionário dentro do outro. A chave no primeiro nível continua sendo o nome da pessoa da agenda. O valor não será um número de telefone nem uma lista de telefones. Será outro dicionário com telefone, e-mail e endereço (e qualquer outra informação do contato que quiser armazenar). As chaves desse novo dicionário serão: telefone, e-mail e endereço. Veja no exemplo:

```
agenda = {
    "Igor": {
        'telefone': ["555-555", "666-666"],
        'email': 'igor@email.com'
    },
    "Jorge": {
        'telefone': "555-666",
        'email': 'jorge@email.com',
        'endereço': 'Rua da Casa, 123'
    },
}
print(agenda)
```

Repare que os dicionários aninhados não precisam ter a mesma estrutura. "Igor" tem "telefone" e "e-mail", enquanto que "Jorge" tem "telefone", "e-mail" e "endereço".

5. Modularização

Um programa complexo tem milhares de linhas de código. Podemos organizá-lo por exemplo utilizando funções, o que facilita a leitura, manutenção e reuso de código. No próximo capítulo veremos como podemos organizar o programas em classes. Mas antes veremos como utilizar módulos no Python, e como criar os próprios módulos para deixar o seu programa mais organizado

✓ Importar e utilizar módulos do Python;

✓ Criar módulos e organizar o programa em módulos.

Módulos em Python são arquivos com extensão .py utilizados para implementar um conjunto de atributos (variáveis), métodos (funções) e classes (tipo) que podem ser reaproveitadas. Ou seja, são programas em Python que implementam tarefas específicas que podem ser reaproveitados. Os módulos podem ser importados para qualquer programa que deseje utilizar os atributos, métodos e classes do módulo.

5.1. Importar módulos

Para incluir, ou importar, um módulo para seu programa basta utilizar a instrução import seguido do nome do módulo. Python provê diversos módulos na instalação, como de matemática, interface gráfica data/hora, etc. Neste link, você consegue acessar uma lista de módulos do Python: <https://docs.python.org/3/py-modindex.html> Usualmente a instrução import vem no começo do programa. Sintaxe básica:

import modulo

Por exemplo, para importar a biblioteca matemática e utilizar a operação de raiz quadrada, Python provê um módulo matemático chamado `math`, que contém funções matemáticas, como seno, cosseno, logaritmo e raiz quadrada. O método deste módulo que calcula a raiz quadrada chama `sqrt(x)`. Este método calcula a raiz quadrada no valor de `x` enviado como parâmetro.

De forma genérica, para utilizar um método de um módulo é preciso:

- Importar o módulo para o projeto, utilizando `import`
- Utilizar o método desejado do módulo da seguinte forma:

```
import modulo  
modulo.metodo()
```

Repare que o nome do módulo foi repetido no uso do método, separados por um ponto (igual ao uso dos métodos de listas). Esta forma é chamada de importação absoluta. Para utilizar o método `sqrt(x)` do módulo `math` faça o seguinte:

```
import math  
  
n = input('Digite um número: ')  
raiz = math.sqrt(n)  
  
print(raiz)
```

Para utilizar um atributo de um módulo, a sintaxe é a mesma. A única diferença é que atributos são variáveis. Portanto, não se coloca parênteses nem há parâmetros. Veja o exemplo a seguir para trabalhar com datas:

- Módulo `time`
- Método `asctime()`: Retorna data/hora atual
- Atributo `timezone`: Armazena os segundos de diferença entre UTC (hora em Greenwich) e o horário local

```
import time  
  
#data/hora atual  
print(time.asctime())  
# diferença do UTC em segundos  
print(time.timezone)
```

Sempre que utilizar a instrução `import modulo`, todos os métodos e atributos do módulo estarão disponíveis utilizando `modulo.metodo()` ou `modulo.atributo`. Se for necessário utilizar apenas um método ou atributo do módulo, utiliza-se a seguinte instrução de importação:

```
from modulo import metodo
```

Dessa forma, o método ou atributo pode ser utilizado diretamente pelo nome, sem colocar o nome do módulo an-

tes. Esta forma é chamada de importação relativa. No exemplo da raiz quadrada, a importação relativa do método `sqrt()` do módulo `math` ficará da seguinte forma:

```
from math import sqrt

n = input('Digite um número: ')
raiz = sqrt(n)

print(raiz)
```

Outro exemplo: importar o atributo `name` do módulo `os`:

```
from os import name

#nome do sistema operacional
print(name)
```

Mais um exemplo: importar o atributo `pi` no módulo `math`:

```
from math import pi

#valor de Pi
print(pi)
```

Caso seja necessário importar mais de um método ou atributo de um ou mais de um módulo, pode-se utilizar uma importação relativa por linha.

```
from time import asctime
from time import timezone
from time import sleep
from math import log2

print(timezone)    # diferença do UTC em segundos

print(log2(10))    # logaritmo de 10 na base 2

print(asctime())    # Data/hora atual
sleep(10)           # Pausa execução por 10 segundos
print(asctime())    # Data/hora atual
```

Os métodos e atributos de um mesmo módulo podem estar em uma mesma linha, separados por vírgulas.

```
from time import asctime, timezone, sleep
from math import log2, pi

print(timezone)    # diferença do UTC em segundos

print(log2(pi))     # logaritmo de Pi na base 2

print(asctime())    # Data/hora atual
sleep(10)           # Pausa execução por 10 segundos
print(asctime())    # Data/hora atual
```

Outra forma de fazer a importação relativa é importar todos os métodos e atributos do módulo utilizando um asterisco (*) da seguinte forma:

```
from time import *
from math import *
```

Dessa forma, é possível utilizar qualquer atributo e método do módulo importado sem a necessidade de especificar o módulo (modulo.metodo() ou modulo.atributo). Entretanto, utilizar o * não é uma prática recomendada, apesar de ser muito utilizada. Dois módulos podem ter métodos ou atributos com mesmo nome. Ao importar tudo com *, você pode sobrepor a primeira importação e acabar utilizando o recurso errado.

5.2. Criar Módulos

Assim como Python tem suas próprias bibliotecas e módulos de códigos reaproveitáveis, você pode criar os seus módulos. Criar um módulo é bem simples

Módulos são arquivos .py, ou seja, programas em Python. Portanto, para criar um módulo, basta criar um programa .py com seu código e importe para outro programa em Python.

Crie um módulo (novo programa) que chama ola.py com uma função diz_ola, apenas para mostrar uma mensagem. Faça exatamente como você cria um programa em Python: crie um novo arquivo com o nome ola.py, codifique normalmente, e salve o arquivo.

```
# modulo ola.py
def diz_ola():
    print("Olá!")
```

Feito isso, crie um novo programa em Python, teste_ola.py, que importa e utiliza seu modulo. O nome do módulo é o nome do arquivo .py criado anteriormente (ola). Inicialmente, o módulo e o programa que o utiliza devem estar na mesma pasta.


```
# programa teste_ola.py
import ola

# chama metodo diz_ola do modulo ola
ola.diz_ola()
```

Os módulos são importantes para trabalhar com os testes de software. Os testes ficam em um arquivo separado dos arquivos do seu programa, isolando totalmente do programa que vai para produção. No arquivo de teste fazemos a importação do módulo que queremos testar. Por exemplo, escreva a função `first_last6`, para verificar se o número 6 é o primeiro ou o último da lista, e uma função de testes (assim como fez no capítulo de testes)

```
#test_ex01_first_last6.py
def first_last6(nums):
    return

def test_ex01():
    print ('First_last6')
    assert first_last6([1, 2, 6]) == True
    assert first_last6([6, 1, 2, 3]) == True
    assert first_last6([3, 2, 1]) == False
    assert first_last6([3, 6, 1]) == False
    assert first_last6([3, 6]) == True
    assert first_last6([6]) == True
    assert first_last6([3]) == False
```

Feito isso, remova a função `first_last6` do programa de teste e coloque-a em outro arquivo, chamado `ex01.py`

```
#ex01.py
def first_last6(nums):
    return
```

O arquivo de testes (`test_ex01_first_last6.py`) ficará apenas com a função com os casos de testes:

```
#test_ex01_first_last6.py
```

```
def test_ex01():  
    print ('First_last6')  
    assert first_last6([1, 2, 6]) == True  
    assert first_last6([6, 1, 2, 3]) == True  
    assert first_last6([3, 2, 1]) == False  
    assert first_last6([3, 6, 1]) == False  
    assert first_last6([3, 6]) == True  
    assert first_last6([6]) == True  
    assert first_last6([3]) == False
```

Para finalizar, faça a importação da função `first_last6` que está no módulo `ex01`:

```
#test_ex01_first_last6.py
```

```
from ex01 import first_last6
```

```
def test_ex01():  
    print ('First_last6')  
    assert first_last6([1, 2, 6]) == True  
    assert first_last6([6, 1, 2, 3]) == True  
    assert first_last6([3, 2, 1]) == False  
    assert first_last6([3, 6, 1]) == False  
    assert first_last6([3, 6]) == True  
    assert first_last6([6]) == True  
    assert first_last6([3]) == False
```

Seu programa com dois módulos já funciona. Para que os testes sejam executados corretamente, devemos implementar a função `first_last6` corretamente:

```
#ex01.py  
def first_last6(nums):  
    return nums[0] == 6 or nums[-1] == 6
```

6. Objetos em Python

Este capítulo formaliza o conceito de objetos em Python. É importante entender como os objetos funcionam e como criamos novos objetos para criar estruturas mais complexas e organizar os projetos

- ✓ Entender a evolução dos paradigmas e linguagens de programações;
- ✓ Explicar os conceitos de objetos, referência e valor;
- ✓ Compreender as características e comportamentos de conjunto de objetos e suas analogias;
- ✓ Aprender a utilização de objetos e seus membros em Python;
- ✓ Entender o comportamento dos objetos na memória.

6.1. Introdução: Linguagens de Programação

As linguagens de programação são anteriores ao advento do primeiro computador moderno. A chamada Linguagem de Máquina foi a primeira linguagem de programação que surgiu. Essa linguagem introduzia uma sequência de zeros e uns que permitia escrever programas de computador. Por exemplo:

```
00111001 00011101 001101010
```

Com essas sequências de zero e uns, é comum imaginar a quantidade de erros que eram cometidos e a dificuldade de se programar em uma linguagem deste tipo. Logo, criou-se a Linguagem de Montagem, o ASSEMBLY, que contém comandos um pouco mais inteligentes para compreensão humana. Por exemplo:

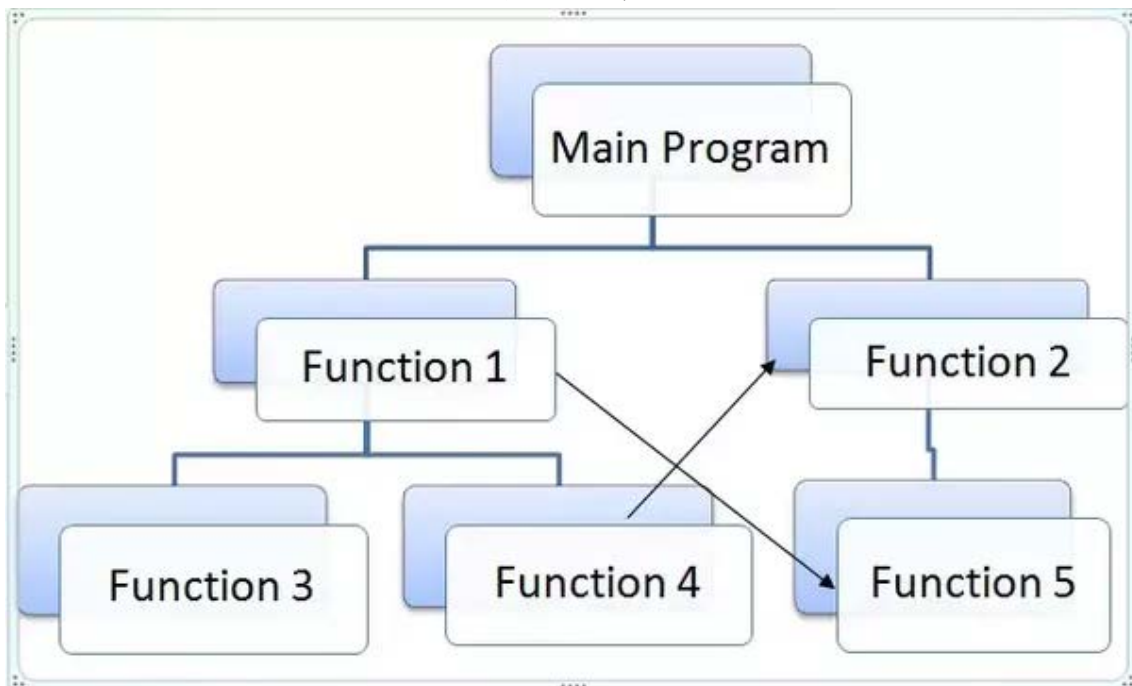
```
ADD AX, BX (Comando para somar o registrador AX no registrador BX)
INC (Comando para incrementar uma variável)
```

Com o ASSEMBLY houve melhora na compreensão e criação de programas, mas ainda era uma linguagem próxima do funcionamento da máquina e longe da linguagem natural humana. Então, começaram a criar linguagens de alto nível. Em particular, os cientistas preocupados em construir uma tradução de fórmulas matemáticas para a linguagem de montagem, criaram a linguagem FORTRAN (FORMula Translator). O FORTRAN foi a primeira linguagem moderna de alto nível criada na década de 50. Nesta mesma década, outras linguagens com um modelo conceitual um pouco mais sofisticadas foram desenvolvidas, como LISP (LIST Processor), COBOL (Common Business Oriented Language) e ALGOL (ALGOrithmic Language). Essas linguagens marcaram o início do desenvolvimento das linguagens de alto nível.

A partir das linguagens de alto nível, surgiram os Paradigmas de Linguagens de Programação, que fornecem e determinam a visão que o programador possui sobre a estruturação e execução do programa. Pode-se dividir a linguagem em 3 grandes grupos: paradigma funcional, paradigma estruturado (onde pode-se subclassificar em paradigma orientado a objetos) e paradigma declarativo.



No paradigma funcional ou procedural, as funções são o principal bloco de implementação dos programas, assim como estamos acostumados a trabalhar com as funções em Python.



Fonte: <https://www.quora.com/Whats-the-difference-between-Object-Oriented-Programming-and-Procedural-Programming>

No paradigma orientado a objetos, os programadores abstraem um programa como uma coleção de objetos que interagem entre si. Agora, os objetos são o bloco fundamental. As funções continuam a existir como componentes dentro dos objetos, e são chamadas de métodos.

A ideia principal da POO é abstrair coisas do mundo real. O mundo real está cercado de Objetos: aluno, professor, sala de aula, livro, carro, animal, televisão, etc. Um objeto é algo que tem um grupo de características e um grupo de comportamentos, onde comportamentos estão relacionados às características. Em programação orientada a objetos, variáveis do objeto definem as características e métodos definem o comportamento do objeto.

Por exemplo, considere as duas imagens a seguir, de uma onça e de um carro. Essencialmente, eles podem ser vistos como objetos. Cada objeto contém uma série de características e comportamento próprios.



Características (Atributos)	Comportamento (Métodos)	Características (Atributos)	Comportamento (Métodos)
Raça	Comer	Marca	Acelerar
Peso	Dormir	Cor	Frear
Cor dos olhos	Pular	Velocidade	Pintar

6.2. Objetos em Python

Em Python, todos os valores em Python são representados como objetos, desde os valores numéricos até listas e objetos complexos. Lembre-se do uso de listas: o que deve ser feito para:

- Ordenar uma lista?
- Adicionar em uma lista?
- Remover de uma lista?
- Inverter a lista?

