Python e Django

Ramiro B. da Luz

A RNP – Rede Nacional de Ensino e Pesquisa – é qualificada como uma Organização Social (OS), sendo ligada ao Ministério da Ciência, Tecnologia e Inovação (MCTI) e responsável pelo Programa Interministerial RNP, que conta com a participação dos ministérios da Educação (MEC), da Saúde (MS) e da Cultura (MinC). Pioneira no acesso à Internet no Brasil, a RNP planeja e mantém a rede Ipê, a rede óptica nacional acadêmica de alto desempenho. Com Pontos de Presença nas 27 unidades da federação, a rede tem mais de 800 instituições conectadas. São aproximadamente 3,5 milhões de usuários usufruindo de uma infraestrutura de redes avançadas para comunicação, computação e experimentação, que contribui para a integração entre o sistema de Ciência e Tecnologia, Educação Superior, Saúde e Cultura.

RNP

MINISTÉRIO DA **DEFESA**

MINISTÉRIO DA **CULTURA**

MINISTÉRIO DA **SAÚDE**

MINISTÉRIO DA **EDUCAÇÃO**

MINISTÉRIO DA **CIÊNCIA, TECNOLOGIA, INOVAÇÕES E COMUNICAÇÕES**

BRASIL
GOVERNO FEDERAL

Python e Django

Ramiro B. da Luz

Python e Django

Ramiro B. da Luz

Rio de Janeiro
Escola Superior de Redes
2017

Rua Lauro Müller, 116 sala 1103
22290-906 Rio de Janeiro, RJ

Diretor Geral
Nelson Simões

Diretor de Serviços e Soluções
José Luiz Ribeiro Filho

Escola Superior de Redes

Diretor Adjunto
Leandro Marcos de Oliveira Guimarães

Edição
Lincoln da Mata

Coordenador Acadêmico da Área de Desenvolvimento de Sistemas
John Lemos Forman

Equipe ESR (em ordem alfabética)
Adriana Pierro, Alynne Pereira, Camila Gomes, Celia Maciel, Edson Kowask, Elimária Barbosa, Evellyn Feitosa, Felipe Nascimento, Lourdes Soncin, Luciana Batista, Luiz Carlos Lobato, Márcia Correa, Márcia Helena Rodrigues, Monique Souza, Renato Duarte, Thays Farias e Yve Marcial.

Capa, projeto visual e diagramação
Tecnodesign

Versão
1.0.0

Distribuição
Escola Superior de Redes
Rua Lauro Müller, 116 – sala 1103
22290-906 Rio de Janeiro, RJ
http://esr.rnp.br
info@esr.rnp.br

Dados Internacionais de Catalogação na Publicação (CIP)

C885m da Luz, Ramiro
Python e Django / Ramiro B. da Luz – Rio de Janeiro:
RNP/ESR, 2016.
224p.: il.; 28 cm.

ISBN 978-85-63630-02-5

1.Gerenciamento de projetos de tecnologia da informação. 2.Gerenciamento de projetos –Tecnologia da informação – planejamento, gestão e controle. 3.PMBok.I. Título.

CDD 658.4

Sumário

Escola Superior de Redes

A Escola Superior de Redes (ESR) é a unidade da Rede Nacional de Ensino e Pesquisa (RNP) responsável pela disseminação do conhecimento em Tecnologias da Informação e Comunicação (TIC). A ESR nasce com a proposta de ser a formadora e disseminadora de competências em TIC para o corpo técnico-administrativo das universidades federais, escolas técnicas e unidades federais de pesquisa. Sua missão fundamental é realizar a capacitação técnica do corpo funcional das organizações usuárias da RNP, para o exercício de competências aplicáveis ao uso eficaz e eficiente das TIC.

A ESR oferece dezenas de cursos distribuídos nas áreas temáticas: Administração e Projeto de Redes, Administração de Sistemas, Segurança, Mídias de Suporte à Colaboração Digital e Governança de TI.

A ESR também participa de diversos projetos de interesse público, como a elaboração e execução de planos de capacitação para formação de multiplicadores para projetos educacionais como: formação no uso da conferência web para a Universidade Aberta do Brasil (UAB), formação do suporte técnico de laboratórios do Proinfo e criação de um conjunto de cartilhas sobre redes sem fio para o programa Um Computador por Aluno (UCA).

A metodologia da ESR

A filosofia pedagógica e a metodologia que orientam os cursos da ESR são baseadas na aprendizagem como construção do conhecimento por meio da resolução de problemas típicos da realidade do profissional em formação. Os resultados obtidos nos cursos de natureza teórico-prática são otimizados, pois o instrutor, auxiliado pelo material didático, atua não apenas como expositor de conceitos e informações, mas principalmente como orientador do aluno na execução de atividades contextualizadas nas situações do cotidiano profissional.

A aprendizagem é entendida como a resposta do aluno ao desafio de situações-problema semelhantes às encontradas na prática profissional, que são superadas por meio de análise, síntese, julgamento, pensamento crítico e construção de hipóteses para a resolução do problema, em abordagem orientada ao desenvolvimento de competências.

Dessa forma, o instrutor tem participação ativa e dialógica como orientador do aluno para as atividades em laboratório. Até mesmo a apresentação da teoria no início da sessão de aprendizagem não é considerada uma simples exposição de conceitos e informações. O instrutor busca incentivar a participação dos alunos continuamente.

As sessões de aprendizagem onde se dão a apresentação dos conteúdos e a realização das atividades práticas têm formato presencial e essencialmente prático, utilizando técnicas de estudo dirigido individual, trabalho em equipe e práticas orientadas para o contexto de atuação do futuro especialista que se pretende formar.

As sessões de aprendizagem desenvolvem-se em três etapas, com predominância de tempo para as atividades práticas, conforme descrição a seguir:

Primeira etapa: apresentação da teoria e esclarecimento de dúvidas (de 60 a 90 minutos).
O instrutor apresenta, de maneira sintética, os conceitos teóricos correspondentes ao tema da sessão de aprendizagem, com auxílio de slides em formato PowerPoint. O instrutor levanta questões sobre o conteúdo dos slides em vez de apenas apresentá-los, convidando a turma à reflexão e participação. Isso evita que as apresentações sejam monótonas e que o aluno se coloque em posição de passividade, o que reduziria a aprendizagem.

Segunda etapa: atividades práticas de aprendizagem (de 120 a 150 minutos).
Esta etapa é a essência dos cursos da ESR. A maioria das atividades dos cursos é assíncrona e realizada em duplas de alunos, que acompanham o ritmo do roteiro de atividades proposto no livro de apoio. Instrutor e monitor circulam entre as duplas para solucionar dúvidas e oferecer explicações complementares.

Terceira etapa: discussão das atividades realizadas (30 minutos).
O instrutor comenta cada atividade, apresentando uma das soluções possíveis para resolvê-la, devendo ater-se àquelas que geram maior dificuldade e polêmica. Os alunos são convidados a comentar as soluções encontradas e o instrutor retoma tópicos que tenham gerado dúvidas, estimulando a participação dos alunos. O instrutor sempre estimula os alunos a encontrarem soluções alternativas às sugeridas por ele e pelos colegas e, caso existam, a comentá-las.

Sobre o curso

Este é um curso de introdução à linguagem de programação Python e ao framework Django.

São apresentados os tipos de dados utilizados pela linguagem assim como os comandos de controle do fluxo de execução de programas. Conceitos relacionados com orientação a objetos são explorados, oferecendo um conjunto sólido de conhecimentos que são a base indispensável para começar a desenvolver sistemas em Python.

Em seguida é apresentado o Django, que permite acelerar o ciclo de desenvolvimento de aplicações e sistemas em Python, passando pela criação de um projeto, configuração inicial, implementação do models.py, a camada de modelo, até a utilização da API de banco de dados. São apresentadas técnicas para melhorar o aspecto visual das aplicações, utilizando arquivos de layout, imagens e estilos, além do uso das poderosas views genéricas, que tornam o desenvolvimento ainda mais produtivo através, entre outros, do uso de testes automatizados.

Ao final do curso são apresentados os conceitos de configuração de servidor e depuração de aplicações Django, cobrindo a etapa de colocação em produção e manutenção de aplicações escritas em Python/Django.

Cada sessão apresenta um conjunto de exemplos e atividades práticas que permitem a prática dos conceitos e funcionalidades apresentadas.

A quem se destina

Pessoas interessadas no desenvolvimento de software/sistemas utilizando Python como linguagem de programação. Indicado para quem quer começar a dominar a programação de computadores usando Python como porta de entrada para este universo.

Convenções utilizadas neste livro

As seguintes convenções tipográficas são usadas neste livro:

Itálico
Indica nomes de arquivos e referências bibliográficas relacionadas ao longo do texto.

```
Largura constante
```

Indica comandos e suas opções, variáveis e atributos, conteúdo de arquivos e resultado da saída de comandos.

Conteúdo de slide 
Indica o conteúdo dos slides referentes ao curso apresentados em sala de aula.

Símbolo 
Indica referência complementar disponível em site ou página na internet.

Símbolo 
Indica um documento como referência complementar.

Símbolo 
Indica um vídeo como referência complementar.

Símbolo 
Indica um arquivo de aúdio como referência complementar.

Símbolo 
Indica um aviso ou precaução a ser considerada.

Símbolo 
Indica questionamentos que estimulam a reflexão ou apresenta conteúdo de apoio ao entendimento do tema em questão.

Símbolo 
Indica notas e informações complementares como dicas, sugestões de leitura adicional ou mesmo uma observação.

Permissões de uso

Comentários e perguntas

Para enviar comentários e perguntas sobre esta publicação:
Escola Superior de Redes RNP
Endereço: Av. Lauro Müller 116 sala 1103 – Botafogo
Rio de Janeiro – RJ – 22290-906
E-mail: info@esr.rnp.br

Sobre os autores

Ramiro B. da Luz é Bacharel em informática. Iniciou a carreira de desenvolvimento de sistemas em 1991. Realizou mestrado em 2013 pela Universidade Tecnológica Federal do Paraná, onde foi pesquisador da área de Engenharia de Software com ênfase em métodos ágeis do Programa de Pós-Graduação em Computação Aplicada. Programador concursado da Câmara Municipal de Curitiba. Fundador do grupo de usuários de Python do Paraná e um dos fundadores do grupo dojo Paraná. Anfitrião da PythonBrasil[6] em 2010, membro da Associação Python Brasil.

John Lemos Forman é Mestre em Informática (ênfase em Engenharia de Software) e Engenheiro de Computação pela PUC-Rio, com pós-graduação em Gestão de Empresas pela COPPEAD/UFRJ. É vice-presidente do Sindicato das Empresas de Informática do Rio de Janeiro – TI RIO, membro do Conselho Consultivo e de normas Éticas da Assespro-RJ e Presidente do Conselho Deliberativo da Riosoft. É sócio e Diretor da J.Forman Consultoria e atua como especialista em desenvolvimento de sistemas para a Escola Superior de Redes da RNP. Acumula mais de 30 anos de experiência na gestão de empresas e projetos inovadores, com destaque para o uso das Tecnologias da Informação e Comunicação na Educação, Saúde e Gestão Pública e Privada. Mais recentemente vem desenvolvendo projetos que fazem uso de Ciência de Dados e Aprendizagem de Máquina (Machine Learning).