Essencialmente, os tópicos acima são comportamentos de um objeto do tipo lista. Portanto para executar cada um deles, é utilizado um método deste objeto. Ou seja, são rotinas que fazem parte do objeto que contém a lista.

```
lista_nomes = ['Ana', 'Bernando', 'Caio']
```

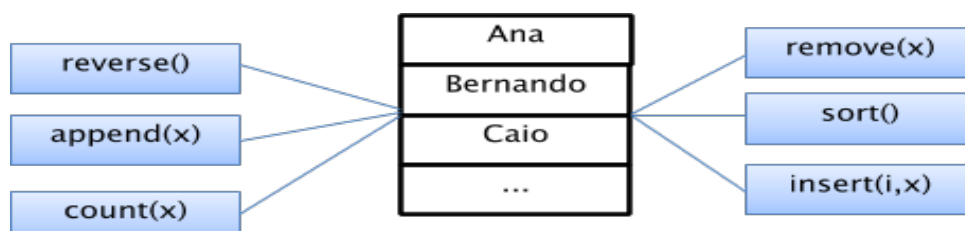
Por exemplo, suponha que exista uma variável chamada `lista_nomes` que recebe um objeto do tipo lista. Sendo uma lista, é possível aplicar qualquer método presente em listas sobre o objeto criado.

Por exemplo, para ordenar a lista basta chamar o método de ordenação (`sort`) presente no objeto (`lista_nomes`), separando por `."` (ponto). O `."` (ponto) é um operador utilizado sempre que for necessário acessar um membro de um objeto (variáveis ou métodos).

```
lista_nomes = ['Ana', 'Bernando', 'Caio']
```

```
lista_nomes.sort()
```

Além de `sort()`, um objeto do tipo lista contém diversos outros métodos que definem outros comportamentos da lista.



Todos os objetos do tipo lista criados em um programa Python contém os mesmos métodos, ou seja, qualquer lista criada pode utilizar os mesmos métodos. Se existis uma outra lista, mas agora de números (lista_numeros, por exemplo), ela também vai ter o método sort. Tanto lista_nomes quanto lista_numeros são objetos do tipo list. Entretanto, os dados nos quais a operação de ordenação será aplicado serão diferentes. Cada objeto tem seus dados, e os métodos agem nestes dados. Portanto, dois objetos do mesmo tipo têm os mesmos métodos. Eles diferem nos valores que guardam (variáveis ou atributos).

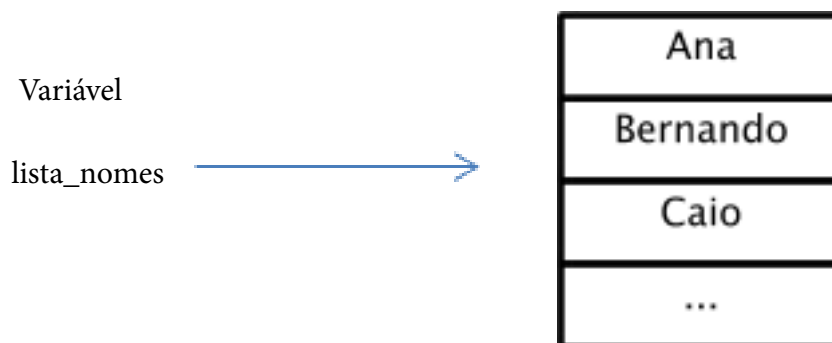
```
lista_nomes = ['Ana','Bernando','Caio']  
lista_nomes.sort()
```

```
lista_numeros = [1, 2, 3]  
lista_numeros.sort()
```

Para lembrar: um objeto contém um grupo de atributos ou características, armazenados em variáveis de instâncias e um grupo de funções, chamados de métodos, que provê o comportamento.

Antes de começar a criar as próprias classes e objetos, é importante entender como Python representa os objetos e o efeito que isso tem na atribuição.

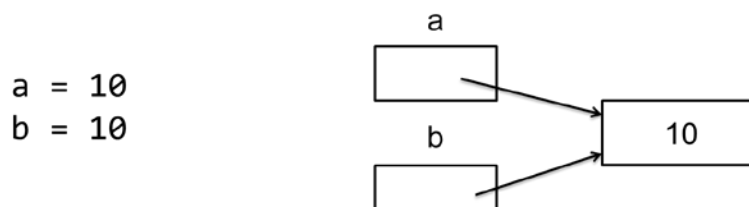
Em Python os objetos são representados como uma referência para um objeto na memória. Ou seja, a variável não contém o objeto propriamente dito, mas uma referência para o local da memória onde o objeto está. Voltando ao exemplo da lista de nome, ao criar uma lista e atribuí-la a uma variável, a variável apenas aponta para o objeto na memória, armazenando a referência. Esta variável é chamada de variável de referência.



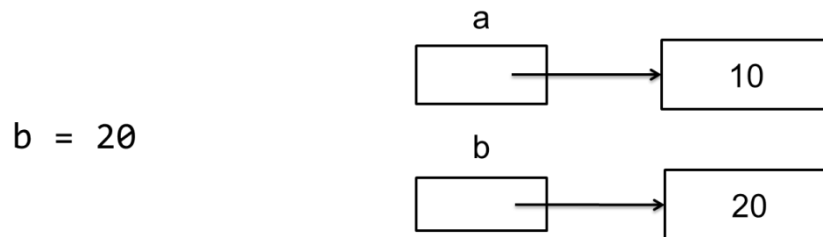
Qualquer objeto criado em Python, independente do tipo, deve ser acessado pela variável de referência. Entretanto o comportamento é diferente dependendo do tipo de dado que referenciado.

Em Python existem dois tipos de objetos: imutáveis, como int, float, bool, tupla e string; e mutáveis, como listas, dicionários, e objetos criados para o projeto. É importante entender como a referência se comporta para cada um dos tipos.

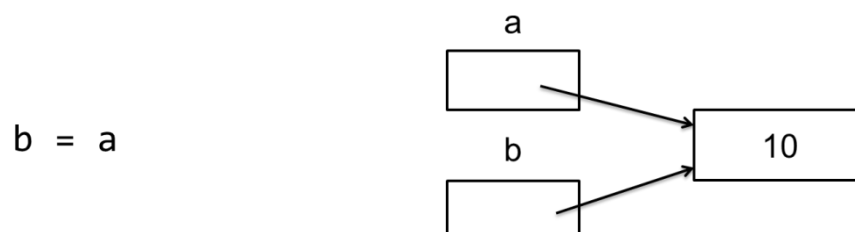
Quando um mesmo valor de um tipo imutável é atribuído a duas variáveis diferentes, as duas variáveis referenciam o mesmo local na memória. Ou seja, as duas variáveis referenciam a mesma instância. Isso é feito para economizar memória. Por exemplo:



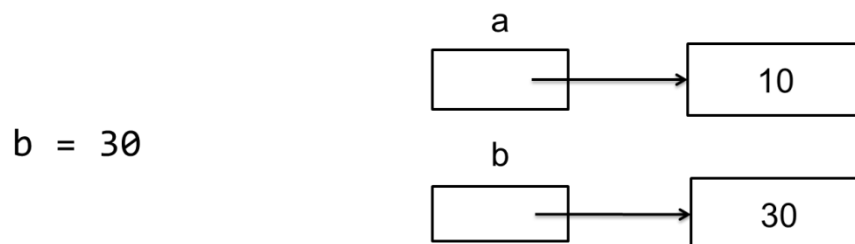
Se a variável *b* receber um novo valor (nova instância), ela passa a referenciar um novo espaço na memória, que armazena esta nova instância. A variável *a* permanece intacta.



Se a variável *b* receber a referência de *a* (ou seja, `b = a`), a variável *b* volta a referenciar a posição de memória onde está o valor 10.



Quando a variável *b* receber outro valor (nova instância), *b* novamente passa a referenciar um novo espaço na memória, que a nova instância.



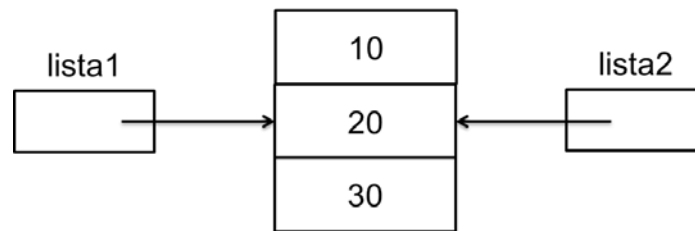
Em resumo, para tipos imutáveis, se duas variáveis referenciam a mesma posição de memória (mesma instância) e uma receber uma nova referência (nova atribuição), o valor referenciado pela outra variável permanece o mesmo, alterando apenas a instância referenciada pela primeira variável.

O comportamento para tipos mutáveis é diferente. Inicialmente, ao atribuir uma lista (tipo `list`) de números à variável `lista1`, Python referencia a memória da mesma forma que tipos imutáveis.



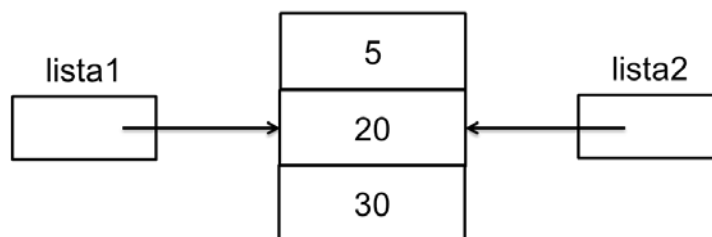
Ao atribuir a referência de `lista1` para uma nova variável `lista2` (`lista2 = lista1`), ambas as variáveis vão referenciar a mesma instância, ou seja, o mesmo espaço na memória.

```
lista2 = lista1
```



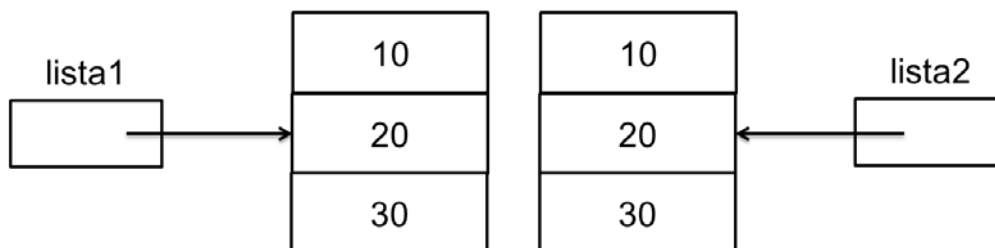
Por se tratar de um tipo mutável e ambas as variáveis referenciarem a mesma instância qualquer alteração em `lista1` refletirá na `lista2`.

```
lista1[0] = 5  
print(lista2[0])  
# a alteração do 5 na primeira posição da lista1  
# refletirá na lista2
```



Este é o comportamento para qualquer tipo mutável, inclusive para novos tipos de objetos criados. Para que `lista2` reference uma nova instância de `lista1` com o mesmo conteúdo, é necessário criar a nova instância. Com listas utiliza-se o construtor `list()`. Dessa forma, cada variável referencia uma instância diferente na memória, e a alteração de uma lista não interfere na outra.

```
lista2 = list(lista1)
```



7. Programação Orientada a Objetos

Este capítulo formaliza o conceito Programação Orientada a Objetos em Python, um paradigma de programação dos mais importantes. Você será introduzido ao mundo das classes e criar este tipo de estrutura em Python, utilizando construtores, métodos, atributos e atributos privados.

- ✓ Explicar o conceito fundamental de orientação a objetos;
- ✓ Explicar o conceito de classes;
- ✓ Entender como a criação de classes pode prover estruturação para programas complexos;
- ✓ Definir e utilizar classes em Python;
- ✓ Definir e utilizar construtores;
- ✓ Utilização de métodos especiais;
- ✓ Criar e utilizar atributos privados;
- ✓ Escrever programas envolvendo definição de classes.

7.1. Classes em Python

No capítulo anterior foi explicado o comportamento dos objetos mutáveis e imutáveis na linguagem Python. Objetos mutáveis incluem aqueles que serão criados daqui para frente. A criação de um novo tipo de objeto é feito a partir da definição de classes, da qual será possível instanciar novos objetos deste tipo.

Classes são como fôrmas que definem os tipos de objetos. Elas definem quais os dados e o código dos objetos (essencialmente, as características e comportamentos). Objeto é a instanciação de uma classe, ou seja, a criação de uma referência na memória do tipo definido pela classe. Cada objeto do tipo da classe terá todas as características e comportamento especificado na classe. Fazendo uma analogia, uma classe é uma forma de um biscoito, enquanto que um objeto são os biscoitos prontos, cada um com suas cores e enfeites próprios.

Os objetos são também chamados de instâncias das classes. Cada objeto tem suas próprias características. Com a fôrma pronta (classe), é possível criar quantos objetos forem necessários utilizando a mesma fôrma. Instâncias da mesma classe têm características comuns.

Por exemplo, relembre a um objeto do tipo onça citado anteriormente. As características comuns de onça, como raça, peso e cor dos olhos são definidos na estrutura da classe. Afinal, toda onça terá uma raça, peso e cor dos olhos, além de outras características. Mas cada onça, ou seja, cada instância dessa classe, a própria raça (onça parda, onça pintada, etc.), próprio peso (onças maiores, onças menores) e a própria cor dos olhos. Ou seja, quaisquer onças criadas terão as mesmas características, mas cada uma com um valor diferente.

Como exemplo prático, pense em definir uma classe do tipo animal, com dois objetos (instâncias): meu_animal e animal_da_fit. Em Python a criação da classe Animal simples (ou seja, a fôrma para qualquer animal) e das duas instâncias será da seguinte forma:

#Criação da classe Animal sem bloco de código (pass)

```
class Animal:  
    pass
```

#Criação de 2 instâncias de Animal

```
meu_animal = Animal()  
animal_da_fit = Animal()
```

#mostra a localização da memória para as instâncias

```
print(meu_animal)  
print(animal_da_fit)
```

Repare que o print() mostra 2 endereços diferente de memória. OU seja, apesar de ser duas instanciações da mesma classe, cada instância ocupa um espaço diferente na memória, portanto são objetos diferentes. As instâncias dos novos tipos (classes) são mutáveis.

Uma instância de animal permite realizar chamadas ou definir atributos (características) por meio do operador ponto ".". Por exemplo, definir os seguintes atributos:

- Nome = Rex
- Tipo = Urso
- Cor = Branco

#Definição de 3 atributos (características) da instância

```
meu_animal.nome = "Rex"  
meu_animal.tipo = "Urso"  
meu_animal.cor = "Branco"
```

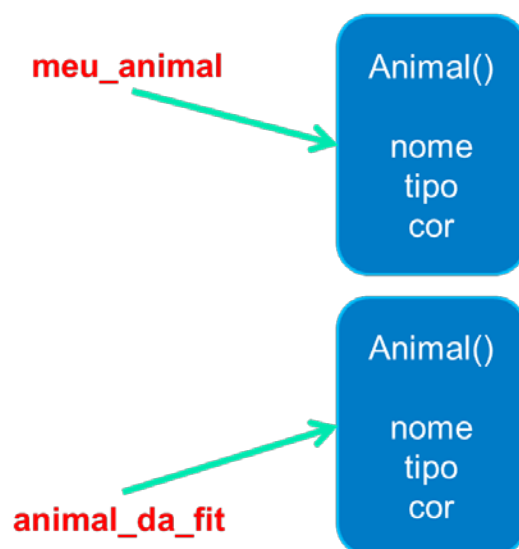
Como as instâncias das novas classes são tipos mutáveis, a criação de uma nova instância de animal e definição de suas características serão independentes. Cada instância ocupa um espaço diferente na memória, apesar de ter definidas as mesmas características.

#Definição de 3 atributos

```
meu_animal = Animal()  
meu_animal.nome = "Rex"  
meu_animal.tipo = "Urso"  
meu_animal.cor = "Branco"
```

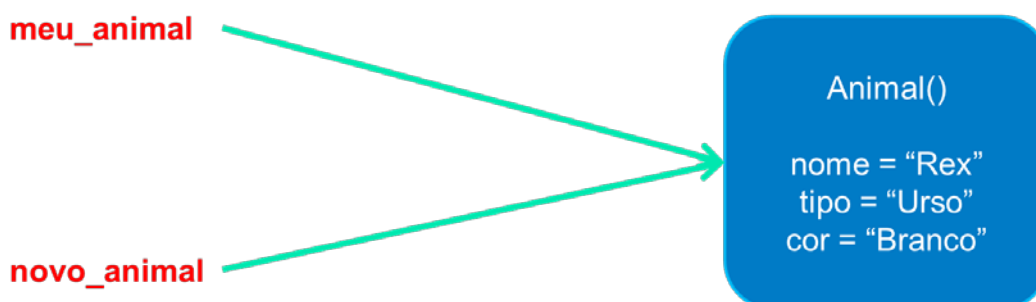
#Definição de 3 atributos

```
animal_da_fit = Animal()  
animal_da_fit.nome = "Policia"  
animal_da_fit.tipo = "Cachorro"  
animal_da_fit.cor = "Preto"
```



Assim como outros tipos de dados em Python, é possível que duas variáveis referenciem o mesmo objeto na memória. Basta que a nova variável receba a referência de um objeto já criado anteriormente.

```
#Atribuição da cópia da referência de meu_animal  
novo_animal = meu_animal
```



Neste caso, qualquer alteração no objeto `meu_animal`, por exemplo, `meu_animal.nome`, causará a alteração do objeto `novo_animal`, uma vez que são tipos mutáveis e referenciam o mesmo objeto na memória.

```
#Atribuição da referência meu_animal  
novo_animal = meu_animal
```

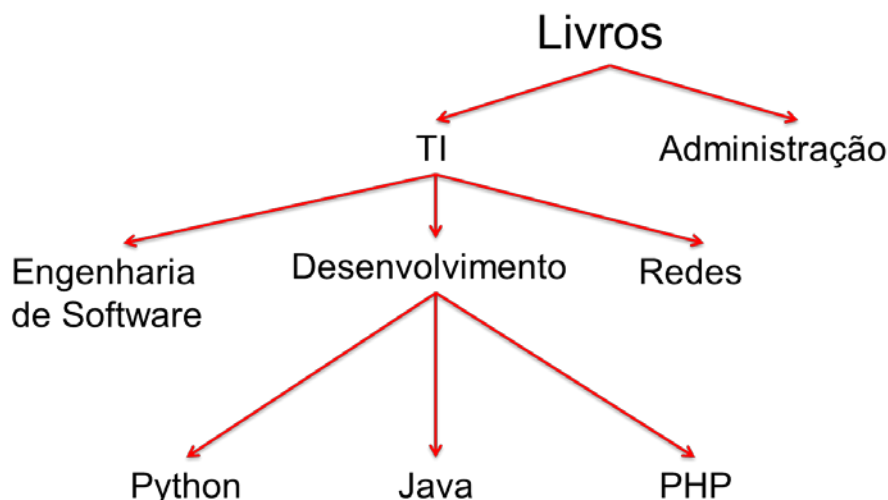
```
#Criação de 2 instâncias de Animal  
novo_animal.nome = "Pegasus"  
novo_animal.cor = "Cinza"
```

```
#mostrar a localização da memória para as instâncias  
print(meu_animal.nome)  
print(meu_animal.cor)
```

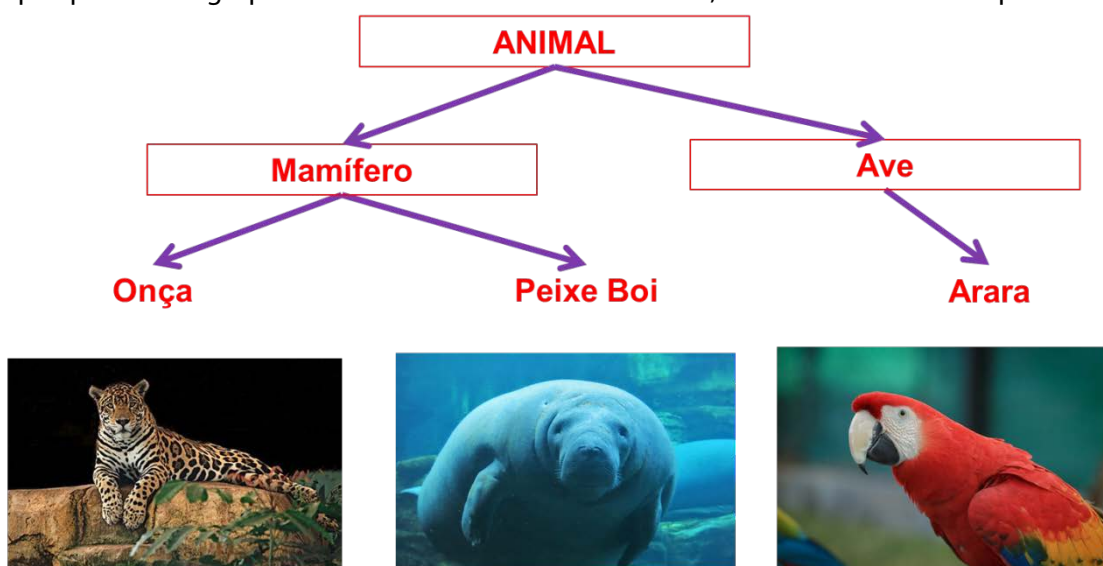
Obs: Neste momento, o nome e a cor do objeto `Animal()` alteraram seu valor, tanto para a referência `novo_animal` quanto para a referência `meu_animal`. Isto porque ambos referenciam o mesmo objeto.

7.2. Modelagem Orientada a Objetos

No capítulo anterior foram mencionados alguns conceitos relacionados à Programação Orientada a Objetos, e podemos trazer os objetos do mundo real para dentro de um programa. Além do mais, no mundo real é comum trabalharmos com classificação, agrupar coisas em categorias que tem algo em comum. Na biblioteca, por exemplo, podemos agrupar os livros de acordo com o tema: desenvolvimento, engenharia de software, redes, sistemas operacionais. Também é comum dividir em subcategorias, e estas em subcategorias. Dentro de desenvolvimento, podemos classificar por linguagens de programação: Python, Java, C#, php.



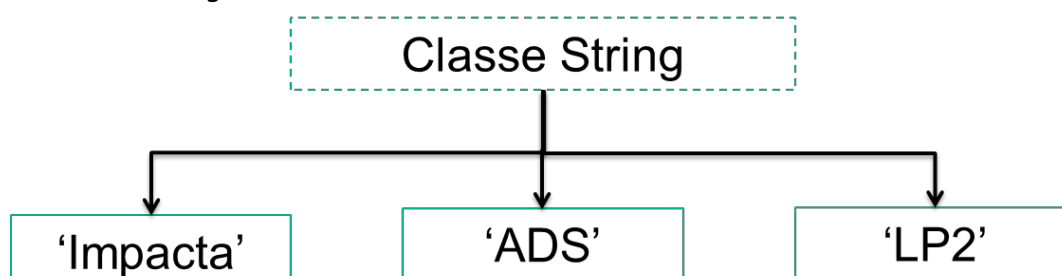
Outro exemplo: podemos agrupar os animais de acordo com a classe, e dentro da classe os tipos dos animais.



Nesta aula veremos como a organização do código em tipos e subtipos de classes é utilizada em Programação Orientada a Objetos. Estudamos na aula passada que tudo em Python é um objeto. Ou seja, utilizamos objetos todo o tempo na programação em Python. Entretanto, isso não significa que estamos utilizando programação orientada a objetos. Precisamos entender o que é uma classe e um objeto.

A Programação Orientada a Objetos (POO) facilita a escrita e manutenção dos programas através da definição de classes e da instanciação de objetos destas classes.

Uma classe é uma definição de um novo tipo de dado que não existe nos módulos do Python. Ela associa dados e operações em uma só estrutura, definindo um tipo de objeto, como se a classe fosse uma fôrma para fazer quantos objetos forem necessários. Por exemplo, str (string), list (listas) e dict (dicionário) são classes do Python. As strings criadas em um programa são instâncias de objetos da classe String. Por exemplo, 'Impacta', 'ADS', e 'LP2' são 3 objetos criados a partir da classe String.



Um objeto é como uma variável cujo tipo é uma classe. Em programação dizemos que um objeto é uma instância de uma classe. A classe String foi definida com uma série de métodos: lower, upper, format, etc. Cada objeto da classe String que for criado terá estes métodos e poderá ser utilizado. O método é sempre aplicado sobre o texto que está no objeto criado.

```
s1 = 'Impacta' # objeto String com o texto Impacta
s2 = 'ADS' # objeto String com o texto ADS

print(s1.lower()) # método lower de String
                  #aplicado em s1 (Impacta)

print(s2.lower()) # método lower de String
                  #aplicado em s2 (ADS)
```

Portanto, uma classe define um conjunto de atributos e métodos que podem ser utilizados pelos objetos da classe. Quando criamos a nova instância de string não é necessário implementar o método lower toda vez que for utilizar um texto. O método já está definido na classe e provê o comportamento para qualquer objeto desta classe. Lembre-se que uma classe é como se fosse uma fôrma.

Um objeto em Python (ou em qualquer outra linguagem orientada a objetos) pode ser pensado como uma representação de um objeto do mundo real: Faculdade, Pessoa, Aluno, Veículo, Carro, Roda, etc. Essa representação é limitada pela quantidade de detalhes que representamos conhecido como abstração. Podemos abstrair um Aluno com apenas com nome e número de matrícula. Em outra abstração a classe Aluno poderia conter também a idade. Ou seja, a abstração depende das necessidades do projeto.

7.2.1. Modelando uma Televisão em Python

Para exemplificar a modelagem de uma classe, sua abstração e como fazer isso em Python será utilizada uma Televisão. Primeiramente, quais as características de uma televisão? Ela pode ter cor, marca, modelo, tamanho, canal, etc. Depois o que podemos fazer com uma televisão, ou seja o comportamento? Ligar, desligar, mudar de canal, mudar o volume, sintonizar, etc.

Esse agrupamento de características e comportamento por meio de variáveis de instância (características) e métodos (comportamento) em um determinado tipo (classe, por exemplo, Televisão), é chamado de encapsulamento. É colocar as características e comportamento em um “pacote”. Este pacote é a classe Televisão.

No encapsulamento também é possível restringir o acesso a determinadas características, conhecido como características privadas. Em uma classe Televisão, ninguém pode alterar a marca ou o modelo, uma vez que poderia causar uma inconsistência. Portanto, encapsulamento é o meio agrupar variáveis de instâncias (atributos/características) e métodos (comportamento) em um determinado tipo, e também um meio de restringir o acesso a determinados membros de uma classe.

Uma classe em Python, com suas características e comportamentos é criada da seguinte forma:

```
class Televisao:
    def __init__(self):
        self.ligada = False
        self.canal = 2
```

Abra um novo projeto no VS Code, crie um novo arquivo chamado `televisoes.py` e coloque o código acima. Pronto, você criou sua primeira classe, `Televisao`, com as características `ligada` e `canal`.

A criação de uma classe simples é feita da seguinte forma:

- A instrução `class` é utilizada para indicar a declaração de uma nova classe. Ao declarar uma classe estamos criando um novo tipo de dados;
- Após `class`, coloca-se o nome da classe (por exemplo, `Televisao`) e finaliza com `:` para iniciar o bloco da classe;
- Como qualquer bloco de execução em Python, este deve ser indentado;
- Os atributos e métodos da classe ficaram no corpo da classe

No corpo do código de exemplo foi colocado um primeiro método especial, `__init__`, conhecido como construtor. Seu nome deve conter 2 sublinhados no começo e 2 sublinhados no fim. O construtor é executado sempre que uma instância da classe for criada, e serve pra iniciar o novo objeto com as características padrões. O método `__init__`, assim como qualquer outro método de classes, recebe um parâmetro `self`, que vai conter uma referência ao próprio objeto criado.

Dentro do construtor os atributos do objeto, ou variáveis de instância são iniciados. Ao utilizar `self.ligada`, por exemplo, estamos referenciando o atributo `ligada` do próprio objeto (`self`). Portanto, `self.ligada` é um atributo do objeto. Sempre que quisermos referenciar um atributo do objeto, utiliza-se com o `self`. Caso `self` não seja utilizada, será apenas uma variável local.

```
self.ligada = False # atributo do objeto  
ligada = True # variável local
```

Até agora a classe foi apenas definida. É preciso instanciar objetos da classe para poder utilizá-la. Crie um segundo arquivo no seu projeto chamado `main.py`, faça a importação da classe e instancie um objeto da seguinte forma:

```
from televisoes import Televisao # importação do módulo  
  
tv = Televisao() # instanciação do novo objeto
```

No exemplo, `tv` é um objeto da classe `Televisao`, ou seja, `tv` é uma instância de `Televisao`. Com a instância criada, é possível acessar os atributos da classe, como `ligada` e `canal`.

```
print(tv.ligada)
```

```
print(tv.canal)
```

Uma vez que a classe foi definida, é possível criar quantas instâncias da classe `Televisao` forem necessárias. Cada instância será independente, ou seja, terão seus próprios atributos e métodos, definidos pela classe. Por exemplo, uma nova instância de `Televisao` é armazenada na variável `tv_sala`.

```
tv_sala = Televisao()
```

Quando os atributos `ligada` e `canal` dessa nova instância são alterados, os atributos de mesmo nome da primeira instância permanecem inalterados.

```
tv_sala.ligada = True  
tv_sala.canal = 5
```

```
print(tv_sala.ligada)  
print(tv_sala.canal)  
print(tv.ligada)  
print(tv.canal)
```

Faça um paralelo com o mundo real: ao desligar a TV da sala, não faz sentido a TV do quarto ser desligada também. As duas Televisões podem ser idênticas (mesma marca, mesmo modelo, mesma quantidade de canais, mesmo tamanho), mas elas são independentes, ou seja, cada uma teve seu processo de fabricação, apenas tem o mesmo molde. Ao transpor isso para objetos de programação, a ideia é a mesma. Quando um objeto é instanciado (fabricado) ele é independente de qualquer outro objeto instanciado da mesma classe, mesmo que suas características iniciais sejam iguais.

Ao criar um objeto de uma classe, ele tem todos os atributos e métodos da declaração da classe. Os atributos podem ser iniciados com o construtor. Essa característica simplifica o desenvolvimento dos programas, uma vez que se pode definir o comportamento de todos os objetos de uma classe (métodos), preservando os característicos individuais (atributos).

métodos == comportamento **atributos == características**

A classe Televisao já tem algumas características. Agora falta colocar o comportamento, ou seja, os métodos. Os métodos de uma classe são como funções, e são escritos da mesma forma. A diferença é que estarão dentro do corpo da classe, e serão conhecidos apenas por instâncias dessa classe. Para definir um método em uma classe:

- Utilize a instrução `def`;
- Coloque o nome do método, com os parêntesis;
- Coloque os dois pontos `:`;
- O corpo do método vai em um bloco indentado

Outra diferença existe entre os métodos de classes e as funções: todo método deve receber como primeiro parâmetro `self`, assim como no construtor. O `self` recebe uma referência do objeto que está sendo manipulado, assim podemos acessar as variáveis de instância. Veja no exemplo abaixo a definição dos métodos `aumenta_canal` e `diminui_canal`.

```
class Televisao:  
    def __init__(self):  
        self.ligada = False  
        self.canal = 2  
  
    def aumenta_canal(self):  
        self.canal += 1  
  
    def diminui_canal(self):  
        self.canal -= 1
```


Repare que para acessar a variável de instância foi utilizado `self.canal`. Se fosse utilizado somente `canal`, não estaria referenciando a variável da instância, mas sim uma variável local.