1

Introdução e tipos básicos de dados

objetivos

Começar a apresentar o universo da linguagem Python através de exemplos simples. Ao fim desta sessão, o aluno terá conhecido os tipos básicos de dados utilizados pela linguagem e como executar programas escritos em Python.

conceitos

História e características da linguagem; PEP; Tipos de dados numéricos; Strings e formatação; docstrings, doctests e o tipo data.

Histórico

- Inspirada na linguagem ABC (interpretada, indentada, tipos de dados de altíssimo nível).
- ABC não permitia extensões.
- Contato com os projetistas da linguagem Modula-3 originou sintaxe e semântica de extensões, além de algumas funcionalidades da linguagem Python.
- Sistema Operacional Amoeba precisava de algo melhor que C e Bourne shell scripts, devido à interface de chamadas de sistemas pouco acessível a Bourne shell.
- Surgiu então a necessidade de uma linguagem de script semelhante à ABC que permitisse extensões genéricas.
- O pontapé inicial foi dado durante os feriados de Natal de 1989.
- Desenvolvida durante o ano seguinte no tempo disponível, foi usada no projeto Amoeba com sucesso crescente.
- Em fevereiro de 1991, depois de pouco mais de um ano de desenvolvimento, foi publicada na USENET.
- Em 2008 foi lançada a versão 3. Reescrita da linguagem sem muita preocupação em manter compatibilidade com versões anteriores. Melhorias na biblioteca padrão devem ocorrer apenas nessa versão.

Por volta do ano de 1989, o autor da linguagem Python, Guido Van Rossum, trabalhava no CWI com o desenvolvimento da linguagem ABC. A linguagem ABC é uma linguagem interpretada, com blocos de código são delimitados por indentação e que possui tipos de dados de altíssimo nível. Entretanto, essa linguagem não permitia extensões.

Essa experiência com a linguagem ABC e o contato com projetistas da linguagem Modula-3 influenciaram a origem da sintaxe e semântica de extensões, além de algumas

funcionalidades da linguagem Python. Guido trabalhava com um sistema distribuído chamado AMOEBA. A equipe dele precisava de uma maneira um pouco melhor para administrar o sistema que os programas escritos em C e scripts Bourne shell, entretanto, a linguagem ABC não tinha acesso à interface de chamadas de sistema do AMOEBA. Da experiência que ele possuía com ABC, sabia da importância que o tratamento de exceções tinha como característica de uma linguagem de programação.

Então, ele sentiu a necessidade de uma linguagem com uma sintaxe similar à ABC, com acesso à interface de chamadas de sistema do AMOEBA e que fosse extensível. Surgia então a inspiração para criar uma linguagem nova. No Natal de 1989, Guido começa o desenvolvimento da linguagem Python em seu tempo disponível, sendo desenvolvida durante o ano seguinte, e progressivamente usada no projeto AMOEBA, tendo sucesso crescente. Muitas melhorias foram acrescentadas após sugestões de seus colegas.

Após pouco mais de um ano de desenvolvimento, em 1991, a linguagem Python foi postada na USENET. Em 2008 foi lançada a versão 3. Essa versão é a primeira reescrita da linguagem sem muita preocupação em manter compatibilidade com versões anteriores, possibilitando uma limpeza e mudanças sem passar pelo tradicional processo de depreciação da versão 2. Melhorias na biblioteca padrão devem ocorrer apenas na versão 3. Por esse motivo, esse curso será baseado na versão 3.

O desenvolvimento da linguagem Python começou no Natal de 1989.

Características do Python

- Interpretada, orientada a objetos, altíssimo nível, multiplataforma, interativa.
- Esquema de numeração de versões: X.Y.Z
- Licença open source, algumas versões são compatíveis com a GPL.
- Python pode usar bibliotecas nativas em C.
- Os tipos nativos de Python são objetos.
- Linguagem de propósito geral, incorporando uma variedade considerável de bibliotecas.
- Python é utilizado pela Disney, Philips, Rackerspace, Globo.com e o Governo Federal.
- O nome veio de uma série de TV dos anos 70, “Monty Python’s Flying Circus”.

A linguagem Python é interpretada, as instruções são processadas e um código intermediário, conhecido como bytecode, é gerado e depois executado por uma máquina virtual. Outra característica da linguagem é ser orientada a objetos, permitindo a criação de programas considerados estruturados e possuir comandos para programação funcional, sendo considerada uma linguagem multiparadigma.

É uma linguagem de altíssimo nível, o que significa dizer que a linguagem possui nível de abstração próximo à linguagem humana. Diferente, portanto, de linguagens de baixo nível com instruções próximas a da máquina, e por isso mesmo conhecidas como linguagens de máquina. Python é uma linguagem multiplataforma e pode ser executada em diversos Sistemas Operacionais. Uma das características mais interessantes é que a linguagem é interativa, permitindo através do seu interpretador ir executando trechos de código e obter os resultados de imediato.

Python possui licença open source 7 e algumas versões são compatíveis com a GPL. Na tabela é apresentado o histórico de versões, respectivas licenças, proprietários e compatibilidade ou não com a GPL.

Release	Derivado de	Ano	Proprietário	Compatível c/ GPL?
0.9.0 a 1.2	n/a	1991-1995	CWI	sim
1.3 a 1.5.2	1.2	1995-1999	CNRI	sim
1.6	1.5.2	2000	CNRI	não
2.0	1.6	2000	BeOpen.com	não
1.6.1	1.6	2001	CNRI	não
2.1	2.0+1.6.1	2001	PSF	não
2.0.1	2.0+1.6.1	2001	PSF	sim
2.1.1	2.1+2.0.1	2001	PSF	sim
2.1.2	2.1.1	2002	PSF	sim
2.1.3	2.1.2	2002	PSF	sim
2.2 e superior	2.1.1	2001-now	PSF	sim

Tabela 1.1 Sumário de versões e licença da linguagem Python.

Olhando a tabela, podemos perceber um padrão de numeração das versões (releases) da linguagem Python, seguindo a estrutura A.B ou A.B.C. Os grandes lançamentos com mudanças significativas são representados pela letra A, que varia muito esporadicamente. Versões menores são representadas pela letra B, sendo considerados lançamentos intermediários. A letra C representa o incremento de menor nível, sendo utilizado sempre que é disponibilizada uma versão com correção de erros.

A linguagem Python possui diversas implementações. A IronPython é a implementação da linguagem para a plataforma.NET, Jython é a implementação para plataforma Java, e PyPy é uma implementação da linguagem escrita em Python. A implementação mais comum e mais usada é a Cpython, escrita na linguagem C, e que por isso mesmo permite utilizar bibliotecas nativas em C.

Em Python, os tipos nativos de dados são objetos. Os principais tipos são:

- Numéricos.
- Sequências.
- Mapeamentos.
- Classes.
- Instâncias.
- Exceções.

Algumas operações são suportadas por diversos tipos de objetos. Praticamente todos os objetos podem ser comparados e testados pelo valor verdade (o que significa ser avaliado como se fosse um booleano : Verdadeiro/Falso). Praticamente todos os objetos podem ser convertidos em strings através das funções repr e str. A função str é usada implicitamente quando um objeto é escrito usando a função print.

Como Python é uma linguagem de propósito geral, ela incorpora uma variedade considerável de bibliotecas consideradas padrão. Como exemplo, podemos citar bibliotecas para processamento de strings, protocolos de internet, engenharia de software e interfaces para Sistema Operacional. Python é usada para o desenvolvimento de jogos, sistemas para internet, animações, administração de sistemas, segurança e computação científica.

Python é uma linguagem que pode ser obtida muito facilmente e está disponível na maioria das distribuições GNU/Linux. Entre os usuários mais famosos da linguagem Python podemos citar: Industrial Light and Magic, Disney Animation, a fabricante de semicondutores da Philips em Nova York e a D-Link Austrália. A Frequentis utiliza Python em seus produtos de controle de tráfego aéreo, enquanto que a United Space Alliance fornece sistemas de automação de fluxo de trabalho para a NASA. Já a Rackerspace (empresa de hospedagem) usa Python em seu sistema CORE, que combina CRM e ERP. No Brasil, adotam a linguagem a Globo.com, o Governo Federal e o Senado.

O nome veio de uma série de TV dos anos 70, "Monty Python's Flying Circus".

Evolução da Linguagem

- Propostas de melhorias.
- Python Enhancement Proposals: PEPs.
- PEP 5. Guia para evolução da linguagem.
- PEP 6. Lançamentos de correção de bugs.
- PEP 8. Guia de estilo para código Python. PEP 20. Zen da linguagem Python.
- PEP 20. Zen da linguagem Python.

A evolução da linguagem Python se dá através de documentos com propostas de melhorias, cujo nome em inglês é Python Enhancement Proposals ou PEPs. Esses documentos devem ter uma especificação técnica e uma justificativa.

O propósito e as diretrizes das PEPs são descritos na PEP1. Na PEP5 são descritas as diretrizes para a evolução da linguagem, abordando especialmente o processo de quebra de compatibilidade com versões anteriores. Primeiro, uma proposta de quebra de compatibilidade deve ser descrita em forma de PEP. Em seguida deve ser implementada uma forma de obter o comportamento que está sendo depreciado. O passo seguinte é formalizar a depreciação do artefato na documentação da linguagem Python.

Podemos, opcionalmente, adicionar um aviso no interpretador Python para lembrar os usuários que o artefato está sendo depreciado sempre que ele for utilizado. Por fim, deve-se observar um intervalo de pelo menos um ano entre o lançamento da versão transitória e a versão que quebra a compatibilidade.

Existe também uma PEP para correção de erros, a PEP6. Entre outras coisas, a PEP6 descreve o processo de correção de bugs. Existe uma pessoa responsável por tomar decisões sobre a aceitação ou não de uma correção, que é chamada de Patch Czar. Uma correção de erro é adicionada ao tronco principal (trunk) do sistema de controle de versão. Uma pessoa, com permissão de escrita no controle de versão, chamado de commiter, deve avaliar se a correção pode ser incluída em um lançamento de versão de correção.

O Patch Czar decide quando há correções suficientes para justificar o lançamento de uma nova versão, lançamento esse que vai incrementar o número das micro revisões da versão, e que corresponde à letra C no esquema de numeração A.B.C mencionado anteriormente. Existe uma diretriz para o lançamento de correções a cada 6 meses, prazo que pode ser antecipado caso seja disponibilizada a correção de um erro crítico. Em geral, apenas a versão mais recente recebe correções de bug, ou seja, depois que é lançada a versão 3.4, a versão 3.3 vai sendo posta de lado (embora a comunidade incentive a atualização ou correção de versões mais antigas).

A PEP que orienta quanto ao estilo de programação, chamada PEP8, é fundamental para facilitar o compartilhamento de código nas comunidades Python. Entre outras coisas a PEP8 define a largura de indentação em quatro espaços, mantendo as linhas com no máximo 79 caracteres. Voltaremos a tratar de estilo e boas práticas de programação mais à frente.