Para ser executado, o método deve ser chamado, assim como uma função. A chamada é feita utilizando a variável que armazena a instância e o nome do método, separados por pontos. É a mesma forma utilizada para chamar métodos de string e listas (`string.lower()`, `lista.sort()`). No módulo `main.py`, adicione os seguintes códigos para chamar os métodos de `Televisao`:

```
tv.aumenta_canal()
tv.diminui_canal()
```

Repare que na chamada não há a passagem de nenhum parâmetro. A instância da `Televisao` (`self`) é colocada automaticamente pelo interpretador. Também não é preciso enviar o número do canal como parâmetro. O canal utilizado está na própria classe. Na modelagem orientada a objetos, características importantes ficam armazenadas nos atributos da classe, evitando passá-los como parâmetro na chamada do método.

7.2.2. Construtores

Na classe `Televisao` não existe controle de quantos canais podem existir. O menor e o maior canal são características de uma televisão configuradas de fábrica, ou seja, quando ela é construída. Em orientação a objetos, um método construtor pode receber parâmetros como qualquer outro método, basta declará-los logo após o `self`.

```
class Televisao:
    def __init__(self, min, max):
        self.ligada = False
        self.canal = 2
        self.cmin = min
        self.cmax = max
```

No exemplo anterior, `min` e `max` são os parâmetros do construtor. Portanto, na instanciamento da do objeto `Televisao`, eles devem ser enviados como parâmetros.

```
tv_sala = Televisao(2,50)
```

Existem situações que é necessário instanciar 2 objetos da mesma classe de formas diferentes. Por exemplo, a classe pode estar preparada para instanciar objetos cuja quantidade de canais é fixa ou que não tenha limite de canais. Portanto, seria necessário 2 formas de construir um objeto da classe:

- Um que não recebe nenhum parâmetro além do `self`;
- Outro que recebe o canal mínimo e o canal máximo, além do `self`.

Em Python não é possível definir 2 métodos com o mesmo nome. Para que a situação anterior seja possível, devemos colocar parâmetros opcionais no construtor, da seguinte forma:


```

class Televisao:
    def __init__(self, min=None, max=None):
        self.ligada = False
        self.canal = 2
        if min != None:
            self.cmin = min
        if max != None:
            self.cmax = max

tv = Televisao() # objeto criado com construtor padrao
tv_sala = Televisao(2,50) # objeto criado com o\
                        construtor com parâmetros

```

7.2.3. Atributos Privados

O menor e o maior canal são características de uma televisão que não podem ser alteradas inadvertidamente. São características configuradas de fábrica. Estes atributos/características devem ser protegido para que não sejam modificados por alguma operação externa à classe. A proteção é feita utilizando atributos privados.

Um atributo privado de uma classe em Python é nomeado com dois sublinhados (__) no começo do nome do atributo:

- Canal máximo: `__cmax`
- Canal mínimo: `__cmin`

```

class Televisao:
    def __init__(self, min=None, max=None):
        self.ligada = False
        self.canal = 2
        if min != None:
            self.__cmin = min
        if max != None:
            self.__cmax = max

```

Como posso ter certeza que funcionará? Vamos fazer os seguintes testes:

- Crie uma nova instância de TV utilizando o construtor com parâmetros:

```
tv_sala = Televisao(2,50)
```

- Acesse a variável de classe canal (que não é privada) e mostre seu valor: funciona corretamente

```
print(tv_sala.canal)
```

- Agora tente acessar a variável de classe privada `__cmin`. Será lançado um erro

```
print(tv_sala.__cmin) #erro. Atributo privado
```

- Agora altere o valor de `__cmin`, e depois mostre o valor.

```
tv_sala.__cmin = 20  
print(tv_sala.__cmin)
```

Funcionou? Sim! Mas o que foi alterado não foi o atributo privado `__cmin`. Python cria outro `__cmin` para poder alterar. Portanto, cuidado ao utilizar e acessar atributos privados!

Usualmente, os atributos privados têm associados dois métodos: um para alterar (set) e um para retornar (get). Para terminar o exemplo anterior, crie um método na classe `Televisao` chamado `get_cmin`, que retorna o canal mínimo da Televisão e veja que o valor é diferente do que foi atribuído no comando `tv_sala.__cmin = 20`

```
class Televisao:  
    def __init__(self, min=None, max=None):  
        self.ligada = False  
        self.canal = 2  
        if min != None:  
            self.__cmin = min  
        if max != None:  
            self.__cmax = max  
  
    def get_cmin(self):  
        return self.__cmin  
  
tv = Televisao(2,50)  
print(tv.canal)  
tv.__cmin = 20  
print("__cmin alterado fora do objeto: ", tv.__cmin)  
print("__cmin do objeto: ", tv.get_cmin())
```

7.2.4. Métodos com parâmetros

Existem situações em que é necessário enviar parâmetros para os métodos da classe. Por exemplo, escolher qual canal deseja ver. Crie o método `muda_canal(canal)` que recebe como parâmetro o canal desejado pelo usuário e altera o atributo `canal` da classe. Vale lembrar que o primeiro parâmetro de qualquer método da classe sempre será o `self`, independentemente se houver outros parâmetros ou não. Na chamada do método, também não é preciso enviar o `self` (será colocado automaticamente pelo interpretador).

```
class Televisao:
    def __init__(self, min=None, max=None):
        self.ligada = False
        self.canal = 2
        if min != None:
            self.cmin = min
        if max != None:
            self.cmax = max

    def aumenta_canal(self):
        if self.canal == self.__cmax:
            self.canal = self.__cmin
        else:
            self.canal += 1

    def diminui_canal(self):
        if self.canal == self.__cmin:
            self.canal = self.__cmax
        else:
            self.canal -= 1

    def muda_canal(self, canal):
        self.canal = canal


tv = Televisao(2,50)
tv.muda_canal(10)
print(tv.canal)
```

7.2.5. Considerações

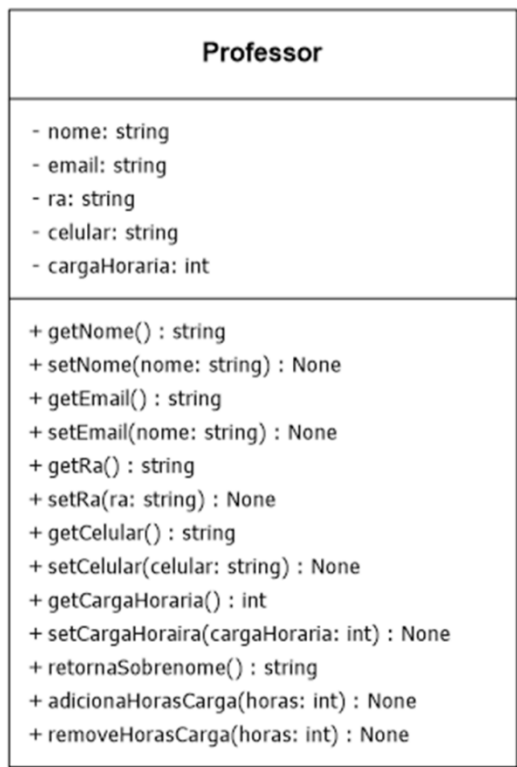
Trabalhar com classes significa representar em Python uma abstração do problema. Uma abstração reduz os detalhes do problema ao necessário para sua solução. Fazendo isso estamos construindo um modelo, ou seja, modelando classes e objetos. Cada pessoa modela de uma forma. Uma das partes mais difíceis é decidir o quanto representar e onde limitar o modelo. No nosso modelo de TV não colocamos nada sobre tomada, volume, controle remoto, etc. Dica: modele apenas as informações que precisa, acrescentando conforme a necessidade.

Métodos de classes são como funções, ou seja pode aproveitar tudo que aprendeu. As principais diferenças são:

- Um método é associado a uma classe e atua em um objeto;
- O primeiro parâmetro de um método é sempre self, e representa a instância sobre a qual o método atua;
- Na chamada não é preciso enviar self como parâmetro. Isso é feito automaticamente pelo interpretador.

7.3. Exemplo 2 – Modelando Professor, Aluno e Disciplina

Para fixar melhor o conteúdo, vamos modelar 3 objetos: Professor, Aluno e Disciplina, e fazer o relacionamento entre eles. A Modelagem de aluno ficará como tarefa, e você conseguirá fazer sem problemas após acompanhar este exemplo. Vamos começar modelando a classe Professor a partir do seguinte diagrama de classes:



Algumas regras:

- Todos os atributos da classe são privados;
- A classe deve possibilitar 2 formas construtores: 1 sem nenhum argumento e outro com argumentos para iniciar todos os atributos privados;
- Os métodos get devem retornar os valores dos seus respectivos atributos privados;
- Os métodos set devem alterar os valores dos respectivos atributos privados;
- O método `retornaSobrenome` deve retornar apenas o sobrenome do professor (baseado no atributo `nome`);
- O método `adicionaHorasCarga(horas)` deve adicionar a quantidade de horas do argumento `horas` no atributo `carga_horaria`. Caso a carga horária final passe de 40 horas, não deve adicionar e retornar uma mensagem de erro;
- O método `removeHorasCarga(horas)` deve remover a quantidade de horas do argumento `horas` do atributo `carga_horaria`. Caso a carga horária final seja menor que 0, não deve diminuir e retornar uma mensagem de erro.

class Professor:

```
def __init__(self, nome = None, email = None, ra = None, celular = None, cargaHoraria = 0):
```

```
    self.__nome = nome
```

```
    self.__email = email
```

```
    self.__ra = ra
```

```
    self.__celular = celular
```

```
    self.__cargaHoraria = cargaHoraria
```

```
def getNome(self):
```

```
    return self.__nome
```

```
def setNome(self, nome):
```

```
    self.__nome = nome
```

```
def getEmail(self):
```

```
    return self.__email
```

```
def setEmail(self, email):
```

```
    self.__email = email
```

```
def getRa(self):
```

```
    return self.__ra
```

```
def setRa(self, ra):
```

```
    self.__ra = ra
```

```
def getCelular(self):
```

```
    return self.__celular
```

```
def setCelular(self, celular):
```

```
    self.__celular = celular
```

```
def getCargaHoraria(self):
```

```
    return self.__cargaHoraria
```

```
def setCargaHoraria(self, cargaHoraria):
```

```
    self.__cargaHoraria = cargaHoraria
```

```
def retornaSobrenome(self):
```

```
    return ''.join(self.__nome.split(' ')[1:])
```

```
def adicionaCargaHoraria(self, horas):
```

```
    total = self.__cargaHoraria + horas
```

```
    if total > 40:
```

```
        return "Quantidade de horas inválida. SoCarga horária será maior que 40"
```

```
    self.__cargaHoraria = total
```

```
def removeCargaHoraria(self, horas):
```

```
    total = self.__cargaHoraria - horas
```

```
    if total < 40:
```

```
        return "Quantidade de horas inválida. Carga horária será menor que 0"
```

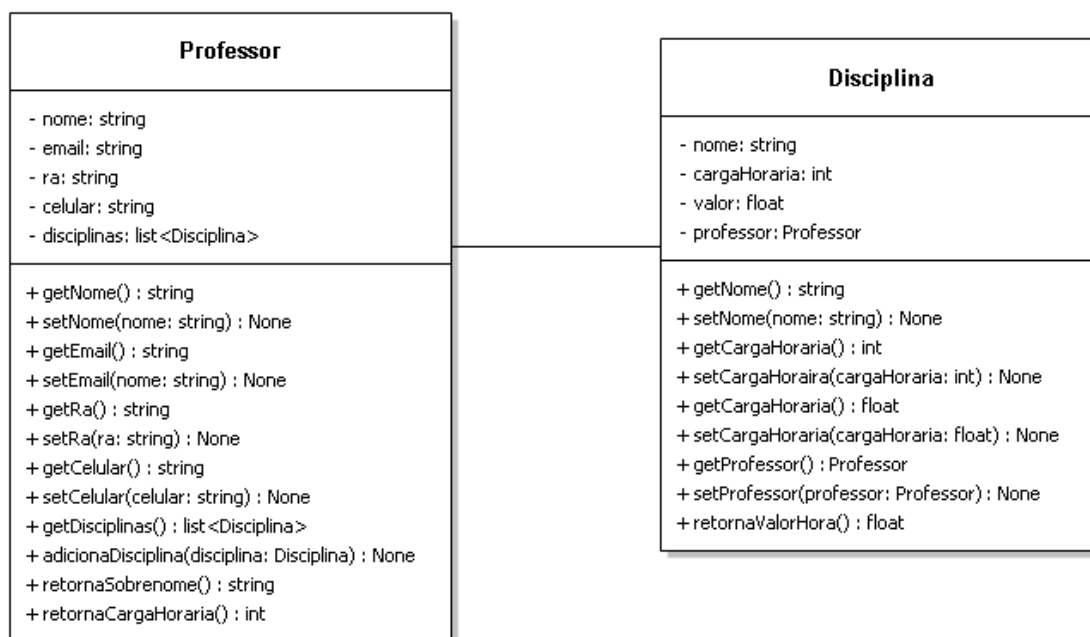
```
    self.__cargaHoraria = total
```

7.3.1.

Associação de classes

Durante a modelagem de um sistema, é muito comum fazer a associação de classes. Ou seja, os atributos de uma classe referenciam instâncias de objetos de outras classes. Por exemplo: um Professor leciona uma ou mais disciplinas. Portanto, existe um atributo da classe Professor que será uma lista de elementos de outra classe: Disciplina. A Disciplina também pode ter uma referência para a classe Professor.

Vamos alterar a modelagem para: incluir a classe Disciplina, de modo que um professor pode ministrar várias disciplinas, mas uma disciplina será ministrada por apenas um professor; remover a carga horária do professor para colocar na disciplina, calcular a carga horária de cada professor de acordo com o que está na disciplina. Siga o seguinte diagrama de classes:



Considere ainda as seguintes regras para a classe Disciplina:

- Todos os atributos são privados
- Deve possibilitar 2 formas construtores: 1 sem nenhum argumento e outro com argumentos para iniciar todos os atributos privados
- Os métodos get devem retornar os valores dos seus respectivos atributos privados
- Os métodos set devem alterar os valores dos respectivos atributos privados
- O método retornaValorHora deve retornar o valor cobrado pela hora da disciplina, considerando 6 meses de

Considere também as seguintes regras para a classe Professor:

- Não deve haver o campo cargaHoraria;
- O construtor não deve aceitar a disciplina;
- O atributo disciplina tem somente o método get;
- Utilize o método adicionaDisciplina para adicionar uma disciplina na lista associada ao professor. Só deve adicionar se o professor for o mesmo da instância (verificando pelo RA). Caso contrário, retorna uma mensagem de erro;

- O método `retornaCargaHoraria` deve retornar a carga horária semanal total do professor, baseado nas disciplinas que leciona e em 20 semanas de aulas.

class Professor:

```
def __init__(self, nome = None, email = None, ra = None, celular = None):
```

```
    self.__nome = nome
```

```
    self.__email = email
```

```
    self.__ra = ra
```

```
    self.__celular = celular
```

```
    self.__disciplinas = []
```

```
def getNome(self):
```

```
    return self.__nome
```

```
def setNome(self, nome):
```

```
    self.__nome = nome
```

```
def getEmail(self):
```

```
    return self.__email
```

```
def setEmail(self, email):
```

```
    self.__email = email
```

```
def getRa(self):
```

```
    return self.__ra
```

```
def setRa(self, ra):
```

```
    self.__ra = ra
```

```
def getCelular(self):
```

```
    return self.__celular
```

```
def setCelular(self, celular):
```

```
    self.__celular = celular
```

```
def retornaSobrenome(self):
```

```
    return ''.join(self.__nome.split(' ')[1:])
```

```
def getDisciplinas(self):
```

```
    return self.__disciplinas
```

```
def adicionaDisciplina(self, disciplina):
```

```
    if disciplina != None:
```

```
        professor = disciplina.getProfessor()
```

```
        if professor != None and professor.getRa() == self.__ra:
```

```
            self.__disciplinas.append(disciplina)
```

```
        else:
```

```
            return "Disciplina não pertence ao professor"
```

```
    else:
```

```
        return "Disciplina não pertence ao professor"
```

```
def retornaCargaHoraria(self):  
    cargaTotal = 0  
    for disc in self.__disciplinas:  
        cargaTotal += disc.getCargaHoraria() / 20  
    return cargaTotal
```

```
class Disciplina:
```

```
    def __init__(self, nome = None, cargaHoraria = 0, mensalidade = 0, professor= None):  
        self.__nome = nome  
        self.__cargaHoraria = cargaHoraria  
        self.__mensalidade = mensalidade  
        self.__professor = professor
```

```
    def getNome(self):  
        return self.__nome
```

```
    def setNome(self, nome):  
        self.__nome = nome
```

```
    def getCargaHoraria(self):  
        return self.__cargaHoraria
```

```
    def setCargaHoraria(self, cargaHoraria):  
        self.__cargaHoraria = cargaHoraria
```

```
    def getMensalidade(self):  
        return self.__mensalidade
```

```
    def setMensalidade(self, mensalidade):  
        self.__mensalidade = mensalidade
```

```
    def getProfessor(self):  
        return self.__professor
```

```
    def setProfessor(self, professor):  
        self.__professor = professor
```

```
    def retornaValorHora(self):  
        valor_hora = (self.__mensalidade * 6) / self.__cargaHoraria  
        return valor_hora
```


8. Herança e Polimorfismo

Já estudamos como funciona um programas orientados a objetos (OO) e como definir classes em Python. Muitos dos sistemas são desenvolvidos utilizando OO. Entender conceitos como encapsulamento, polimorfismo e herança são essenciais para a modelagem correta de sistemas. Um sistema modelado corretamente facilita sua manutenção e reuso de componentes.