Outra PEP muito conhecida pelos desenvolvedores Python é a PEP20. É chamada de "O Zen do Python". Para ler ou reler, basta abrir o interpretador do Python e executar o seguinte código:

```
>>> import this
```

Todo programador Python que se preze executou esse código em algum momento de sua carreira – alguns até conseguem citar todo o resultado verbalmente.

Exemplo 1: abrir o interpretador e rodar o comando que exibe o Zen do Python.

- python3
- Para encerrar:
  - Em Unix, Linux, BSD, Mac-OS: Ctrl-D.
  - No Windows: Ctrl+Z.
  - Ou com a função: quit().
- No Windows, eventualmente é necessário configurar a variável de ambiente PATH.

Funcionamento do Python

O interpretador da linguagem Python é frequentemente pré-instalado em distribuições GNU/Linux. Da mesma forma, já está presente no caminho de busca de programas padrão do sistema, a variável de ambiente PATH. Portanto, basta abrir um terminal para executar o interpretador:

```
# python
Python 2.7.3 (default, Mar 13 2014, 11:03:55)
[GCC 4.7.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Como pretendemos usar a versão 3 da linguagem, usaremos o seguinte comando:

```
# python3
Python 3.2.3 (default, Feb 20 2013, 14:44:27)
[GCC 4.7.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

No Sistema Operacional Microsoft Windows é necessário configurar a variável de ambiente PATH. Isso pode ser feito no console (cmd.exe), considerando que o Python foi instalado no diretório padrão:

```
> set path=%path%;C:\python34
```

Se por ventura o Python tiver sido instalado em outro local, é necessário alterar o comando acima para o local correto. Então será possível digitar o comando para iniciar o interpretador:

```
C:\> python
Python 3.4 (default, Apr 10 2014, 22.71:26) [MSC v.1500 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Para encerrar o interpretador, basta executar o comando quit() ou utilizar a tecla de atalho CTRL-D no GNU/Linux e CTRL+Z no Windows.

```
>>> quit()
#
```

O interpretador Python aceita diversas opções de linha de comando, por exemplo, -h (que exibe uma mensagem de ajuda):

- Opções do interpretador:
  - python3 -h
  - python3 -c COMANDO.
  - python3 -m MÓDULO.
- Opções após -c ou -m são ignoradas para serem processadas pelo módulo ou comando.

```
# python3 -h
usage: python3 [option] ... [-c cmd | -m mod | file | -] [arg] ...
Options and arguments (and corresponding environment variables):
-b     : issue warnings about str(bytes_instance), str(bytearray_instance)
         and comparing bytes/bytearray with str. (-bb: issue errors)
-B     : don't write .py[co] files on import; also PYTHONDONTWRITEBYTECODE=x
-c cmd : program passed in as string (terminates option list)
-d     : debug output from parser; also PYTHONDEBUG=x
-E     : ignore PYTHON* environment variables (such as PYTHONPATH)
-h     : print this help message and exit (also -help)
…
…
PYTHONHASHSEED: if this variable is set to 'random', the effect is the same
   as specifying the -R option: a random value is used to seed the hashes of
   str, bytes and datetime objects.  It can also be set to an integer
   in the range [0,4294967295] to get hash values with a predictable seed.
```

Outra opção interessante e muito usada é -c, que permite executar um comando Python:

```
# python3 -c 'print("Isso é só o começo!")'
Isso é só o começo!
```

Também é possível executar um módulo da biblioteca padrão, como por exemplo, iniciar um servidor de arquivos (para encerrar usamos a combinação de teclas Ctrl+C).

```
# python3 -m http.server 8000
Serving HTTP on 0.0.0.0 port 8000 ...
^C
Keyboard interrupt received, exiting.
#
```

Na versão 2 da linguagem, o nome do módulo era um pouco diferente:

```
# python -m SimpleHTTPServer 8000
Serving HTTP on 0.0.0.0 port 8000 ...
^CTraceback (most recent call last):
  File "/usr/lib/python2.7/runpy.py", line 162, in _run_module_as_main
...
...
...
Muitas mensagens de Traceback depois ...
    return func(*args)
KeyboardInterrupt
#
```

Exemplo 2: ao iniciar o interpretador, já é possível começar a programar em Python. Inicie o interpretador e digite o código exemplo a seguir:

```
# python3
Python 3.2.3 (default, Feb 20 2013, 14:44:27)
[GCC 4.7.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> meu_primeiro_codigo_python = True
>>> if meu_primeiro_codigo_python:
...     print("Me aguardem, vou ficar bom nisso!!")
...
Me aguardem, vou ficar bom nisso!!
>>>
```

Mais adiante vamos nos aprofundar nos conceitos desse código. Por enquanto é importante adiantar o seguinte:

- Variáveis em Python:
  - São referências a endereços de memória.
  - Não precisam ser declaradas.
  - Precisam ser inicializadas.

Não podemos usar uma variável que nunca tenha sido inicializada. Podemos fazer uma analogia com etiquetas coladas em caixas, a etiqueta é importante, mas o conteúdo da caixa é mais importante que a etiqueta. No código exemplo anterior, a variável meu_primeiro_codigo_python referencia o booleano True e é isso que importa para o Python, já que o interpretador sabe que se trata de um booleano e faz a avaliação necessária para booleanos.

Outro conceito importante desse código é a indentação, a linguagem Python delimita os blocos de código com indentação, assim aquilo que está indentado faz parte do bloco de código do comando if e será executado quando a expressão for avaliada como verdadeira, por isso a função print é executada. O bloco é finalizado quando pressionamos a tecla <ENTER> sem recuo de indentação em uma linha vazia. Então, o parâmetro da função print é impresso no terminal.

Trabalhar com o interpretador interativo é muito útil para aprender, testar algum trecho de código ou escrever pequenos programas. Entretanto, ter de escrever ou reescrever código toda vez que quiser executar uma tarefa não é muito produtivo. Como nas demais linguagens de programação, o correto é escrever o código em um arquivo, que todos conhecemos como código-fonte. Não é necessário, mas foi convencionado que arquivos com código-fonte Python possuam a extensão .py. Para executar o código Python, invocamos o interpretador passando o nome do nosso arquivo como parâmetro.

```
Módulo:
python3 meu_modulo.py
chmod u+x meu_modulo.py
./meu_modulo.py
```

Considere o código-fonte a seguir, em um arquivo chamado meu_modulo.py:

```
#! /usr/bin/env python3
# -*- coding: utf-8 -*-
# A linha acima é redundante, no python3 a
# codificação de caracteres já é utf-8.
# meu_modulo.py
print("Agora a coisa tá melhorando!")
```

Executamos o arquivo Python da seguinte forma:

```
# python3 meu_modulo.py
Agora a coisa tá melhorando!
#
```

As primeiras linhas do código são apenas comentários, mas a primeira e a segunda têm um significado especial. A primeira é muito conhecida de administradores de sistema GNU/Linux: ela indica qual o interpretador utilizado para executar o código. Nesse caso é o Python. Isso permite que o arquivo seja executado em sistemas GNU/Linux e similares, como o Unix, BSD e Mac OS (que é baseado em BSD). Basta que seja fornecida a permissão de execução, como no exemplo:

```
# chmod u+x meu_modulo.py
#./meu_modulo.py
Agora a coisa tá melhorando!
#
```

A segunda linha informa qual é a codificação de caracteres utilizada no código-fonte, nas versões anteriores à 3, isso era necessário para permitir o uso de acentos no código-fonte (a palavra “tá” da frase impressa nesse exemplo). No Python3 a codificação padrão do arquivo fonte é utf-8, por isso essa linha é opcional.

Exemplo 3: digite o código do meu_modulo.py em um arquivo, salve e execute.

Primeiro exemplo

Logo vamos nos aprofundar nas características da linguagem, mas primeiro vamos matar a curiosidade vendo um exemplo de um código completo. O código adivinha.py exibe algumas opções e solicita ao usuário que adivinhe qual fruta foi escolhida. Caso acerte o programa termina ou então solicita novamente.

```
1  #!/usr/bin/env python3
2  # adivinha.py - Primeiro exemplo de módulo.
3  """
4  Solicita que usuário adivinhe qual a opção escolhida.
5  """
6
7  import random
8
9
10 palpite, tentativas = "", 0
11 frutas = ["Pera", "Laranja", "Melancia", "Ameixa"]
12 minha_preferida = random.choice(frutas).upper()
13 while palpite != minha_preferida:
14     print(frutas)
15     palpite = input("Adivinhe minha fruta favorita: ")
16     palpite = palpite.upper()
17     tentativas += 1
18     if palpite != minha_preferida:
19         print("Não é essa não, tente novamente.")
20 print("Muito bem, você acertou. "
21       "Número de tentativas = {}".format(tentativas))
```

Vamos entender um pouco o código. A primeira linha, como vimos anteriormente, indica qual o interpretador deve ser usado quando o script for executado em sistemas Linux e similares. A segunda linha é apenas um comentário. Da linha 3 a 5 vemos uma string delimitada com três aspas, que podem ser simples ou duplas. Esse tipo de string é chamada de string multilinhas ou doc strings. São muito úteis para documentação do código.

Na linha 7 informamos ao interpretador que queremos utilizar o módulo random da biblioteca padrão (em português, usaríamos o termo importar).

Na linha 10, acontece uma atribuição múltipla. O primeiro elemento do lado esquerdo recebe o primeiro elemento do lado direito, e o segundo elemento do lado esquerdo recebe o segundo elemento do lado direito.

Na linha 11 inicializamos uma lista e na linha seguinte usamos o método choice, que vai sortear um elemento entre aqueles que compõem a lista, convertendo ainda as letras do item sorteado para maiúsculas através do método upper(). Na linha 13, usamos o comando de laço while, testando se o palpite do usuário é diferente da opção sorteada e repetindo o bloco de código que está indentado (linhas 14 até 19).

Na linha 14 imprimimos a lista de opções. Na linha 15 utilizamos a função input para solicitar uma informação do usuário. Na linha 16 transformamos o palpite do usuário em maiúsculas. A linha 17 contém um operador de atribuição e soma que funciona de forma similar ao código a seguir:

```
tentativas = tentativas + 1
```

Na linha 18 verificamos se o palpite, transformado em maiúsculas, é igual à opção sorteada, transformada em maiúsculas também. Isso permite que o usuário forneça uma opção com uma ou mais letras diferentes do esperado, por exemplo, "laranja". Se a opção for incorreta, a mensagem da linha 19 será impressa. Quando o usuário acertar, o laço será interrompido e a mensagem de acerto da linha 20 será então impressa. Repare que a string da mensagem continua na linha 21. Essa construção concatena o texto que começou na linha 20 com o texto da linha 21. A seguir vamos nos aprofundar nos conceitos da linguagem.

Exemplo 4: digitar o código do adivinha.py a partir da linha 7, salvar e executar.

Tipos numéricos

Os tipos numéricos mais comuns em Python são, inteiro, ponto flutuante e decimal. Os inteiros são do tipo int. Os números de ponto flutuante são do tipo float. Os números decimais são objetos da classe Decimal, definida no módulo decimal. Podemos realizar as operações aritméticas como em qualquer linguagem. Vamos ver alguns exemplos envolvendo inteiros e pontos flutuante:

```
>>> 7 + 3    # int + int ->  int
10
>>> 7 + 3.0  # int + float ->  float
10.0
>>> 7 * 3    # int * int ->  int
21
>>> 7 * 3.0  # int * float ->  float
21.0
>>> 7 / 3    # int / int ->  float
2.3333333333333335
>>> 7 // 3  # Divisão truncada de inteiros.
2
>>> 7 // 3.0  # Divisão truncada com float.
2.0
>>> 7 % 3    # Resto da divisão.
1
>>> 7 ** 3   # Potenciação.
343
>>>
```

Nos exemplos anteriores estão documentadas as conversões automáticas entre tipos da linguagem Python. A soma de dois inteiros resulta em um inteiro, enquanto a soma de inteiro com ponto flutuante resulta em um número de ponto flutuante. A divisão, seja de inteiro por inteiro, ponto flutuante por inteiro, ou ponto flutuante por ponto flutuante, resulta sempre em um número de ponto flutuante. A divisão truncada de inteiros ("//") resulta em um inteiro.
A divisão truncada com ponto flutuante ("//") resulta em um ponto flutuante. O resto e a potenciação seguem esse comportamento. Operações entre inteiros resultam em números inteiros. Operações com números de ponto flutuante resultam em números de ponto flutuante.

Quando estamos usando o interpretador interativo, os resultados das expressões são exibidos logo após a execução. Isso não ocorre quando executamos um código-fonte. O mais comum é armazenar os resultados em variáveis e exibir quando houver interesse, utilizando a função print(). Vamos ver mais alguns exemplos.

```
>>> calculo = 3*2 + 5
>>> calc     # Variáveis não inicializadas causam erro.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'calc' is not defined
>>> # Acessar variável apresenta seu valor no modo interativo.
>>> calculo
11
>>> cambio = 2.25
>>> valor_importacao = 800.00
>>> imposto = (valor_importacao - 500) * 1.05
>>> valor_importacao + imposto
1115.0
>>> # No modo interativo o resultado da última operação
>>> # fica armazenado na variável _
>>> _ * cambio
2508.75
>>>
```

As operações aritméticas seguem uma ordem de precedência. Importante saber no momento que a exponenciação (“**”) tem precedência sobre a multiplicação (“*”), divisão (“/”), divisão truncada (“//”) e módulo (“%”), que por sua vez tem precedência sobre a soma (“+”) e a subtração (“-”). Recomenda-se uso dos parênteses para deixar o código mais claro e garantir a execução do cálculo como esperado. O uso dos parênteses pode ser empregado para mudar essa ordem de precedência.

Na primeira linha do exemplo, primeiro é executada a multiplicação de 3 por 2 e em seguida a soma do produto com 5. A variável “calculo” recebe uma referência para o resultado.

A linguagem Python possui uma tipagem dinâmica. Isso quer dizer que o tipo de um conteúdo é avaliado em tempo de execução. Entretanto, não é possível referenciar uma variável que nunca foi inicializada. Quando isso é feito, ocorre o erro apresentado no exemplo. Ao acessamos o valor de uma variável já inicializada no modo interativo, o seu valor é exibido no terminal. O mesmo efeito pode ser obtido ao executar um arquivo com código Python se utilizarmos a função print().

Outra coisa interessante do interpretador no modo interativo é que ele armazena o resultado da última operação em uma variável especial, o sublinhado (“_”). Essa variável foi utilizada nas últimas linhas do exemplo anterior.

O módulo decimal oferece o tipo Decimal, útil para aplicações que precisam de representação decimal exata, controle sobre a precisão, controle sobre o arredondamento, rastreamento de casas decimais significativas e aplicações onde os usuários esperam resultados que correspondem a cálculos matemáticos feitos à mão [15]. Vamos ver alguns exemplos:

```
>>> from decimal import Decimal
>>> Decimal('1.0') % Decimal('0.1')  # Resto da divisão decimal.
Decimal('0.0')
>>> 1.0 % 0.1  # Resto da divisão ponto flutuante.
0.09999999999999995
>>> # Comparação decimal.
>>> sum([Decimal('0.1')]*10) == Decimal('1.0')
True
>>> # Comparação ponto flutuante.
>>> sum([0.1]*10) == 1.0
False
```

Para usar o tipo Decimal precisamos primeiro importar sua definição, que está no módulo decimal. Isso é necessário pois o Decimal não é um tipo nativo como strings, inteiros e ponto flutuante. Ele foi proposto pela PEP-327 [16] e faz parte da biblioteca padrão desde a versão 2.4 do Python. Na segunda linha é possível reparar que o cálculo do resto com decimal é exato. Já no cálculo do resto com ponto flutuante [17] ocorre a aproximação padrão decorrente dos limites inerentes à arquitetura dos computadores. Vamos ver mais alguns exemplos:

```
>>> from decimal import getcontext
>>> from decimal import Decimal
>>> getcontext()
Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999999,
Emax=999999999, capitals=1, clamp=0, flags=[], traps=[Overflow,
DivisionByZero, InvalidOperation])
>>> getcontext().prec = 4
>>> Decimal(7) / Decimal(3)
Decimal('2.333')
>>> getcontext().prec = 7
>>> Decimal(7) / Decimal(3)
Decimal('2.333333')
>>> Decimal('2.3333')  # Decimal à partir de string.
Decimal('2.3333')
>>> # Decimal à partir de float. Opa e a precisão?
>>> Decimal(2.3333)
Decimal('2.33329999999999992965626915975008159875869750976562 5')
>>> # Precisão serve para operações.
>>> Decimal(2.3333) + Decimal(10.1)
Decimal('12.43330')
```

No começo desse código é importada a função getcontext. Essa função é usada para acessar um contexto, que nesse caso é um ambiente para operações aritméticas. O contexto gerencia a precisão, regras de arredondamento, determina quais sinais são tratados como exceção e limita a faixa de expoentes. Podemos observar o uso do contexto nas mudanças de precisão e nas operações de divisão do exemplo. Perceba também que o Decimal pode ser criado utilizando números inteiros, strings e ponto flutuante. É possível observar também que a precisão, definida no contexto, só afeta operações.

Quando é criado um Decimal a partir de um ponto flutuante, a precisão não é observada.

Tipo string

- Strings podem ser representadas com aspas simples ou duplas.
- A \ pode ser usada para proteger as aspas.
- No modo interativo, o acesso à variável não expande caracteres especiais (\ + caractere).
- Para prevenir a interpretação de caracteres especiais, existem as raw strings.

A linguagem Python tem muitos atrativos, mas para muitas pessoas o tipo string é um dos mais interessantes por conta da sua facilidade de uso e versatilidade. As strings são representadas com aspas simples ou duplas. Se for necessário incluir o caractere aspa no conteúdo da string, podemos usar o caractere de escape (“\”). Vamos ver alguns exemplos:

```
>>> frase = 'Strings em python são muito poderosas.'
>>> frase = "É muito fácil usar ' ou \"."
>>> frase
'\xc3\x89 muito f\xc3\xa1cil usar \' ou ".'
>>> print(frase)
É muito fácil usar ' ou ".
>>> frase = 'c:\nome'
>>> frase
'c:\nome'
>>> print(frase)
c:
ome
>>> frase = r'c:\nome'
>>> frase
'c:\\nome'
>>> print(frase)
c:\nome
>>>
```

Ao acessar a variável frase na terceira linha do exemplo, vemos que o conteúdo exibido no terminal é meio estranho, cheio de barras e números. Acontece que dessa forma o interpretador nos mostra a string como ela é armazenada pela linguagem. Os caracteres acentuados são representados na forma de códigos hexadecimais. Quando usamos a função print(), os caracteres aparecem conforme esperamos. Caracteres especiais podem ser usados na string. Nesse exemplo temos o uso do \n, que representa nova linha (em inglês: new line), ou <ENTER>. Para evitar que o caractere \n seja tratado como nova linha, podemos usar duas barras consecutivas ou então usar uma string chamada de raw string, usando a letra “r” na frente da representação.

Mais à frente veremos como as strings multilinha são utilizadas para documentar o código, sendo por isso mesmo chamadas de docstrings.

Algumas das características de strings em Python são um tanto quanto intuitivas, mas outras são novidade, pois não são encontradas em outras linguagens de programação. As strings em Python podem ser declaradas com aspas triplas e são conhecidas como strings multilinhas. Ainda assim, se for necessário, é possível suprimir a inserção automática de caractere de fim de linha em strings multilinha usando o caractere de escape barra (“\”).

- Com aspas triplas são declaradas strings de múltiplas linhas. O final de linha é adicionado automaticamente.
- Para evitar isso, pode ser adicionada uma \ no fim da linha.
- O operador de adição + concatena strings. O operador de multiplicação repete a string.
- Strings literais em sequência são concatenadas automaticamente, mas não é possível fazer isso com variáveis e operações.

As strings podem ser concatenadas usando o operador de adição (“+”). A concatenação de strings literais, que corresponde ao texto entre aspas, pode ser obtido apenas escrevendo as strings em sequência – esse comportamento não funciona com variáveis. Vejamos alguns exemplos:

```
>>> frase = '''
... Strings grandes podem usar várias linhas.
...
... Mais legível do que diversos caracteres de fim de
... linha no meio do texto.
... '''
>>> frase = """\
... A \ no fim da linha evita nova linha automática
... """
>>> print(frase)
A \ no fim da linha evita nova linha automática.
>>> linguagem = "p" + 3*"y" + "thon"
>>> print(linguagem)
pyyython
>>> titulo = "Curso " "de" "python"
>>> print(titulo)
Curso depython
>>> descricao = ("Curso de introdução ao python e "
...              "Django para a ESR/RNP.")
>>> print(descricao)
Curso de introdução ao python e Django para a ESR/RNP.
>>>
```

Na linguagem Python, existe o conceito de tipos sequenciais, sendo a string um deles. Os tipos sequenciais são indexados, ou seja, é possível acessar seu conteúdo através de um índice. Os índices começam em 0, como os vetores de C, mas na linguagem Python é possível utilizar índices negativos.

- Strings podem ser indexadas, começando pelo índice 0.
- São aceitos índices negativos, a partir de -1, que é a última posição.
- Substrings podem ser obtidas pelo fatiamento de strings.
- No fatiamento, o começo é sempre incluído e o fim é sempre omitido.
- Por padrão, o primeiro parâmetro quando omitido é zero.
- O último parâmetro quando omitido é por padrão o tamanho da string.

Também é possível recuperar parte do todo, utilizando um conceito chamado de fatiamento. Para isso devem ser usados os colchetes (“[ ]”), informando o índice inicial e o final separados por dois pontos (“[i:f]”). No fatiamento o índice inicial é sempre incluído e o final sempre omitido.

Quando passamos apenas o índice final, o inicial é considerado 0. Quando passamos apenas o índice inicial, o final é considerado o tamanho da sequência. Seguem alguns exemplos:

```
>>> palavra = "Bicicleta"
>>> palavra[0]
'B'
>>> palavra[3]
'i'
>>> palavra[-1]  # Índice negativo, retorna a última letra.
'a'
>>> palavra[0:4]  # Fatiamento. Primeira-quarta letra = índice 3.
'Bici'
>>> palavra[4:7]
'cle'
>>> palavra[:4]  # Por padrão, inicia em 0.
'Bici'
>>> palavra[4:]  # Por padrão, fim da palavra.
'cleta'
>>>
```

- Índices negativos podem ser usados no fatiamento.