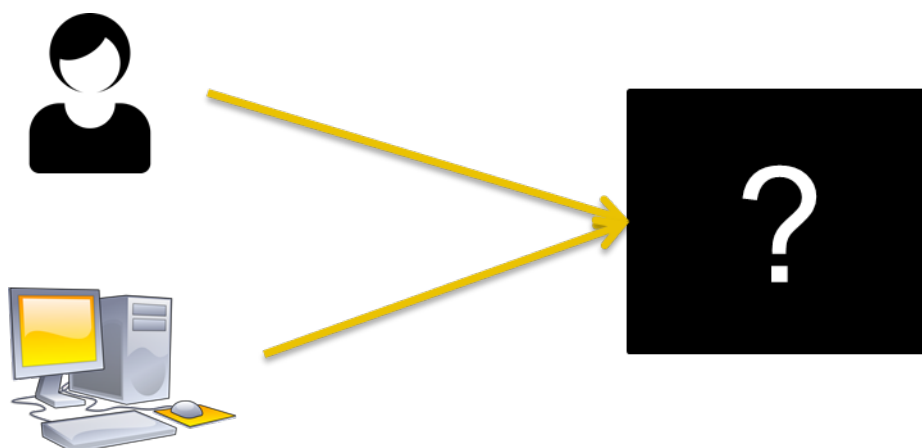
- ✓ Entender o conceito de encapsulamento;
- ✓ Entender o conceito de Herança;
- ✓ Entender o conceito de Polimorfismo;
- ✓ Desenvolver programas em Python que utilizam herança e polimorfismo

Quando utilizamos um software podemos pensar nele como uma caixa preta. Você sabe o que ele faz, mas não como.



Pense por exemplo no sistema que você acessa sua conta bancária. Você sabe quais os serviços que o sistema oferece: login, extrato, transferência, pagamento, etc. Mas para o usuário, a forma como cada serviço é implementado vai mudar a forma de interação usuário/ serviço?

No mundo do desenvolvimento dizemos que o sistema provê uma série de serviços através de uma interface. Uma interface descreve o que o serviço faz, mas não como ele faz. Quem faz uso dessa interface é conhecido como cliente. Um cliente pode ser um usuário final, um módulo do sistema ou até mesmo outra aplicação.



Outro exemplo é uma loja virtual. Quando um pagamento de uma compra em uma loja virtual é realizado, o sistema

da loja utiliza um componente de pagamento da operadora de cartão de crédito. O sistema da operadora oferece uma série de serviços através de interfaces, que informam o que o sistema faz, mas não como. Ou seja, encapsula uma tarefa. O sistema da loja virtual conhece o que o sistema da operadora faz através da interface. Não importa como a operadora implementa a transação.

Usualmente os detalhes internos da implementação de um serviço podem mudar radicalmente sem afetar o cliente. Da mesma forma, o componente que implementa o serviço não precisa considerar como o serviço será utilizado. A única restrição é ter certeza que o serviço será provido. Esta separação, conhecido como baixo acoplamento, é o que possibilita a criação de sistemas complexos.

Na OO, essa caixa preta são os objetos, definidos através das classes. Quando definimos uma classe, podemos ignorar como a classe funciona e trabalhar apenas com suas interfaces – ou seja, seus métodos.

Se conseguirmos quebrar um problema muito grande em uma série de Classes, reduzimos bastante a complexidade necessária para entender uma pequena parte do problema. Isso é conhecido como “dividir para conquistar”. Modelagem orientada a objetos é o processo de encontrar e definir classes úteis para o problema.

Antes de começar a modelar um projeto orientado a objetos é importante entender alguns conceitos relacionados à OO. Os três principais são: Encapsulamento, Polimorfismo e Herança.

8.1. Encapsulamento

Objetos conhecem coisas e fazem coisas, ou seja, contém dados (atributos) e operações (métodos). O processo de empacotar dados e operações executadas nestes dados é chamado de encapsulamento. É exatamente a forma como vemos o mundo. No mundo existem objetos que interagem entre si. Cada objeto tem sua própria identidade. Entendendo o tipo do objeto possibilita entender sua capacidade.

As principais vantagens do encapsulamento são:

- Separação do ‘o que’ será feito (interface, métodos) do ‘como’ será feito (implementação dos métodos)
- Isolamento das funcionalidades através das interfaces
- A implementação de um objeto fica independente da forma do seu uso
- A mudança de uma implementação não afeta o uso do objeto
- A implementação pode mudar, mas se a interface continuar a mesma, outros componentes que dependem do objeto não serão afetados
- Reuso de código

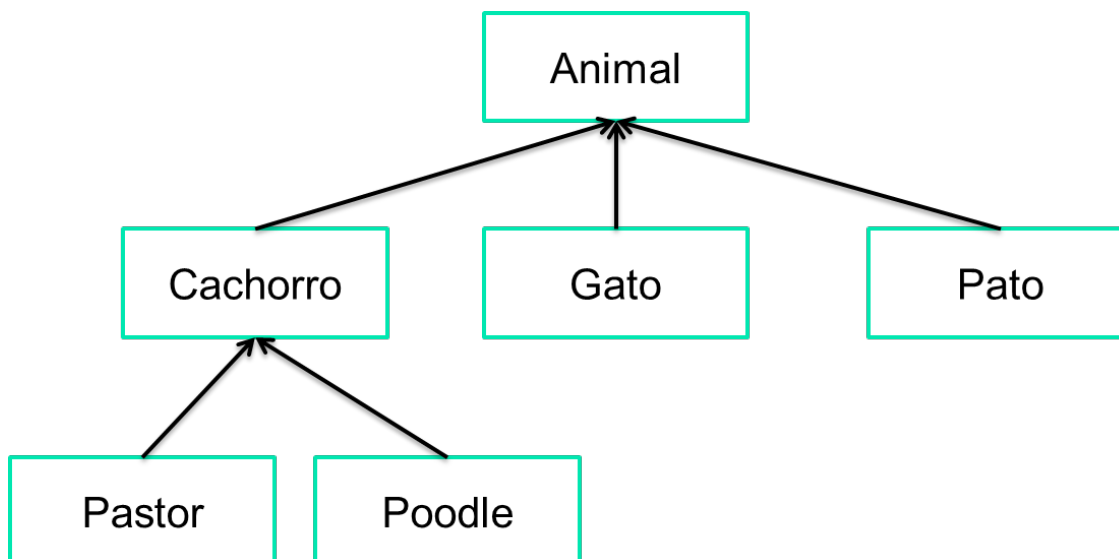
Vamos voltar ao exemplo da Televisão. A classe Televisao encapsula o que uma televisão faz. Faz sentido a classe Televisao ter um método enxaguar? De qual classe poderia ser este método?

Outro exemplo, do modelo do LMS. A classe Professor contém uma associação com Disciplina. Para saber a carga horária total, é preciso utilizar o método getCargaHoraria() da classe disciplina. Se a classe Disciplina mudar a forma como retorna a carga horária, por exemplo, nada precisa ser alterado na classe Professor para que o programa continue funcionando. Não é tarefa do Professor saber o a carga horária de cada disciplina

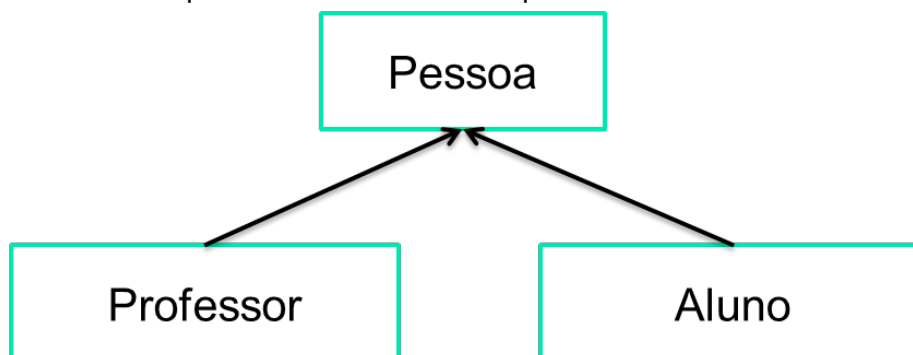
```
class Professor:
    def retornaCargaHoraria(self):
        cargaTotal = 0
        for disc in self.__disciplinas:
            cargaTotal += disc.getCargaHoraria() / 20
        return cargaTotal
```

8.2. Herança

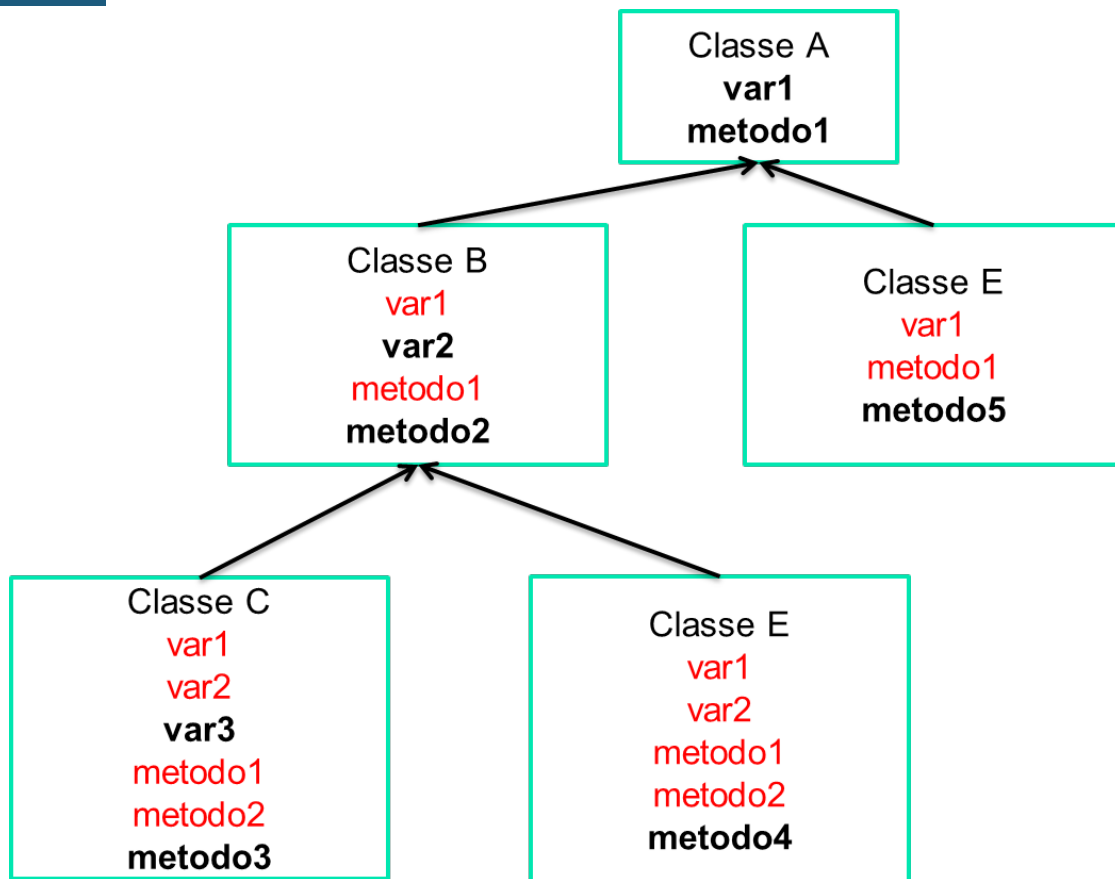
Aplicar Herança em um projeto orientado a objetos mostra a real capacidade da programação orientada a objetos. A ideia da herança é que uma nova classe pode ser definida emprestando (herdando) características e comportamento de outra classe, como se fossem da sua própria definição. A nova classe que herda, é chamada de subclasse ou classe filha. A classe existente, herdada, é chamada de superclasse, ou classe mãe (ou classe pai). A superclasse pode também herdar de outras classes, formando uma hierarquia de classes. Por exemplo, Cachorro, Gato e Pato podem ser subclasses da superclasse Animal; Pastor e Poodle podem ser subclasses de Cachorro; Cachorro é subclasse animal e superclasse de Pastor e Poodle.



Outro exemplo, Professor e Aluno podem ser subclasses da superclasse Pessoa.



Exemplo geral (membros herdados estão em vermelho):



Usualmente a superclasse contém características e comportamentos que são comuns a todas as subclasses:

- Todas as pessoas no LMS tem nome;
- Todas as pessoas no LMS retornam a carga horaria;
- Todas as pessoas no LMS retornam o sobrenome.

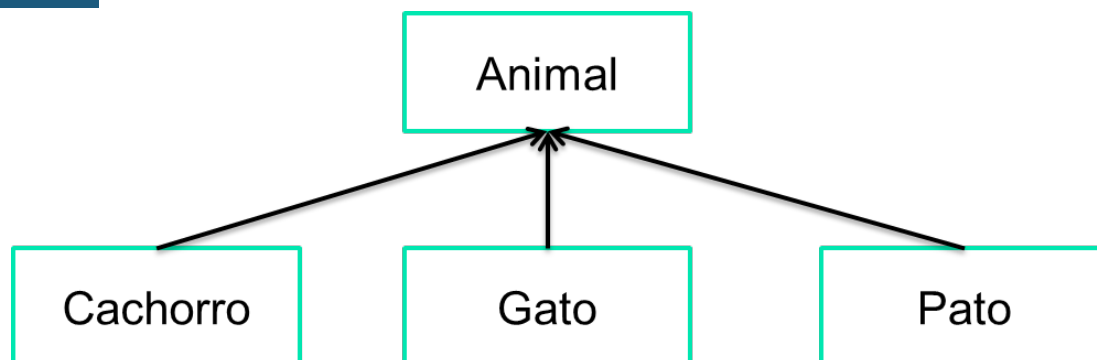
Professor e Aluno compartilham as características e comportamentos anteriores. Entretanto, Aluno pode ter um método `retornaValorMensalidade()`, que não está presente em Professor.

Os principais benefícios da herança são:

- Classes de um sistema podem ser estruturadas para evitar duplicidade de operações;
- Não é preciso escrever duas operações de para retornar o sobrenome para Professor e Aluno se elas realizarem o mesmo tipo de operação;
- Novas classes podem ser baseadas em classes existentes, promovendo o reuso de código.