```
+---+---+---+---+---+---+
| P | y | t | h | o | n |
+---+---+---+---+---+---+
 0   1   2   3   4   5   6
-7  -6  -5  -4  -3  -2  -1
```

- Para entender o fatiamento, imagine os índices entre os caracteres.
- Usar um índice maior que a string causa uma exceção. Mas em fatiamento há um tratamento especial.
- Strings são imutáveis.
- A função len retorna o tamanho de uma string.

No fatiamento podemos usar índices negativos. Para facilitar o entendimento, imagine os índices entre os elementos, entre os índices 0 e 1 está a letra “P”, entre os índices 2 e 5 estão as letras “tho”, entre os índices -6 e -1 temos as letras “Pytho”:

```
+---+---+---+---+---+---+
| P | y | t | h | o | n |
+---+---+---+---+---+---+
 0   1   2   3   4   5   6
-7  -6  -5  -4  -3  -2  -1
```

Ao tentar acessar um elemento usando um índice maior que o tamanho da sequência, ocorre uma exceção. Entretanto, no fatiamento são admitidos índices maiores que o tamanho da sequência. Nesses casos, tomando como o exemplo da palavra “Python”, índices menores que -7 são tratados como se fosse o começo da sequência. Já os índices maiores que 6 são tratados como se fossem o tamanho da sequência. A string é um dos tipos imutáveis do Python, o que significa dizer que não é possível atribuir um valor a um elemento da string. Caso seja necessário, o tamanho de uma sequência pode ser obtido com a função len.

```
>>> palavra[11]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> palavra[4:11]
'cleta'
>>> palavra[11:]
''
>>> palavra[1] = 'I'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> 'Minha ' + palavra
'Minha Bicicleta'
>>> 'Moto' + palavra[2:]
'Motocicleta'
>>> len(palavra)
9
>>> palavra[::2]
'Bccea'
>>> palavra[::3]
'Bie'
>>> palavra[::-1]
'atelciciB'
>>>
```

O fatiamento de strings possui ainda uma terceira opção, que é o passo. Quando ele não é informado, o Python assume que ele é positivo e igual a 1. Os últimos três comandos do exemplo anterior mostram o uso do passo com tamanhos diferentes. Quando é passado o número 2, o fatiamento retorna o primeiro elemento, o terceiro elemento, o quinto elemento e assim por diante. Ao passar o número 3, são retornados o primeiro, o quarto e o sétimo elementos. E se for passado um passo negativo, o retorno é invertido, do último para o primeiro.

Formatando strings

- O método format() é usado para formatar strings.
- Chaves são usadas para indicar o local da substituição.
- Caractere dois pontos: usado para aplicar modificadores.
- Caractere < alinha à esquerda, > à direita, ^ centraliza. Pode ser passada uma largura. {:<30}
- Aceita referência a índice, {0} ou {1}.

Um dos métodos mais poderosos e úteis para manipulação de strings na linguagem Python é o método format(). Quando é necessário formatar strings utilizando variáveis de diversos tipos de dados misturados, são utilizadas marcas nos trechos onde deve ocorrer a substituição. Essas marcas são delimitadas com chaves ('{' e '}'). Ao chamar o método format, podem ser passados argumentos posicionais ou nomeados. Também é possível combinar os argumentos posicionais e nomeados.

Há uma gama grande de modificadores que podem ser usados. É possível modificar, entre outros, o alinhamento, casas decimais consideradas para arredondamento e preenchimento. Vejamos alguns exemplos.

```
src1/modelo09.txt
>>> '{}'.format('Formatando strings')
'Formatando strings'
>>> texto = 'Formatando strings'
>>> '{:<30}'.format(texto)
'Formatando strings           '
>>> '{:>30}'.format(texto)
'           Formatando strings'
>>> '{:^30}'.format(texto)
'      Formatando strings      '
>>> '{:*^30}'.format(texto)
'******Formatando strings******'
>>> 'Alô mamãe estou {:*^30} !!!'.format(texto.lower())
'Alô mamãe estou ******formatando strings****** !!!'
>>> '{0} é uma {1}'.format('maçã', 'fruta')
'maçã é uma fruta'
>>> '{1} é uma {0}'.format('maçã', 'fruta')
'fruta é uma maçã'
>>> '{0} é uma {0}'.format('maçã', 'fruta')
'maçã é uma maçã'
>>> '{1} é uma {1}'.format('maçã', 'fruta')
'fruta é uma fruta'
>>>
```

Na primeira linha do exemplo a frase 'Formatando strings.' é passada como parâmetro para o método format() de uma string que tem apenas a marca delimitando o local da substituição, gerando assim uma string com o conteúdo substituído.

Usando o caractere dois pontos ":", podemos aplicar modificadores que afetam a formatação. No exemplo anterior foi utilizado o modificador de alinhamento e de tamanho ':<30', indicando alinhamento à esquerda e com 30 caracteres de tamanho. Os outros exemplos alinharam o texto à direita ':>30', ao centro ':^30' e centralizado com 30 caracteres de tamanho e trocando espaços por asteriscos ':*^30'. Os caracteres que estão fora da delimitação das chaves não interferem na formatação, como foi demonstrado na formatação da frase 'Alô mamãe estou {:*^30} !!!'. Nesse caso foi passado como parâmetro o resultado do método texto.lower(), convertendo todos os caracteres da variável texto para minúsculas e efetuando a substituição depois.

Outra facilidade do método format() é que ele aceita uma sequência de valores para utilizar na substituição. O trecho seguinte do exemplo substitui os marcadores {0} e {1} pelos elementos passados como parâmetro nessa mesma ordem.

Também são aceitos parâmetros de outros tipos além de string.

- Modificador de inteiros é o caractere {d}.
- O alinhamento e a largura funcionam da mesma forma '{0:0<3d}'
- Podem ser representados como hexadecimais {0:x}, octais {0:o} ou binários {0:b}.
- Para usar o padrão 0x para hexadecimais {0:#x}, 0o para octais {0:#o} e 0b para binários {0:#b}.

Logo a seguir está um exemplo com números inteiros.

```
src1/modelo10.txt
>>> 'Idade -> {0}'.format(42)
'Idade -> 42'
>>> 'Idade -> {0:0>3d}'.format(42)
'Idade -> 042'
>>> 'Idade -> {0:x>3d}'.format(42)
'Idade -> x42'
>>> 'Idade -> {0:x<3d}'.format(42)
'Idade -> 42x'
>>> 'Idade -> {0:x^10d}'.format(42)
'Idade -> xxxx42xxxx'
>>> "int: {0:d}; hex: {0:x}; oct: {0:o}; bin: {0:b}".format(42)
'int: 42; hex: 2a; oct: 52; bin: 101010'
>>> "int: {0:d}; hex: {0:#x}; oct: {0:#o}; bin: {0:#b}".format(42)
'int: 42;  hex: 0x2a;  oct: 0o52;  bin: 0b101010'
```

O modificador de inteiros é a letra d, {d}. Se não for informada uma letra e for passado um inteiro como parâmetro, ele é apenas substituído como uma string. Quando é usado o delimitador {0}, significa apenas que o primeiro parâmetro passado será substituído no local indicado.

O alinhamento funciona da mesma forma que strings. Para alinhar à direita com preenchimento de zeros à esquerda '{0:0>3d}', ou com a letra x, '{0:x>3d}'. Para o alinhamento à direita temos '{0:x<3d}', mas nesse caso é recomendado evitar preencher com zeros, pois vai confundir com outro número ('420'). O modificador para centralizar também funciona '{0:x^10d}'.

Os modificadores x, o e b representam o número inteiro como hexadecimal, octal e binário. Para utilizar o símbolo que caracteriza hexadecimais, octais e binários, o modificador cerquilha (#) é usado. Assim hexadecimais iniciam com zero x (0x), octais com zero ó (0o) e binários com zero b (0b).

Entre outros tipos que o método format() aceita, está o ponto flutuante.

- Pode representar percentual usando modificador {:%}.
- É usado o caractere {:f} para números de ponto flutuante.
- Para exibir o sinal de números positivos, é usado o modificador mais {:+f}.
- A largura e a quantidade de casas decimais pode ser definida {:7.2f}.

```
src1/modelo11.txt
>>> desconto = 0.15
>>> 'Desconto: {:.2%}'.format(desconto)
>>> preco = 223.56
>>> 'Preço: {:f}'.format(preco)
'Preço: 223.560000'
>>> 'Preço: {:+f}'.format(preco)
'Preço: +223.560000'
>>> 'Preço: R$ {:7.2f}'.format(preco)
'Preço: R$  223.56'
>>> 'Preço: R$ {:9.2f}'.format(preco)
'Preço: R$    223.56'
>>> 'Preço: R$ {:5.2f}'.format(preco)
```

```
'Preço: R$ 223.56'
>>> 'Saldo: {:+5.2f}'.format(-20.45)
'Saldo: -20.45'
>>> 'Saldo: {:+5.2f}'.format(20.45)
'Saldo: +20.45'
>>> 'Saldo: {:5.2f}'.format(20.45)
'Saldo: 20.45'
>>> 'Saldo: {:5.2f}'.format(-20.45)
'Saldo: -20.45'
>>>
```

Um ponto flutuante pode representar percentual, usando o modificador '{:%}' e para representá-lo com duas casas decimais '{:.2%}'.

O modificador de ponto flutuante é a letra f, '{:f}'. Para mostrar o sinal de números positivos, o modificador mais usado é '{:+f}'. Delimitadores de tamanho podem ser usados, onde '{:7.2f}' indica que o número deve ser formatado com no mínimo sete caracteres. Um deles é o ponto separador da casa decimal, dois são as casas decimais e outros quatro a parte inteira. Se o número tiver menos do que 7 caracteres, serão acrescentados espaços à esquerda, mas se o número tiver mais do que sete caracteres, ele vai ser representado normalmente. Sempre que um número negativo for passado como parâmetro, o sinal negativo será apresentado. Para exibição do sinal positivo é necessário usar o modificador mais ({:+5.2f}).

Docstrings

- São strings literais na primeira linha em um módulo, função, classe ou definição de método.
- Tornam-se um atributo especial chamado __doc__.
- Por consistência devem usar aspas triplas.
- Podem ter uma linha ou multilinhas.

Docstrings com multilinhas devem ter uma linha com descrição curta, uma linha em branco seguida de uma descrição mais detalhada.

As docstrings são strings comuns, mas que devem ficar na primeira linha em um módulo, função, classe ou definição de método. Quando são utilizadas dessa forma, se tornam um atributo especial chamado __doc__.

Por uma questão de consistência, devem ser usadas aspas triplas para definir docstrings. As docstrings podem ter apenas uma linha ou podem ser divididas em várias linhas. Por isso é útil usar aspas triplas, mesmo em docstrings com uma linha, quando houver necessidade de aumentar a documentação e criar novas linhas que já usam aspas triplas. O padrão da linguagem Python é que as docstrings multilinha tenham uma linha com uma descrição curta, uma linha em branco e o restante da documentação com uma descrição mais detalhada.