8.3. Subtipos

Um subtipo é algo que pode substituir e agir como o tipo pai (e o tipo pai, e o tipo pai...). Considere novamente a hierarquia de animais. Cachorro, gato e pato têm todas as características de animais, uma vez que todos são animais.



Para entender melhor, considere a seguinte situação: “Fulano gosta muito de animais. Ele tem muitos livros sobre o assunto e sempre que pode sai para ver animais”.

Como Cachorro É UM animal (subtipo de animal), podemos reescrever a frase trocando animal por cachorro que continua fazendo sentido: “Fulano gosta muito de cachorros. Ele tem muitos livros sobre o assunto e sempre que pode sai para ver cachorros”.

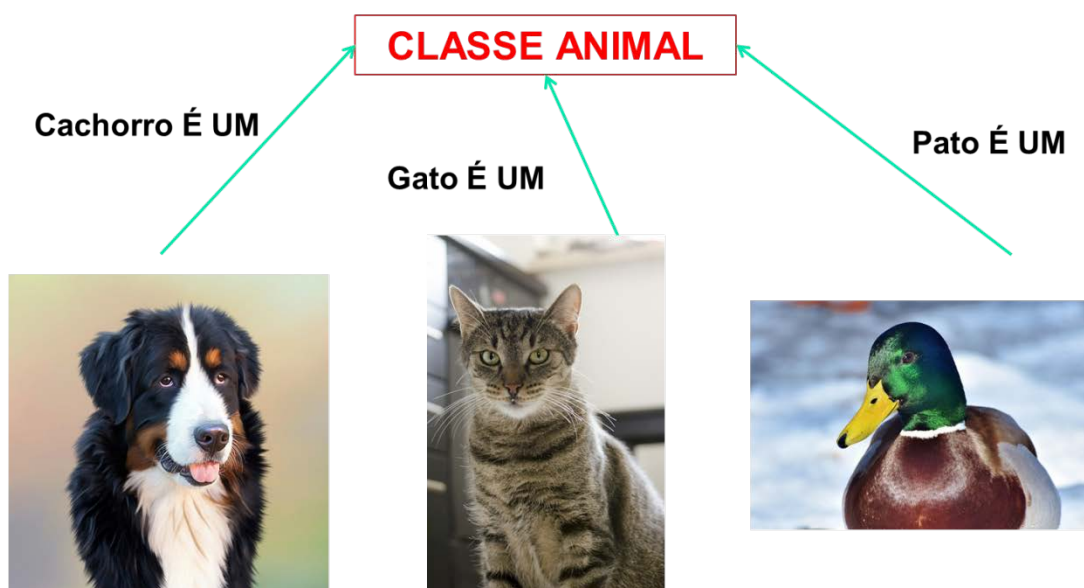
Agora vamos trocar a frase pela seguinte: “Fulano tem muito medo de cachorros porque não gosta do barulho que fazem quando latem”. Se trocarmos cachorro por pato, a frase fica sem sentido: “Fulano tem muito medo de patos porque não gosta do barulho que fazem quando latem”. Isso acontece porque pato não é subtipo de cachorro. Pato não é um cachorro.

8.4. Polimorfismo

Polimorfismo significa muitas formas. Ou seja, um objeto pode assumir diferentes formas. Em OO, isso mostra que o comportamento de um objeto em resposta a uma chamada depende do tipo do objeto.

Voltado ao exemplo do LMS, Professor e Aluno são Pessoas e compartilham características em comum, como nome, e-mail e ra e a carga horária. Entretanto, a operação de retornaCargaHoraria() deve ser diferente para cada um deles. A carga horária do aluno está relacionada à sua carga horária semestral, enquanto do professor é semanal. Portanto, uma Pessoa pode ter várias formas, Aluno e Professor.

Outro exemplo, que vimos no capítulo anterior é a classe Animal. Um animal pode assumir várias formas, como cachorro, gato ou pato.



Animal é uma classe genérica, que tem características e comportamentos comuns a todos os animais. Cachorro é um animal, Gato é um animal, Pato é um animal. Animal poderia, por exemplo, ter um método falar(), supondo que

todos os animais falam. Mas o comportamento falar() da cachorro é diferente do comportamento falar do gato ou do pato.

8.5. Herança e Polimorfismo em Python

Para mostrar o uso de herança e polimorfismo em Python, vamos retomar a modelagem do LMS

No LMS temos duas classes, Professor e Aluno, que tem características e comportamentos em comum:

- Nome;
- E-mail;
- RA;
- Telefone;
- Disciplinas;
- Métodos getters e setters;
- retornaCargaHoraria;
- adicionaDisciplina.

Dessa forma, podemos criar uma superclasse, por exemplo, Pessoa, que tem todas as características comuns entre Professor e Aluno. Assim evitamos a duplicidade do código. Por exemplo, o método retornaSobrenome() está implementado nas duas classes e fazem a mesma coisa! Porque não colocá-lo em uma superclasse, e aproveitá-lo por herança?

Primeiro, crie um novo arquivo pessoas.py e defina classe Pessoa com:

- Um construtor para iniciar os atributos de classe nome, e-mail, ra, celular e disciplinas;
- Métodos get e set para todos os atributos, menos disciplinas
- Método get para a disciplina
- Método retornaSobrenome()

A implementação destes métodos deve ser igual ao que foi implementado em Aluno e Professor.

Juntando com que fizemos na aula de Classes, nosso modelo tem 3 classes: Professor, Aluno, Pessoa

Temos que fazer agora Professor e Aluno herdarem as características de Pessoa. Todos os atributos e métodos comuns ficarão na superclasse. Subclasses vão conter apenas métodos e atributos específicos do subtipo.

Para utilizar herança em Python, basta colocar o nome da superclasse entre parêntesis, após o nome a subclasse, da seguinte forma:

```
class Professor (Pessoa):      class Aluno (Pessoa):  
  
    # corpo da classe          corpo da classe
```

Ao fazer isso, todas as características e comportamento de Pessoa estarão presentes em aluno. Lembre-se de que se cada classe estiver em arquivos separados, temos que importar os módulos (from pessoas import Pessoa)

Para ver a herança funcionando nas classes Aluno e Professor:

- Remova o construtor;
- Remova os métodos get e set para cada um dos atributos de classe
- Deixe apenas os métodos adicionaDisciplina e retornaCargaHoraria para Professor
- Deixe apenas os métodos adicionaDisciplina, retornaCargaHoraria, aumentaDesconto, diminuiDesconto e retornaValorMensalidade para Aluno.

Teste agora o funcionamento da herança. No módulo main.py, crie uma instância de Professor e uma de Aluno, enviando o nome como parâmetro do construtor e mostre o nome associado a cada instância.

```
from alunos import Aluno
from professores import Professor

p = Professor('Fernando')
print(p.getNome())

a = Aluno('André')
print(a.getNome())
```

Veja que não foi necessário especificar o construtor nas subclasses. O construtor da superclasse é chamado automaticamente. Ele também vem por herança. Todos os métodos existentes em pessoa, existem também em professor e aluno.

Uma subclasse pode definir e implementar os próprios métodos. Por exemplo, a classe aluno tem o método `retornaValorMensalidade()`, que não está presente em professor. Portanto, se a instância for do tipo Aluno, ela pode chamar este método. Caso contrário, o método não existirá.

Suponha que os alunos só podem se matricular em 2 disciplinas por vez. A classe Aluno pode ter um atributo que é válido para qualquer instância. Ou seja, independente da instância de aluno criada qualquer aluno só poderá se matricular em duas disciplinas. Não faz sentido colocar este dado como atributo da instância. Isso é conhecido como variável estática. Ou seja, independentemente da instância (objeto) da classe, seu valor é o mesmo. Para criar uma variável estática basta defini-la e iniciá-la dentro da classe, na mesma indentação dos métodos.

```
class Aluno(Pessoa):
    qtd_disciplinas = 2

    # corpo da classe
```

Para acessar a variável da classe basta utilizar ou o próprio nome da classe antes da variável, tanto dentro da classe quanto fora dela. Exemplo:

```
# main.py
n = Aluno.qtd_disciplinas # não é necessário ter a instância de Aluno
print(n)
```

Dentro da própria classe o uso dos atributos estáticos também funciona. Altere o método `adicionaDisciplina()` da classe Aluno para deixar adicionar uma disciplina somente se o tamanho da lista for menor que a quantidade máxima de disciplinas que o aluno pode se matricular. Teste seu código. Ele funciona?

O programa não funciona porque `disciplinas` (atributo `__disciplinas` da superclasse Pessoa) é privado. Ou seja, mesmo a subclasse não tem acesso a este atributo. A solução neste caso é tirar os `__` do atributo `disciplinas`, assim poderá ser utilizado dentro das subclasses, ou então criar `__disciplinas` dentro da subclasse.

9. Padrões de Projeto

Neste capítulo vamos compreender o uso de padrões de projeto em variadas situações de desenvolvimento de projetos de software orientados a objetos. Vamos nos familiarizar com os tipos de padrões de projeto de estrutura, de

criação e de comportamento, e com a construção e utilização de padrões de projeto em Python.

- ✓ Compreender os padrões de projeto para desenvolvimento orientado a objetos;
- ✓ Familiarizar-se com tipos de padrões de estrutura, criação e comportamento;
- ✓ Construir e utilizar padrões em Python.

Ao desenvolver projetos de software nos deparamos com diversos problemas repetitivos. A experiência ajuda a aplicar soluções para esses problemas, e que podem ser aplicadas rotineiramente. Essas soluções, preferencialmente, devem ser desenvolvidas de forma elegante e prática!

Um padrão de projeto é uma descrição de um problema comum de projeto de software, uma solução, suas vantagens e desvantagens. Cada padrão possui um nome único, que facilita a discussão e memorização. O padrão não é um algoritmo ou implementação de uma biblioteca. Deve ser estudado no contexto de uma linguagem, e prover casos de uso conhecidos.

Ao utilizar padrões de projeto podemos reutilizar soluções comprovadamente eficientes para tratar diversos problemas. O livro mais popular que aborda esse assunto é o “Design Patterns: Elements of Reusable Object-Oriented Software” Descreve 23 padrões com exemplos em C++ e SmallTalk.

Podemos dividir os padrões de projetos em 3 categorias:

- Padrões de Estrutura:
 - Abordam modos de especificar a composição de classes, métodos e objetos;
- Padrões de Criação:
 - Abordam problemas comuns e soluções para instanciar objetos;
- Padrões de Comportamento:
 - Descrevem como classes e objetos interagem e dividem responsabilidades.

9.1. Padrões de Criação

Existem 2 importantes padrões de criação: Factory e Singleton.

9.1.1. Factory

Factory é um padrão usado para encapsular o processo de criação de objetos. Envolve a definição de uma classe que define uma interface abstrata para objetos de um tipo específico. Deixa a decisão de criação às subclasses. Isso é implementado por uma classe Factory, que cria e retorna os objetos.

Por exemplo, criamos uma classe Pessoa que possui métodos para definir o nome e sexo da pessoa. Depois, criamos duas classes filhas (Homem e Mulher) definem uma mensagem de boas vindas, de acordo com o sexo do objeto. A classe Factory é responsável por criar Homens e Mulheres. O código principal usa o Factory para criar novas pessoas.


```
class Pessoa:
    def __init__(self):
        self.nome = None
        self.sexo = None
    def getNome(self):
        return self.nome
    def getSexo(self):
        return self.sexo
```

```
class Homem(Pessoa):
    def __init__(self, nome):
        print("Olá Sr." + nome)
```

```
class Mulher(Pessoa):
    def __init__(self, nome):
        print("Olá Sra." + nome)
```

```
class PessoaFactory:
    def getPessoa(self, nome, sexo):
        if sexo == 'M':
            return Homem(nome)
        if sexo == 'F':
            return Mulher(nome)

if __name__ == '__main__':
    factory = PessoaFactory()
    pessoa = factory.getPessoa('Ana', 'F')
```

Factory é muito utilizado nos casos em que é necessária a associação extensível entre classe e classe criadora. A classe criadora (factory) se baseia em parâmetros para definir qual objetos criar. O código que utiliza o factory se torna escalável e de fácil manutenção. Para criar novos tipos, basta estender o código do factory

9.1.2. Singleton

O padrão singleton faz com que uma classe tenha apenas um único objeto ativo na aplicação. Pode ser usado quando é necessário restringir o acesso a recursos a um único contexto. Por exemplo, um conector de uma base de dados. Como lida com sincronização de dados e manipulação, o singleton certifica que nenhum outro objeto interage com o banco. . Existem algumas formas de fazer isso em Python.

Estratégia 1: Sobrescrever o método `__new__` da classe:

```
class Singleton:
    ...
    def __new__(cls):
        if not hasattr(cls, '_instance'):
            cls._instance = super().__new__(cls)
        return cls._instance
```

```
class MinhaClasse(Singleton):
    def __init__(self):
        self.tributo = 1

um = MinhaClasse()
dois = MinhaClasse()
dois.tributo = 3
print(um.tributo) # 3
```

Essa solução tem o problema quando é usada herança na classe Singleton. Todos os objetos serão instâncias da classe MinhaClasse.

```
class MinhaOutraClasse(MinhaClasse):
    ...
    b = 2

tres = MinhaOutraClasse()
print(tres.b) # Erro! Objeto tres não possui atributo b
```

Estratégia 2: Usar um módulo. Um Singleton não deve possuir vários níveis de herança, pois já é uma classe específica. Muitos desenvolvedores Python consideram esse padrão um modo muito complexo de lidar com unicidade em uma aplicação. Uma solução mais simples seria utilizar um módulo com funções como Singleton.

9.2. Padrões de Estrutura

Os padrões de estrutura principais são: Adapter, Facade e Decorator.

9.2.1. Adapter

Podemos desenvolver um código pensado para funcionar com uma classe A. Porém não há problema utilizar objetos de outra classe B, desde que B possa prover os métodos usados por A. Essa filosofia é chamada duck-typing (tipagem pato): “Se anda como um pato, e fala como um pato, então é um pato!”. O padrão Adapter é baseado nessa filosofia. É definido um “embrulho”, onde uma classe é adaptada para funcionar em outro contexto.

A classe StringIO é um bom exemplo. Ela adapta o tipo str para ser utilizado como um tipo file (arquivo). Recebe um string como entrada e oferece os mesmos métodos do tipo file.

```
from StringIO import StringIO
arq = StringIO(u'conteúdo string')
arq.read() # u'conteúdo string'
arq.seek(0)
arq.read(1) # u'c'
```

Esse padrão também muda o modo como desenvolvedores trabalham. Adaptar um objeto para funcionar em outro contexto faz com que não importe a classe de um objeto. O que realmente importa é que a classe implemente as informações que o código cliente espera chamar.

9.2.2. Facade

O padrão Facade provê uma interface simplificada para parte de um sistema. Normalmente é implementado ao prover um conjunto reduzido funções de alto nível de um sistema mais complexo. O Facade apenas chama as funções dos objetos originais. Os objetos originais permanecem disponíveis caso seja necessário chamar funções adicionais. Exemplo: Suponha que existam várias classes de teste em nossa aplicação T1, T2, ..., Tn. Podemos criar um Facade para executar todos os testes de uma só vez.

```
class T1:
    def run(self):
        print('Teste 1... ')
        ....
```

```
class T2:
    def run(self):
        print('Teste 2... ')
        ....
```

```
class T3:
    def run(self):
        print('Teste 3... ')
        ....
```

- **Facade**

```
class TestFacade:
    def __init__(self):
        self.T1 = T1()
        self.T2 = T2()
        self.T3 = T3()
    def run_all(self):
        self.T1.run()
        self.T2.run()
        self.T3.run()
```

- **Programa principal**

```
if __name__ == '__main__':
    testes = TestFacade()
    testes.run_all()
```

9.2.3. Decorator

O padrão Decorator torna possível alterar a funcionalidade de um método ou classe sem a necessidade alterar o código fonte da classe ou método decorado. Isso é possível ao criar um novo método, que “embrulha” o método decorado. Internamente o embrulho chama o método decorado, e possui outras funcionalidades para estender o seu comportamento. Chamadas ao método original são substituídas por chamadas ao embrulho.