```
#!/usr/bin/env python3
# modelo12.py
"""Módulo de exemplo de docstrings.
Módulo com exemplos de docstrings, define duas funções
simples para mostrar docstrings em uma linha e docstrings
multilinhas.
"""
```

```
def dobro(numero):
    """Calcula o dobro do número e retorna o resultado."""
    return numero * 2
def triplo(numero):
    """Retorna o triplo de um número.
    Calcula o triplo de um número e retorna o resultado.
    numero: valor numérico a ser triplicado.
    """
    return numero * 3
```

O exemplo possui uma docstring documentando o módulo, bem no começo. Define uma função com uma docstring com uma linha e outra função com uma docstring multilinha.

Para testar, basta abrir o interpretador, importar o módulo e usar a função help().

```
src1/modelo13.txt
# python3
Python 3.4.2 (default, Oct  8 2014, 10:45:20)
[GCC 4.9.1] on linux
Type "help", "copyright", "credits" or "license" for more ...
>>> import modelo12
>>> help(modelo12)
Help on module modelo12:
...
>>> print(modelo12.dobro.__doc__)
Calcula o dobro do número e retorna o resultado.
>>> help(modelo12.dobro)
Help on function dobro in module modelo12:
...
>>> help(modelo12.triplo)
Help on function triplo in module modelo12:
...
```

Doctests

- Doctests são semelhantes a docstrings.
- Permitem que o código seja testado rodando exemplos embutidos na docstring.
- Linhas iniciadas com “>>>” indicam exemplos de código.
- Linhas iniciadas com “...” indicam continuação do comando anterior.
- A linha seguinte ao comando indica a saída esperada.
- Linhas em branco encerram o exemplo.

Doctests são strings de documentação que utilizam aspas triplas com uma pequena diferença: doctests permitem que código seja testado rodando exemplos embutidos na docstring.

As linhas que iniciam com três sinais de maior que (>>>) representam o prompt do interpretador da linguagem Python. Essas linhas indicam um exemplo de código. Para representar blocos de código com mais de uma linha, são usados três pontos (...), imitando o comportamento do interpretador da linguagem Python. A linha seguinte ao exemplo de código representa a saída esperada do comando. Uma linha em branco indica o fim do exemplo.

```
#!/usr/bin/env python3
# modelo14.py
"""Módulo de exemplo de doctests.
>>> dobro(3) + triplo(4)
18
>>> dobro(5) + triplo(5)
25
"""
def dobro(numero):
    """Calcula o dobro do número e retorna o resultado.
    >>> dobro(6)
    12
    >>> dobro(7)
    14
    """
    return numero * 2
def triplo(numero):
    """Retorna o triplo de um número.
    >>> triplo(6)
    18
    >>> triplo(7)
    21
    """
    return numero * 3
if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

No exemplo anterior é documentada a execução das funções com valores de exemplo e os resultados esperados para a execução da função com esses valores.

Para rodar os testes, basta executar o módulo com o interpretador python. Se houver alguma falha, será exibido um relatório com a falha. Se não houver falhas, nada é exibido.

- Rodando o módulo, os testes são executados, mas se todos estiverem corretos, parece que nada aconteceu.
- # python3 modelo14.py
- Os testes são executados pelo método testmod() do módulo doctest, importado no fim do arquivo.

Os testes são executados pelo método testmod() do módulo doctest.

```
if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

Para que o relatório de execução completo seja exibido, pode ser usada a opção -v.

- O modo verboso apresenta um relatório de execução dos testes.
- # python3 modelo14.py -v

Serão exibidos todos os testes executados e as saídas esperadas, no final é apresentado um relatório sumarizado da execução.

```
# python3 modelo14.py -v
Trying:
    dobro(3) + triplo(4)
Expecting:
    18
ok
...
3 items passed all tests:
   2 tests in __main__
   2 tests in __main__.dobro
   2 tests in __main__.triplo
6 tests in 3 items.
6 passed and 0 failed.
Test passed.
```

Outra maneira de executar os testes de um módulo é chamando o módulo doctest pela linha de comando com a opção -m do interpretador da linguagem Python. Nesse modo pode ocorrer um problema se o módulo for parte de um pacote e importar algum submódulo, pois este não será reconhecido pelo módulo de testes e vai gerar um erro.

- Também é possível utilizar o módulo doctest pela linha de comando. Esse modo pode não funcionar se o módulo for parte de um pacote e importar algum submódulo.
- # python3 -m doctest -v modelo14.py

```
# python3 -m doctest -v modelo14.py
Trying:
    dobro(3) + triplo(4)
...
3 items passed all tests:
   2 tests in modelo14
   2 tests in modelo14.dobro
   2 tests in modelo14.triplo
6 tests in 3 items.
6 passed and 0 failed.
Test passed.
```

Trabalhando com datas

- O módulo datetime é usado para manipulação de data e hora.
- O foco principal é na extração de partes da data e hora, para formatação e manipulação.
- Permite operações aritméticas com data e hora.
- O tipo datetime recebe parâmetros inteiros na ordem, ano, mês, dia, hora, minuto e segundo.
- Outros módulos similarmente úteis são time e calendar.

O módulo datetime é usado para manipular data e hora, extrair partes, formatar, transformar texto ou inteiros para data e hora – e vice-versa. Também permite realizar operações aritméticas como soma e subtração de data e hora.

Além do módulo datetime, Python possui módulos que são especialmente úteis para lidar com hora e horários, chamado time. E um módulo com funções de calendário, chamado calendar.

Exemplo de manipulação de data

Para trabalhar com datas, é necessário importar o tipo datetime do módulo datetime. O tipo datetime é usado para obter datas.

```
src1/modelo15.xt
Exemplo de uso do módulo datetime.
>>> from datetime import datetime
>>> data = datetime(2015, 9, 7, 12, 15, 25)
>>> print(data)
2015-09-07 12:15:25
>>> print(repr(data))
datetime.datetime(2015, 9, 7, 12, 15, 25)
>>> print(type(data))
<class 'datetime.datetime'>
>>> print(data.year, data.month, data.day)
2015 9 7
>>> print(data.hour, data.minute, data.second)
12 15 25
>>> print(data.microsecond)
0
>>> print(data.strftime("%d/%m/%Y %H:%M:%S"))
07/09/2015 12:15:25
```

O tipo datetime recebe parâmetros inteiros na ordem, ano, mês, dia, hora, minuto e segundo. Ao imprimir o valor da variável data, ela é impressa usando um padrão chamado ISO 8601. A função repr mostra a representação da data como um objeto. A função type mostra de que tipo a variável data é, uma classe datetime do módulo datetime. O tipo datetime possui diversas propriedades, year que armazena o ano, month armazena o mês, day armazena o dia, hour as horas, minute os minutos, second os segundos e microsecond os microsegundos. O método strftime é utilizado para formatar a data utilizando como parâmetro uma string indicando o formato.

- A data e hora atual pode ser obtida com os métodos datetime.today() e datetime.now().
- O método datetime.today() não possui parâmetros.
- O método datetime.now() aceita um parâmetro para indicar o fuso horário.
- Se nenhum parâmetro for passado para datetime.now(), ele se comporta como o método datetime.today().

A classe datetime também permite que seja obtida a data e hora atuais. Dois métodos podem ser usados, now e today. Eles têm uma diferença: now aceita um parâmetro para indicar um zona de tempo de fuso horário, mas sem parâmetro eles se comportam da mesma forma.

```
src1/modelo16.xt
Exemplo de uso do módulo datetime.
>>> from datetime import datetime
>>> def hoje_para_teste():
...     return datetime(2015, 10, 15, 12, 15, 20)
>>> def agora_para_teste():
...     return datetime(2015, 10, 15, 12, 15, 21)
>>> def hoje(today=datetime.today):
...     return today()
>>> def agora(now=datetime.now):
...     return now()
>>> print(hoje(hoje_para_teste))
2015-10-15 12:15:20
>>> print(agora(agora_para_teste))
2015-10-15 12:15:21
```

No exemplo foram criadas duas funções que servem para gerar uma data fixa, simulando o funcionamento dos métodos today e now. Dessa forma é possível obter resultado previsível para os testes. Quando essas funções forem chamadas sem parâmetro, utilizam os métodos padrão: hoje() e agora() vão retornar datetime.today() e datetime.now().

- Com datetime.timedelta cálculos com datas e horas podem ser efetuados.
- Aceita como argumentos days para quantidade de dias, seconds para segundos, microseconds para microssegundos, miliseconds para milissegundos, minutes para minutos, hours para horas e weeks para semanas.
- Operações com duas datas resultam em um objeto timedelta.

O módulo datetime possui um tipo especial para cálculos com datas e horas, o tipo timedelta. Para utilizar o timedelta, deve ser indicada uma diferença de dias, horas ou segundos. Os dias são indicados pelo parâmetro days, horas pelo parâmetro hours e os segundos pelo parâmetro seconds. Cálculos envolvendo duas datas retornam um objeto timedelta.

```
src1/modelo17.txt
>>> from datetime import datetime, timedelta
>>> hoje = datetime(2015, 10, 15, 21, 2, 50, 976371)
>>> ontem = hoje: timedelta(days=1)
>>> amanha = hoje + timedelta(days=1)
>>> mes_passado = hoje: timedelta(days=30)
>>> proximo_mes = hoje + timedelta(days=30)
>>> print(ontem)
2015-10-14 21:02:50.976371
>>> print(amanha)
2015-10-16 21:02:50.976371
>>> print(mes_passado)
2015-09-15 21:02:50.976371
>>> print(proximo_mes)
2015-11-14 21:02:50.976371
```

```
>>> periodo_anterior = hoje: timedelta(hours=12)
>>> periodo_seguinte = hoje + timedelta(hours=12)
>>> print(periodo_anterior)
2015-10-15 09:02:50.976371
>>> print(periodo_seguinte)
2015-10-16 09:02:50.976371
>>> hoje - ontem
datetime.timedelta(1)
>>> print(hoje - ontem)
1 day, 0:00:00
>>>
```

No exemplo foi inicialmente criada uma data chamada de hoje, com o timedelta foram calculadas duas datas, um dia anterior e um dia posterior, ontem e amanha. Em seguida foi calculada uma data em um mês anterior e uma data em um mês posterior.

Para o exemplo de horas, foram calculados dois períodos de 12 horas: um adiante e um no passado. E por fim foi feito um cálculo com duas datas que resultou em um objeto timedelta de um dia.

Atividades práticas 

2

Outros tipos de dados, estruturas de controle e funções

objetivos

Apresentar outros tipos de dados da linguagem Python, assim como comandos que permitem controlar o fluxo de execução dos programas. O uso de funções complementa os recursos apresentados nesta sessão e que são mais utilizados na programação do dia a dia.

conceitos

Listas; Tuplas; Dicionários; Conjuntos; Booleanos; Valor nulo (none); Comandos if, while, for; Funções; Parâmetros; Variáveis e seu escopo.

Outros tipos de dados

Na sessão anterior, foi apresentado o tipo String, que é um tipo sequencial do Python, ou seja, é indexado. A seguir, vamos apresentar outros tipos de dados com essas características, bem como o tipo booleano.

Listas

- Um dos tipos mais versáteis para agrupamento de dados.
- A lista é representada usando colchetes.
- Aceita elementos de diversos tipos.
- Assim como strings, é indexada e pode ser particionada.
- Particionamento cria uma nova lista.
- Podem ser concatenadas.
- Listas são mutáveis.
- Permitem adição de partições.
- São capazes de conter listas aninhadas.
- Compreensão de listas geram novas listas.

As listas servem para agrupar objetos. Para inicializar uma lista, são usados colchetes. Entre os colchetes, os objetos são separados por vírgula. Também é possível utilizar a função nativa list. São aceitos objetos de todos os tipos, que podem ser misturados, inclusive criando listas dentro de listas (listas aninhadas).

As listas, como todos os tipos sequenciais, são indexadas e aceitam fatiamento e concatenação. O particionamento ou fatiamento gera uma nova lista. Uma diferença com relação às strings é que listas são objetos mutáveis, ou seja, é possível atribuir um valor a um item da lista. Permitem a adição de partições. Python permite ainda a compreensão de listas através da qual é possível gerar uma nova lista a partir de uma lista já existente.

```
src/modelo01.txt

>>> estados = ['AC', 'AL', 'AP', 'AM', 'BA', 'CE', 'DF', 'ES',
...     'GO', 'MA', 'mt', 'ms', 'mg', 'pa', 'pb', 'pr', 'pe',
...     'pi', 'rj', 'RM', 'RS', 'RO', 'RR', 'SC']
>>> estados[0]
'AC'
>>> estados[-1]
'SC'
>>> estados[-5]
'RM'
>>> estados[-5:]
['RM', 'RS', 'RO', 'RR', 'SC']
>>> estados[-5] = 'rn'
>>> estados[-5:]
['rn', 'RS', 'RO', 'RR', 'SC']
>>> estados[19] = 'RN'
>>> estados[-5:]
['RN', 'RS', 'RO', 'RR', 'SC']
>>> estados[19:]
['RN', 'RS', 'RO', 'RR', 'SC']
>>>

Continua ...
```

No exemplo anterior vemos a declaração de uma lista, indexação positiva e negativa, fatiamento e atribuição de valor a um elemento. Vejamos outros exemplos:

```
src/modelo01 (continuação)

>>> estados = estados + ['SP', 'SE']
>>> estados[-5:]
['RO', 'RR', 'SC', 'SP', 'SE']
>>> estados.append('TO')
>>> estados[-5:]
['RR', 'SC', 'SP', 'SE', 'TO']
>>> estados[10:19]
['mt', 'ms', 'mg', 'pa', 'pb', 'pr', 'pe', 'pi', 'rj']
>>> estados[10:19] = ['MT', 'MS', 'MG', 'PA', 'PB', 'PR',
...     'PE', 'PI', 'RJ']
>>> estados[10:19]
['MT', 'MS', 'MG', 'PA', 'PB', 'PR', 'PE', 'PI', 'RJ']
>>> estados[10:19] = []
```

```
>>> estados
['AC', 'AL', 'AP', 'AM', 'BA', 'CE', 'DF', 'ES', 'GO', 'MA',
 'RN', 'RS', 'RO', 'RR', 'SC', 'SP', 'SE', 'TO']
>>> estados[:] = []
>>> estados
[]
>>>
```

Os exemplos apresentados mostram a concatenação, anexação, atribuição de partição ou fatia, eliminação de elementos usando uma atribuição de partição com uma lista vazia e a eliminação de todos os elementos da lista usando também o fatiamento.

```
src/modelo02.txt

>>> vogais = ['a', 'e', 'i', 'o', 'u']
>>> len(vogais)
5
>>> numerais = [1, 2, 3, 4, 5]
>>> relacao = [vogais, numerais]
>>> relacao
[['a', 'e', 'i', 'o', 'u'], [1, 2, 3, 4, 5]]
>>> relacao[0]
['a', 'e', 'i', 'o', 'u']
>>> relacao[0][1]
'e'
>>> relacao[1][1]
2
>>> # Compreensão de listas.
>>> [letra.upper() for letra in vogais]
['A', 'E', 'I', 'O', 'U']
>>> [numero ** 2 for numero in numerais]
[1, 4, 9, 16, 25]
>>>
```

Nesse exemplo, usamos a função len para obter o tamanho da lista. Declaramos uma lista aninhada e acessamos elementos das listas aninhadas. No final, utilizamos a compreensão de listas para gerar novas listas. No exemplo, a lista gerada é apenas apresentada pelo interpretador, uma vez que não atribuímos esse resultado a nenhuma variável. Uma maneira de entender a compreensão de listas é ler o comando da seguinte forma, já traduzindo para o português: utilize o método upper de letra para cada letra na lista vogais. Ou então eleve o número ao quadrado para cada número na lista numerais.

Tuplas

- Assim como listas e strings, tuplas são um dos tipos de sequência do Python.
- Consistem em uma sequência de valores separados por vírgula.
- As tuplas são imutáveis, mas podem conter dados mutáveis como listas.
- Tuplas podem ser empacotadas ou desempacotadas durante uma atribuição.

As tuplas também fazem parte dos tipos sequenciais de Python. São criadas de forma semelhante a listas, exceto pelo fato de usarem parênteses em vez de colchetes. Podem ser criadas utilizando a função tuple, que recebe como parâmetro um iterável (um objeto que pode ser usado em laços). Vamos ver mais sobre isso adiante; no momento basta saber que tipos sequenciais são iteráveis.

As tuplas, assim como strings, são imutáveis. Por outro lado, podem conter dados mutáveis, como listas. Uma característica interessante de tuplas é a capacidade de empacotar e desempacotar valores. Vejamos os exemplos:

src/modelo03.txt

```
>>> coordenada = 13, 45
>>> coordenada
(13, 45)
>>> coordenada[0]
13
>>> pontos = coordenada, (3,89)
>>> pontos
((13, 45), (3, 89))
>>> vetores = ([3, 4, 7], [5, 8, 0])
>>> vetores
([3, 4, 7], [5, 8, 0])
>>> vazia = ()
>>> unica = (4,)  # Vírgula necessária para representar tupla.
>>> len(vazia)
0
>>> len(unica)
1
>>> x, y = coordenada
>>> x
13
>>> y
45
>>>
```

A atribuição de alguns valores separados por vírgula cria uma tupla, o que é chamado de empacotamento. Elas podem ser indexadas e particionadas. Podemos criar tuplas aninhadas, ou seja, tuplas dentro de tuplas e listas dentro de tuplas. Para criar uma tupla vazia, basta usar abre e fecha parênteses ou a função tuple() sem parâmetros. Já para criar uma tupla com um único elemento, é necessário usar uma vírgula extra após o elemento (isso é necessário para diferenciar de uma expressão matemática).

Quando atribuímos uma tupla a variáveis, ocorre o desempacotamento, mas o número de variáveis precisa ser igual ao número de elementos da tupla.

Dicionário

- Dicionários são conjuntos de pares chave:valor.
- Não existem chaves duplicadas.
- As chaves precisam ser de tipos imutáveis, como strings e números.
- São representados por listas de pares chave:valor separados por vírgula e entre abre '{' e fecha chaves '}'.

Os dicionários são um dos tipos nativos de Python, conhecido como um tipo de dado de mapeamento. Em outras linguagens existem estruturas parecidas, arrays associativos, memória associativa ou hashmaps.

São compostos por pares de chave e valor, e inicializados utilizando chaves (“{}”), par ou pares de chave valor, compostos pela chave, dois pontos (“:”) e o valor. Os pares devem ser separados por vírgulas. As chaves precisam ser de algum tipo imutável, como strings ou números. As chaves podem ser tuplas, desde que as tuplas não contenham dados mutáveis, como listas.

```
src/modelo04.txt

>>> estados = {"AC": "Acre", "AL": "Alagoas", "AP": "Amapá",
...     "AM": "Amazonas", "BA": "Bahia", "CE": "Ceará",
...     "DF": "Distrito Federal", "ES": "Espírito Santo",
...     "GO": "Goiás", "MA": "Maranhão", "MT": "Mato Grosso",
...     "MS": "Mato Grosso do Sul", "MG": "Minas Gerais",
...     "PA": "Pará", "PB": "Paraíba", "PR": "Paraná",
...     "PE": "Pernambuco", "PI": "Piauí", "RJ": "Rio de Janeiro",
...     "RN": "Rio Grande do Norte", "RS": "Rio Grande do Sul",
...     "RO": "Rondônia", "RR": "Roraima", "SC": "Santa Catarina",
...     "SP": "São Paulo", "SE": "Sergipe", "TO": "Tocantins"}
>>> siglas = sorted(estados.keys())
>>> siglas
['AC', 'AL', 'AM', 'AP', 'BA', 'CE', 'DF', 'ES', 'GO', 'MA', 'MG',
 'MS', 'MT', 'PA', 'PB', 'PE', 'PI', 'PR', 'RJ', 'RN', 'RO', 'RR',
 'RS', 'SC', 'SE', 'SP', 'TO']
>>> estados['AM']
'Amazonas'
>>> estados['DF']
'Distrito Federal'
>>>

Continua ...
```

- A instrução “del” remove um par (chave:valor) do dicionário.
- Atribuir valor a uma chave cria um elemento. Se a chave já existir, sobrescreve o valor anterior.
- Acessar uma chave inexistente gera um erro.
- Para verificar se uma chave existe, usar a instrução “in”.
- Usar o construtor “dict” cria um dicionário.
- Compreensão de dicionários cria novos dicionários.

Esse dicionário contém as siglas e os nomes dos estados brasileiros, a sigla é a chave e o nome é o valor de cada item do dicionário. Podemos obter uma lista das chaves usando o método keys e ordenamos a lista com a função sorted. Para obter um valor, nesse caso o nome de um estado, basta usar a chave como índice.

src/modelo04.txt (continuação).