Para usar um decorator, usamos a sintaxe @nome_do_decorator antes da definição do método. Um exemplo é o decorator property, que promove encapsulamento de métodos. O encapsulamento faz com que não seja possível fazer o acesso direto a um atributo. É necessário chamar um método que retorna o valor do atributo, ou altera o valor do atributo indiretamente. Dessa forma podemos controlar esse acesso.

Suponhamos que exista um atributo x em uma classe P, e queremos encapsular esse atributo. Não queremos que o valor x seja negativo e nem que ele tenha um valor maior que 1000. Como forçar que as restrições sejam respeitadas? Usando o decorator @property, definimos métodos que serão chamados quando a variável x for acessada.

```
class P:
    def __init__(self,x):
        self.x = x

    @property # decorator que controla o acesso a x
    def x(self):
        return self._x

    @x.setter # decorator que controle alterações a x
    def x(self, x):
        if x < 0:
            self._x = 0
        elif x > 1000:
            self._x = 1000
        else:
            self._x = x
```

Exemplos de acesso à variável x:

```
p = P(0)
print(p.x) # Imprime 0
p.x = 5000
print(p.x) # Imprime 1000
p.x = 200
print(p.x) # Imprime 200
p.x = -100
print(p.x) # Imprime 0
```

9.3. Padrões de Comportamento

Os principais padrões de comportamento são: Observer e Visitor.

9.3.1. Observer

O padrão Observer Faz com que um objeto (sujeito) mantenha uma lista de dependentes (observadores) e os notifique caso haja alguma alteração em seu estado. A notificação é feita ao chamar um dos métodos de todos os observadores. Essa padrão é bastante usado para implementar gerenciadores de eventos (ex. interfaces gráficas). Por exemplo, Usando o padrão observer em uma Mensagem (sujeito) para atualizar janelas (observadores) em um sistema gráfico.

```
class Sujeito:
    def __init__(self):
        pass
    def register(self):
        pass
    def notify_all(self):
        pass
```

```
class Observer:
    def __init__(self):
        pass
    def notify(self):
        pass
```

```
class Mensagem(Sujeito):  
    def __init__(self, msg):  
        self._observers = []  
        self.msg = msg  
  
    def register(self, observer): # adiciona um observador  
        if observer not in self._observers:  
            self._observers.append(observer)  
  
    def notify_all(self): # notifica os observadores  
        for observer in self._observers:  
            observer.notify(self.msg)  
  
    def set_msg(self, new_msg): # atualiza o conteúdo da msg  
        self.msg = new_msg  
        notify_all()
```

```
class Janela(Observer):  
    def __init__(self):  
        pass  
    def notify(self, msg): # imprime uma nova mensagem  
        print(msg)  
  
if __name__ == '__main__':  
    msg = Mensagem('Mensagem de teste')  
    janela1 = Janela()  
    janela2 = Janela()  
    msg.register(janela1)  
    msg.register(janela2)  
    msg.set_msg('Nova mensagem!')
```

9.3.2. Visitor

O padrão Visitor permite representar uma operação a ser realizada em uma estrutura de dados/objeto. Em outras palavras, permite separar a estrutura de dados dos algoritmos que irão operar sobre a estrutura. Dessa forma é possível definir uma nova operação sobre um objeto sem alterar o código que usa a estrutura.

Cada visitante (Visitor) pode implementar um algoritmo que irá operar sobre a classe a ser visitada (Visitable). O Visitable armazena apenas a estrutura de dados, e possui um método accept, que irá escolher qual método do visitante será invocado. Dessa forma, o código que utiliza o Visitable não necessita ser alterado. Novas operações dependem apenas da criação de novos Visitors e adaptação do método accept. Por exemplo, um visitor Imprimir atua sobre os visitables Lista e Dicionário.

```

class Imprimir:
    def visit_lista(self, lista):
        print('Elementos da lista: ', lista)
    def visit_dic(self, dic):
        print('Valores do dicionario: ', dic.values())

```

```

class Lista(list):
    def accept(self, visitor):
        visitor.visit_lista(self)

```

```

class Dicionario(dict):
    def accept(self, visitor):
        visitor.visit_dic(self)

```

```

if __name__ == '__main__':
    lista = Lista([1,2,3,4,5])
    lista.accept(Imprimir())
    dic = Dicionario({1:'Um', 2:'Dois', 3:'Três'})
    dic.accept(Imprimir())

```

10. Arquivos

Arquivos são estruturas interessantes para armazenar e recuperar dados dos programas. Servem como estruturas de entrada e saída. Neste capítulo vamos estudar as principais formas de trabalhar com arquivos em Python.....

✓ Entender os conceitos básicos de processamento de arquivos e técnicas para leitura e escrita de arquivos em Python;

✓ Entender e escrever programas que processam informação textual.

10.1. Objetivos

Os objetivos deste capítulo são:

- Entender os conceitos básicos de processamento de arquivos e técnicas para leitura e escrita de arquivos em Python
- Entender e escrever programas que processam informação textual

10.1.1. Conceitos

- Arquivos são estruturas interessantes para armazenar e recuperar dados dos programas

- Servem como estruturas de entrada e saída

Abrir um arquivo em Python é bastante simples, basta utilizar a função `open()` da biblioteca padrão do Python, enviando como parâmetro o caminho do arquivo:

`open(caminho_arquivo)`

O parâmetro `caminho_arquivo` é uma string que pode ser o caminho absoluto ou relativo, finalizando com o nome do arquivo:

`caminho_ate_arquivo/arquivo.txt`

Um caminho absoluto inclui todo o caminho começando da raiz do sistema de arquivos. Se for Windows, começa com a letra do drive. Por exemplo:

`C:\caminho_ate_arquivo\arquivo.txt`

Se for Linux ou Mac, começa com `/`. Por exemplo:

`/caminho_ate_arquivo/arquivo.txt`

Um caminho relativo inclui apenas o nome do arquivo, ou parte do caminho, iniciando na pasta corrente onde o programa está sendo executado. Exemplos:

`arquivo.txt`

`subpasta/arquivo.txt`

SOs diferentes utilizam barras diferentes para separar as pastas e arquivos:

- `\` para Windows
- `/` para Linux e Mac
-

Entretanto, Python identifica a barra comum (`/`) para todos os SO, mesmo no Windows. Portanto, para ler um arquivo no Windows, utilize:

`C:/caminho_ate_arquivo/arquivo.txt`

A função `open()` retorna um objeto do arquivo, também conhecido como um objeto de stream. Com este objeto, é possível fazer operações no arquivo.

`arquivo = open(caminho_arquivo)`

Para ler o arquivo inteiro de uma vez, utilize o método `read()` do objeto do arquivo.

`conteudo = arquivo.read()`

Faça o teste: escreva um programa em Python para ler um arquivo e mostrar o conteúdo na tela. Utilize o arquivo `tweets.txt` disponibilizado


```
# coding=utf-8
# abre o arquivo
arquivo = open('tweets.txt')
# lê todo o conteúdo
conteudo = arquivo.read()
print(conteudo)
```

Obs: a primeira linha de comentário é necessária para identificar os caracteres em UTF-8

Ao ler um arquivo, Python guarda a posição atual que está sendo lida.

Ao utilizar o método `read()`, que lê o arquivo inteiro, a posição será o final de arquivo. Se tentar utilizar o método `read()` novamente, o retorno será uma string vazia.

```
# coding=utf-8
arquivo = open('tweets.txt')
conteudo = arquivo.read()
print(conteudo)
# tenta ler novamente o conteúdo
conteudo = arquivo.read()
print(conteudo)
```

Para mudar a posição atual do arquivo, basta utilizar o método `seek()`, enviando o deslocamento de bytes como parâmetro. Para voltar ao começo do arquivo utilize `seek(0)`.

```
# coding=utf-8
arquivo = open('tweets.txt')
conteudo = arquivo.read()
print(conteudo)
# volta para o começo do arquivo
arquivo.seek(0)
conteudo = arquivo.read()
print(conteudo)
```

Para verificar qual a posição atual do arquivo, utilize o método `tell()`.

```
# coding=utf-8
arquivo = open('tweets.txt')
posicao = arquivo.tell()
print(posicao) # mostra posição após abrir
conteudo = arquivo.read()
posicao = arquivo.tell()
print(posicao) # mostra posição após ler arquivo todo

arquivo.seek(0)
conteudo = arquivo.read()
print(conteudo)
```

O método `read()` aceita um argumento para definir quantos bytes deseja ler.

```
# coding=utf-8
arquivo = open('tweets.txt')
posicao = arquivo.tell()
print(posicao) # mostra posição após abrir
conteudo = arquivo.read(10) # le 10 bytes
print(conteudo)
posicao = arquivo.tell()
print(posicao) # mostra posição após ler 10 bytes
```

Na maior parte das vezes um caractere representa um byte. Entretanto, para arquivos UTF-8 ou UTF-16, um caractere pode ocupar mais de um byte. Portanto, cuidado ao utilizar `read()` e `seek()` para arquivos codificados em UTF-8 ou UTF-16.

É uma boa prática fechar o arquivo quando não for mais utilizar. Se abrir muitos arquivos no seu programa, Python pode lançar um erro "too many open files". Para fechar o arquivo, basta utilizar o método `close()` do objeto do arquivo.

```
arquivo = open('tweets.txt')
# executa operações
arquivo.close()
```

É possível verificar se o arquivo está fechado a partir do atributo `closed`.

```
arquivo = open('tweets.txt')
print('arquivo fechado?', arquivo.closed)

if not arquivo.closed:
    arquivo.close()

print('arquivo fechado?', arquivo.closed)
```

É possível fechar um arquivo automaticamente utilizando a cláusula `with`, da seguinte forma:

```
with open('tweets.txt') as arquivo:  
    # bloco para executar operações no arquivo
```

Quando o bloco de código finalizar ou ocorrer algum erro na execução, o arquivo será fechado automaticamente. Para ler linha a linha de um arquivo, basta utilizar um loop for:

```
with open('tweets.txt') as arquivo:  
    for linha in arquivo:  
        print(linha)
```

Veja que a saída coloca uma linha em branco após cada linha. Isso ocorre porque o último caractere de cada linha do arquivo é uma quebra de linha (\n).

Para remover qualquer espaço em branco após o texto, incluindo uma quebra de linha utilize o método `rstrip()` de strings.

```
with open('tweets.txt') as arquivo:  
    for linha in arquivo:  
        print(linha.rstrip())
```

Sempre que abrir um arquivo é possível especificar o modo como ele será aberto, no segundo argumento da função `open()`.

`open(caminho_arquivo, modo)`

O padrão, sem o argumento, é abrir apenas para leitura. Para modificar o modo, utilize um caractere (string) no segundo argumento, conforme a tabela a seguir:

Modo	Descrição
r	Leitura (padrão)
w	Escrita, apagando o arquivo antes
x	Cria um novo arquivo e abre para leitura
A	Escrita, adicionando no final se ele existir
b	Modo binário
t	Modo texto (padrão)
+	Abre para atualização (leitura e escrita)

Para escrever em um arquivo é necessário:

- Abrir o arquivo em algum modo de escrita (w, a, x ou +);
- Chamar o método `write()`

```
# abre para escrita
with open('saida.txt', 'w') as arquivo:
    arquivo.write('Texto 1')
    arquivo.write('Texto 2')

# abre para leitura
with open('saida.txt') as arquivo:
    print(arquivo.read())
```

Repare que o método `write()` escreve exatamente o que foi pedido. No exemplo anterior, não foi definida nenhuma quebra de linha. Portanto, a segunda chamada de `write()` escreveu logo após a última letra do primeiro texto. Para ter uma quebra de linha é necessário escrever `\n` ou `\r` no arquivo. Para Windows pode-se utilizar também `\r\n` para identificar a nova linha

```
# abre para escrita
with open('saida.txt', 'w') as arquivo:
    arquivo.write('Texto 1\r\n')
    arquivo.write('Texto 2\r\n')

# abre para leitura
with open('saida.txt') as arquivo:
    print(arquivo.read())
```

11. Tratamento de Exceções

No capítulo de testes percebemos que os erros em programas podem acontecer, e que temos que testar bem o software. Agora, veremos como podemos tratar erros e exceções inesperadas que podem acontecer durante a execução de um programa, de modo que o programa não pare abruptamente.

- ✓ Entender os tipos de erros que podem acontecer em um programa;
- ✓ Indicar a ocorrência de um erro inesperado em Python;
- ✓ Entender o StackTrace do Python quando um erro acontece;
- ✓ Tratar exceções em Python (`try/ except/ finally`);
- ✓ Criar exceções customizadas;
- ✓ Lançar exceções;
- ✓ Apresentar alguns erros comuns;
- ✓ Trabalhar com assertões.

11.1. Tipos de Erros

Existem basicamente 4 tipos de erros que podem acontecer em um programa:

- Sintáticos
- Semânticos
- Execução (runtime)

- Erros definidos pelo usuário

Erros sintáticos são erros na estrutura do programa. Por exemplo, uma variável não iniciada, um bloco mal indentado, erros de grafia dos comandos Python. Se algum erro deste tipo ocorrer, o programa é rejeitado pelo interpretador.

Erros semânticos são um pouco diferentes. Eles não emitem uma mensagem de erro, mas o programa não faz o esperado. Este tipo de erro são encontrados quando realizamos testes unitários.

O erro de execução acontece durante a execução de um programa. A estrutura do código está completamente correta, mas algum problema ocorreu ao validar uma variável, acessar algum componente de alguma biblioteca, retornar uma posição inexistente de um dicionário ou lista. Normalmente este tipo de erro lança uma exceção.

Outro tipo de erro é o erro definido pelo usuário. São erros bem específicos que são tratados dentro de um programa. Por exemplo, o usuário colocou a senha errada para entrar em um sistema.

11.2. Tratamento de Erros no Programa

Um programa não pode simplesmente para quando está sendo executado, sem dar nenhuma informação para o usuário. Então, como indicar que um problema não previsto dentro do seu código ocorreu?

Vamos ver essa situação com um exemplo: Considere uma função que calcule a porcentagem, dado um valor e uma taxa (dentro 0 a 100).

```
def porcentagem(valor, taxa):  
    if (taxa > 100 or taxa < 0):  
        return -1  
    else:  
        return valor * taxa / 100
```

É uma boa ideia? Imagine o usuário utilizando seu programa e quer saber qual é o valor final de aplicar uma taxa de 130% sobre 3000. É -1? Isso deixaria o usuário confuso. Não fica claro como os erros são notificados e que tipos de erros podem acontecer.

porcentagem(3000,130)

Por exemplo se aplicarmos uma taxa de 1% sobre o valor -100, o resultado será -1. Isso é um erro ou o valor correto? Neste caso é o valor correto. Mas o que diferencia do erro? Nada!

porcentagem(-100,1)

Pense em situações mais complexas: o retorno de uma função pode ser uma String, ou até mesmo uma classe. Ficaria muito mais difícil de tratar. Existem também operações que podem causar vários problemas diferentes. Por exemplo, acessar um arquivo: ele pode não existir, pode estar bloqueado, posso ter o acesso negado, etc. Uma dica: nunca utilize variáveis globais para armazenar erros! Pode acontecer problemas de concorrência. O erro será referente a qual parte do programa? Você pode esquecer-se de limpar o erro, e outros componentes podem encontrar

erros anteriores.

11.3. Exceções

Exceções são tipos especiais de classes que representam erros que podem ser lançados para o caller (quem chamou o comando). É sempre interessante criar as próprias exceções. Fica mais fácil de rastrear de onde veio o erro, e que tipo de erro aconteceu.

Exceções são criadas em classes que estendem a classe `Exception` do Python. Ou seja, a sua exceção é uma subclasse de `Exception`, e recebe por herança toda a estrutura necessária para tratar uma exceção. Para criar uma classe de exceção chamada `MinhaExcecao`, basta fazer o seguinte:

```
class MinhaExcecao(Exception):  
    pass
```

Essa classe já pode ser utilizada para lançar exceções no seu programa. Por exemplo, na função `porcentagem` que fizemos anteriormente, se a taxa não estiver entre 0 e 100, lançamos uma exceção com `raise`. O construtor `MinhaExcecao` recebe uma string, que será a mensagem associada à exceção.

```
def porcentagem(valor, taxa):  
    if (taxa > 100 or taxa < 0):  
        raise MinhaExcecao("Valor inválido")  
    else:  
        return valor * taxa / 100
```

A chamada da função deve estar cercada de um bloco `try/except`. Ou seja, o interpretador tenta executar a função (`try`). Caso uma exceção seja lançada pela função (`raise`), o bloco `except` é executado.

```
try:  
    porcentagem(3000, 130)  
except MinhaExcecao:  
    // tratar a excecao  
    print "Erro: "
```

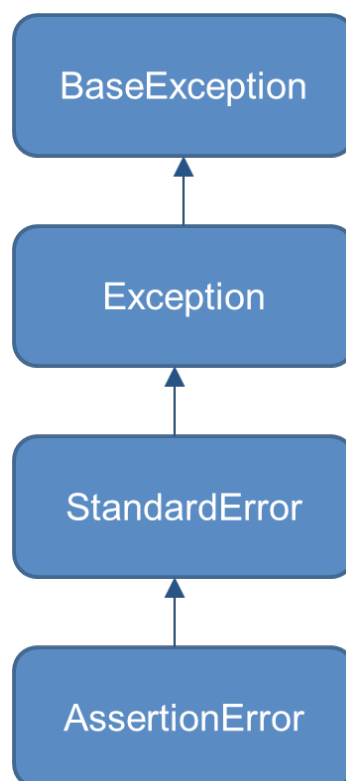
O objeto da exceção capturada pode ser colocada em uma variável, logo após o nome da classe. Assim conseguiremos obter mais detalhes das exceções.

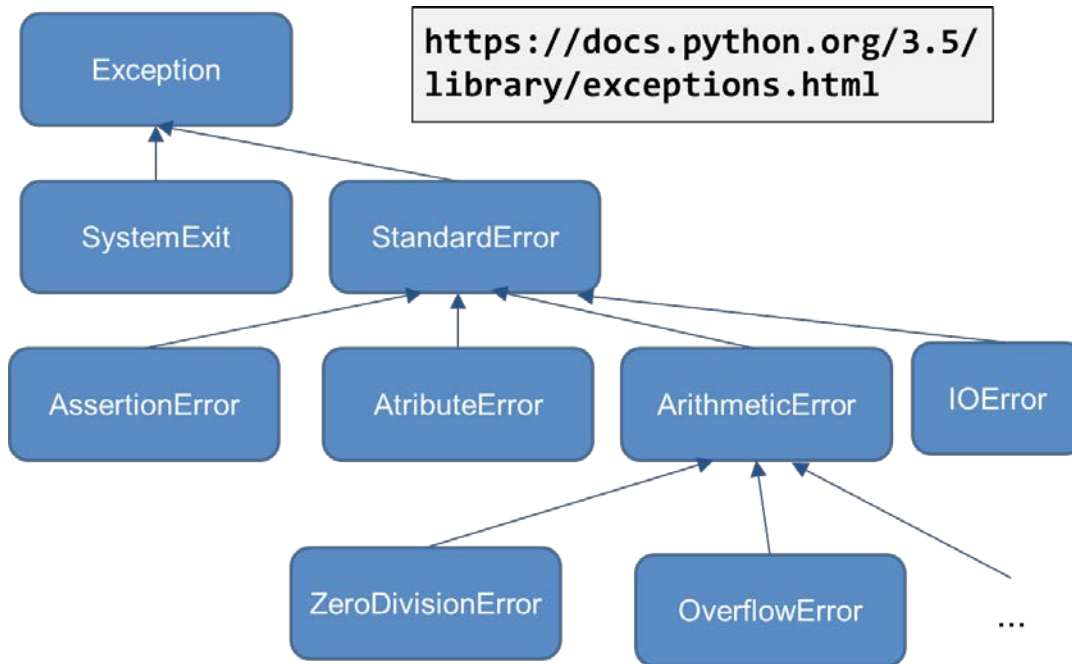
```
try:  
    porcentagem(3000, 130)  
except MinhaExcecao, e:  
    // tratar a excecao  
    print "Erro: ", e
```

```
try:  
    porcentagem(3000, 130)  
except MinhaExcecao as e:  
    // tratar a excecao  
    print "Erro: ", e
```

11.4. Entendendo as Exceções

Todos os erros e exceções em Python são instâncias da classe `BaseException`. Isto permite organizá-los em hierarquia.



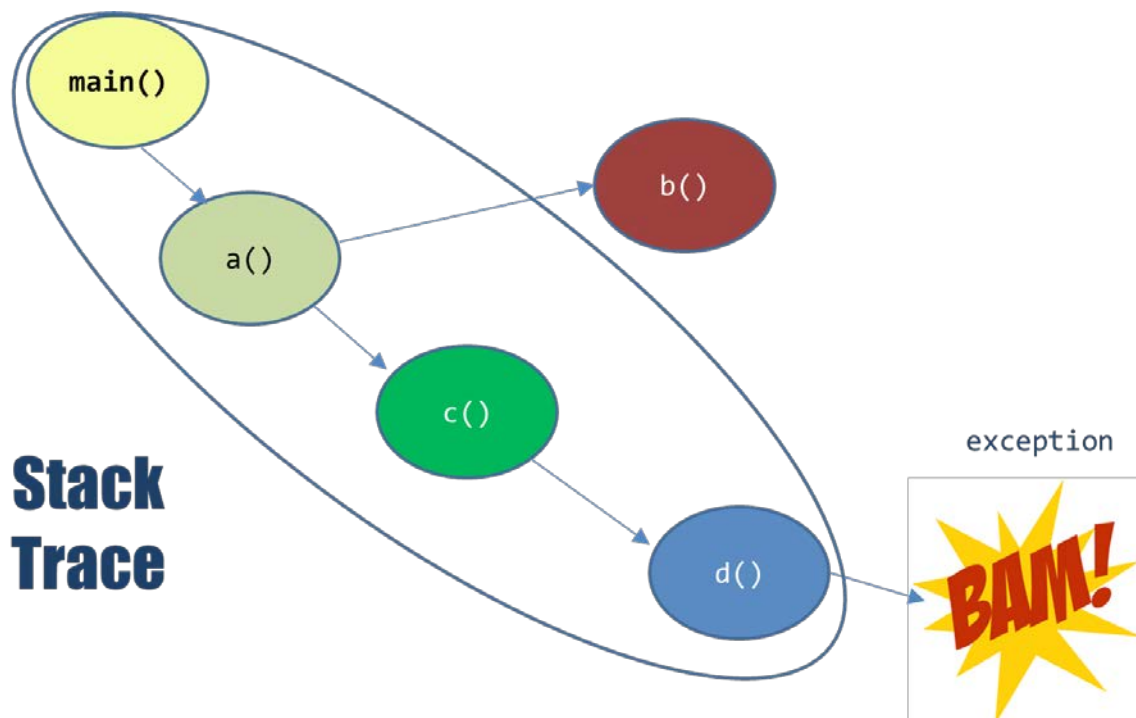
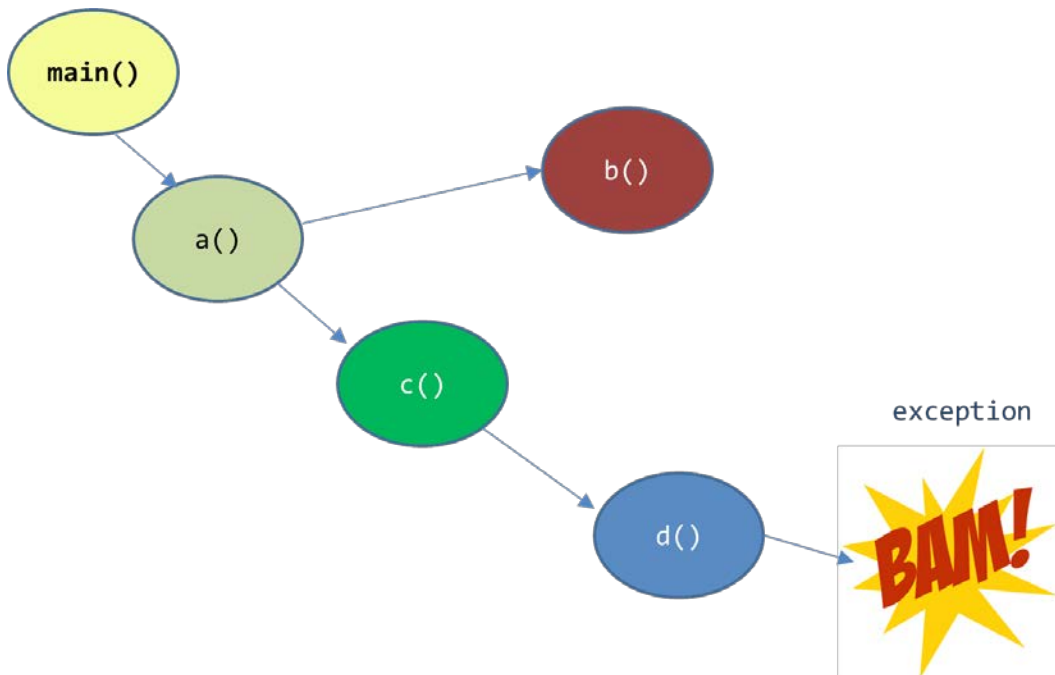


Lista de exceções do Python (built in)

Classe	Descrição
Exception	Classe base para todas as exceções
AttributeError	Falha no acesso ou atribuição a atributo de classe
IOError	Falha no acesso a arquivo inexistente ou outros de E/S
IndexError	Índice inexistente de sequência
KeyError	Chave inexistente de dicionário
NameError	Variável inexistente
SyntaxError	Erro de sintaxe (código errado)
TypeError	Operador embutido aplicado a objeto de tipo errado
ValueError	Operador embutido aplicado a objeto de tipo certo mas valor inapropriado
ZeroDivisionError	Divisão ou módulo por zero

StackTrace

Quando uma Exception é criada, registra-se toda a chamada da pilha a partir daquele ponto.



Como ler a stacktrace?

```
1      class MinhaExcecao(Exception): pass
2
3      def funcao(x):
4          if (x < 0 or x > 100):
5              raise MinhaExcecao("Valor incorreto")
6          else:
7              return x
8
9      def main():
10         funcao(1000)
11
12     if __name__ == '__main__':
13         main()
```


Execução do código acima:

```
$ python exemplo_excecao.py
Traceback (most recent call last):
  File "exemplo_excecao.py", line 22, in <module>
    main()
  File "exemplo_excecao.py", line 16, in main
    funcao(1000)
  File "exemplo_excecao.py", line 8, in funcao
    raise MinhaExcecao("Valor incorreto")
__main__.MinhaExcecao: Valor incorreto
```

11.5. Tratando exceções

Quando a exceção ocorre, a execução pula para o bloco except e continua a partir daí.

```
try :
    funcao();
    outraFuncao();
    ...
except MinhaExcecao, e:
    tratadorErro(e)
depoisTry();
```

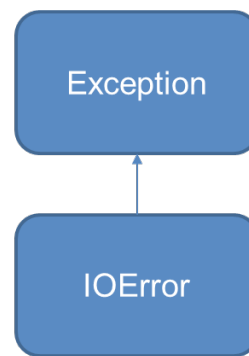


The diagram illustrates the flow of execution in a try-except block. A blue line starts from the right side of the try block, goes down, and then left, ending with an arrow pointing to the except block. Another blue line starts from the right side of the except block, goes down, and then left, ending with an arrow pointing to the final line of code, 'depoisTry();'.

É possível capturar diferentes exceções no mesmo bloco try/except. O importante é que as exceções mais específicas (subclasses) devem vir antes.

```
try :
    funcao();
    outraFuncao();
    ...
except IOError as err:
    print(err)
except Exception as e:
    tratadorErro(e)

depoisTry();
```



Subclasse vem antes!

Exemplo prático: a função `math.sqrt()` gera `ValueError` com um argumento negativo.

```
from math import sqrt

sqrt(-1) # ValueError
```

O tratamento do erro para o `math.sqrt()` seria:

```
from math import sqrt

try:
    num = int(input(Digite um número:))
    resultado = sqrt(num)
    print "A raiz quadrada: ", resultado
except ValueError:
    print "Não é possível tirar a raiz quadrada de ", num
```

O bloco de tratamento de exceção pode ter uma cláusula `else`, que será executado caso não ocorra nenhuma exceção.

```
def divide(x,y):
    try:
        resultado = x / y
    except ZeroDivisionError:
        print "Não é possível divisão por zero."
    else:
        print "resultado: ", resultado
```

Exemplo prático: utilizando a cláusula `else` para a leitura de arquivos:

```
# -*- coding: utf-8 -*-
import sys
nome_arq = sys.argv[1]
texto = []

try:
    arq = open(nome_arq, 'r')
except IOError:
    print "Não foi possível ler o arquivo: ", nome_arq
else:
    texto = arq.readlines()
    arq.close()

if texto:
    print(texto[0:3])
```

A cláusula finally também pode fazer parte do bloco, e sempre será executada independente da ocorrência da exceção:

```
def divide(x,y):
    try:
        resultado = x / y
    except ZeroDivisionError:
        print "Não é possível divisão por zero."
    else:
        print "resultado: ", resultado
    finally:
        print "Executando a cláusula finally"
```

O finally será executado mesmo que exista um return dentro do try. Ou seja, antes de realmente retornar para a chamada da função, o finally será executado.

```
def divide(x,y):
    try:
        resultado = x / y
        return resultado
    except ZeroDivisionError:
        print "Não é possível divisão por zero."
    finally:
        print "Executando a cláusula finally"
```

Uma das aplicações da cláusula finally é ser usada para ações de limpeza, como fechar um arquivo ou conexão com banco de dados.

```
# -*- coding: utf-8 -*-
import sys
nome_arq = sys.argv[1]
texto = []

try:
    arq = open(nome_arq, 'r')
except IOError:
    print "Não foi possível ler o arquivo: ", nome_arq
else:
    texto = arq.readlines()
finally:
    arq.close()

if texto:
    print(texto[0:3])
```

Bibliografia

BECK, K. TDD – Desenvolvimento Guiado por Testes. Porto Alegre: Bookman, 2010.

Cannon J. Python Succinctly [Internet]. Syncfusion; [citado 10 de fevereiro de 2016]. 146 p. Disponível em: <https://www.syncfusion.com/resources/techportal/details/ebooks/python>

DIERBACH, C. Introduction to Computer Science Using Python: A Computational Problem-Solving Focus 1st Edition, New York: Wiley, 2012.

GIRIDHAR C. Aprendendo Padrões de Projeto em Python. São Paulo: Novatec, 2016.

GUTTAG, J.V. Introduction to Computation and Programming Using Python: With Application to Understanding Data. 2nd ed. Cambridge MA: The MIT Press, 2016.

LUTZ, M. Programming Python: Powerful Object-Oriented Programming. 4th ed. Sebastopol CA: O'Reilly, 2011.

RAVINDRAN, A. Django Design Patterns and Best Practices Packt Publishing Ltd, 2015.

RAMALHO, L. Python Fluente. São Paulo: Novatec, 2015

ZELLE J. Python Programming: An Introduction to Computer Science, 2nd Ed. 2 edition. Sherwood, Or: Franklin, Beedle & Associates Inc.; 2010.