```
>>> estados["ZZ"] = "zio zande"
>>> "ZZ" in estados
True
>>> estados["ZZ"] = "Zzio Zzande"
>>> estados["ZZ"]
'Zzio Zzande'
>>> del estados["ZZ"]
>>> "ZZ" in estados
False
>>> estados["ZZ"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'ZZ'
>>> relacao = dict(a=1, b=2, c=3, d=4)
>>> relacao
{'a': 1, 'c': 3, 'b': 2, 'd': 4}
>>> {item[0].upper(): item[1] for item in relacao.items()}
{'A': 1, 'C': 3, 'B': 2, 'D': 4}
>>>
```

Para criar um novo elemento em um dicionário existente, basta atribuirmos um valor a uma chave inexistente. O operador in é usado para verificar a existência de uma chave no dicionário. Atribuição de valor a uma chave existente sobrescreve o valor. O comando del remove um item do dicionário. Acessar uma chave inexistente causa uma exceção. Outro modo de criar um dicionário é utilizando o construtor dict. A compreensão de dicionários gera um novo dicionário, como vemos na última linha do exemplo.

Conjunto

- São coleções sem ordem definida.
- Não possui elementos repetidos.
- Suportam as operações matemáticas de união, interseção, diferença e diferença simétrica.
- Usam-se chaves ou a função set() para criar um conjunto.
- Para criar um conjunto vazio, é imperativo o uso da função set().
- Abre e fecha chaves { } cria um dicionário vazio.

O conjunto é um tipo de dados que faz parte de uma categoria chamada coleções. O conjunto é uma coleção de elementos únicos sem ordem definida. Suporta as operações matemáticas tradicionais de conjuntos: união, interseção, diferença e diferença simétrica.

Criamos conjuntos usando abre e fecha chaves (“{a,b,c}”), e entre as chaves os elementos separados por vírgula. Para criar um conjunto vazio, é necessário usar a função set (abre e fecha chaves sem elementos cria um dicionário vazio).

src/modelo05.txt

```
>>> reais = {1.0, 2.0, 3.0, 3.5, 4.0, 4.5, 5.0, 7.5}
>>> inteiros = set([1, 2, 3, 4, 5, 6])
>>> 3 in inteiros
True
>>> 3 in reais
True
>>> 3.5 in inteiros
False
>>> 3.5 in reais
True
>>> inteiros - reais  # Inteiros que não estão nos reais.
{6}
>>> reais - inteiros  # Reais que não estão nos inteiros.
{3.5, 4.5, 7.5}
>>> reais | inteiros  # Números que estão em um ou outro.
{1.0, 2.0, 3.0, 3.5, 4.5, 5.0, 4.0, 7.5, 6}
>>> reais & inteiros  # Números que estão nos dois conjuntos.
{1, 2, 3, 4, 5}
>>> reais ^ inteiros  # Números em um ou outro, mas não nos dois.
{3.5, 6, 4.5, 7.5}
>>>
```

Nesse exemplo, são criados dois conjuntos, reais e inteiros. O operador in serve para verificar a existência de um elemento no conjunto. A comparação segue o comportamento das comparações da linguagem Python. Assim, o número três (“3”) do tipo inteiro é igual ao número três (“3.0”) do tipo real, afinal três é três. O operador menos (“-”) fornece um novo conjunto que é a diferença entre os conjuntos. O operador pipe (“|”) é o operador de união. O operador “e” comercial (“&”) é o operador de interseção. O operador circunflexo (“^”) é o operador de diferença simétrica. Para saber mais sobre conjuntos, veja a referência da documentação da linguagem [1].

Booleanos

- A linguagem Python tem também um tipo booleano.
- O tipo booleano possui apenas dois valores, True e False. São equivalentes aos inteiros 0 e 1.
- As operações tradicionais de comparação resultam em booleanos. < (menor), <= (menor ou igual), > (maior), >= (maior ou igual), == (igual).

O tipo de dados booleano é muito popular em todas as linguagens, definido por George Boole. Esse tipo de dado é usado especialmente em álgebra booliana. Na linguagem Python, esse tipo é representado por dois valores: True e False, que são equivalentes aos inteiros 1 e 0.

Operações lógicas, ou de comparação, resultam em booleanos. A função bool avalia o parâmetro fornecido a ela e retorna True ou False. Essa mesma avaliação é válida para operações lógicas, usadas em comandos condicionais como o if e while, que serão vistos mais à frente.

```
src/modelo06.txt

>>> bool('')  # strings vazias são avaliadas como False.
False
>>> bool('v')  # strings com algum conteúdo são verdadeiras.
True
>>> bool('f')  # Não importa o conteúdo.
True
>>> bool('false')  # Basta que o tamanho seja maior que zero.
True
>>> bool('False')
True
>>> bool(False)  # O booleano False, é avaliado como False.
False
>>> bool(0)  # Zero é False.
False
>>> bool(1)  # Números diferentes de zero são True.
True
>>> bool(5002)
True
>>> bool([])  # Listas vazias são False.
False
>>> bool([False])  # Não importa o conteúdo.
True
>>> bool({})  # Dicionários e tudo mais, funcionam dessa forma.
False
>>> bool({'0': False})  # Se tem um elemento, é True.
True
>>> bool(set())  # Conjuntos não seriam diferentes.
False
>>> bool({False})
True
>>>
```

Em Python, o comportamento das avaliações de expressões boolianas segue um padrão. Zero ("0") é considerado falso, um ("1") é considerado verdadeiro. Assim, numerais, sejam inteiros, float ou outros com valor zero, são falsos. Se forem maiores que zero, são verdadeiros. Os outros tipos, strings, listas, dicionários e conjuntos são considerados falsos se estiverem vazios. Portanto, se o tamanho deles for maior que zero, são considerados verdadeiros.

Valor nulo, objeto None

- O valor nulo da linguagem Python é o None, que é do tipo NoneType.
- None representa a ausência de valor.
- Uma função que não retorna um valor explicitamente retorna None.
- Em teste booleanos, None é avaliado como False.
- Para testar se um valor é nulo, é usado o operador is.

O valor nulo da linguagem Python chama-se None. None é um objeto e na documentação da linguagem pode ser referenciado como NoneType. Se for realizada uma tentativa de atribuir algum valor a None, uma exceção será lançada. O valor None representa a ausência de valor. Uma função que não possui o comando return não retorna um valor explicitamente. Nesse caso, o valor retornado é o None.

Quando o valor None é utilizado em testes lógicos, é avaliado como False. Se houver necessidade de verificar se um valor é nulo, o operador adequado é o is.

```
src/modelo07.txt

Exemplo de valor nulo, None.

>>> def chama_o_print(texto):
...     if texto:
...         print(texto)
...     else:
...         print('Texto está vazio.')
>>> resultado = chama_o_print('O tempo não para.')
O tempo não para.
>>> resultado is None
True
>>> resultado = chama_o_print(None)
Texto está vazio.
```

O exemplo define uma função que não possui valor de retorno definido explicitamente, e por isso o comando return não é usado. Ao executar a função, o valor retornado é armazenado na variável resultado. Quando é testado se o resultado é nulo, a resposta é verdadeira (True). Quando a função é chamada e o valor None é passado como parâmetro, o valor do parâmetro texto é nulo (None). Ao avaliar o valor da variável texto com valor nulo, este é avaliado como falso e o bloco de código que está na cláusula else é executado, imprimindo "Texto está vazio.".

Estruturas de controle

- Blocos são formados por endentação.
- Iniciam pela palavra reservada do comando, seguido de uma cláusula que depende do comando e, para indicar o fim da cláusula, usa-se o carácter dois pontos (":").
- Recuo de indentação encerra bloco.

Os comandos da linguagem Python possuem uma sintaxe muito parecida, iniciam pela palavra reservada do comando, seguido de uma cláusula que depende do comando e, para indicar o fim da cláusula, usa-se o carácter dois pontos (":"). O delimitador de blocos é a indentação, decisão tomada pelo criador da linguagem para incentivar a legibilidade do código. O recuo de indentação encerra o bloco. Alguns comandos já foram vistos, mas vamos detalhar um pouco mais a partir de agora.

Comando condicional if

- Utilizado para controlar a execução de código.
- Apenas um dos blocos de código é executado.
- A cadeia de if - elif - else é equivalente ao switch/case de outras linguagens.
- Em Python não existe a sintaxe else if: o comando elif deve ser usado nesses casos.

O comando condicional if é utilizado para controlar o fluxo do código, por vezes chamado de desvio condicional ou desvio de fluxo. Em um comando condicional if, o bloco de código é executado somente se uma determinada condição é verdadeira. A linguagem Python não possui comando switch-case, comum em outras linguagens. Para obter comportamento semelhante em Python, faz-se uso da cadeia if-elif-else.

A linguagem Python não possui a sintaxe else if, como acontece em outras linguagens. Em Python, o mais adequado é utilizar o comando elif.

```
src/modelo08.txt

>>> num = int(input("Digite um número entre 1 e 10: "))
Digite um número entre 1 e 10: 20
>>> if num < 1:
...     print("O número deveria ser maior ou igual a 1.")
... elif num > 10:
...     print("O número deveria ser menor ou igual a 10.")
... else:
...     print("O quadrado do número é: ", num ** 2)
...
O número deveria ser menor ou igual a 10.
>>> operacao = 'a'
>>> operador1 = 4
>>> operador2 = 7
>>> if operacao == 'a':
...     print(operador1 + operador2)
... elif operacao == 's':
...     print(operador1 - operador2)
... elif operacao == 'd':
...     print(operador1 / operador2)
... elif operacao == 'm':
...     print(operador1 * operador2)
11
```

No exemplo anterior, o primeiro condicional if verifica o valor de um número e imprime apenas a mensagem adequada de acordo com o seu valor. Caso o valor esteja na faixa entre 1 e 10, é impresso o quadrado do número. O segundo condicional if simula o comportamento de uma calculadora muito simples, onde o valor da variável “operacao” determina qual das opções é executada. É importante nesse exemplo reparar que após as condições é utilizado o caractere dois pontos (“:”). As instruções print estão indentadas, o que indica que a instrução pertence ao bloco. O bloco encerra quando há um recuo na indentação. Quando é deixada uma linha em branco sem indentação, o comando é interpretado.

Comando de laço while

- O laço while executa o bloco de instruções enquanto sua condição for avaliada como verdadeira.
- O laço while pode ter uma cláusula else.
- O bloco de comandos else é executado quando a condição for avaliada como False.
- O bloco else não é executado se o laço for interrompido com break.
- O comando continue ignora as instruções posteriores e executa o laço novamente.

O comando de laço while executa um bloco de instruções enquanto a sua cláusula condicional for avaliada como verdadeira. Na linguagem Python, o comando while pode, opcionalmente, ter uma cláusula else. Os comandos presentes na cláusula else são executados assim que a condição for avaliada como falsa. O comando while pode ser interrompido com o comando break, todos os comandos do bloco do while são ignorados e a execução do laço de repetição é interrompida imediatamente. Quando o comando break for utilizado, a cláusula else não é executada. O comando continue simplesmente ignora as instruções seguintes forçando uma nova execução do laço de repetição.

```
src/modelo09.txt

>>> teste = False
>>> while teste:
...     teste = input('Digite algo, pressione <ENTER> para sair. ')
... else:
...     print('Estou no else do while.')
...
Estou no else do while.
>>> teste = True
>>> while teste:
...     teste = input('Digite algo, pressione <ENTER> para sair. ')
... else:
...     print('Estou no else do while.')
...
Digite algo, pressione <ENTER> para sair. oi
Digite algo, pressione <ENTER> para sair. tudo bem?
Digite algo, pressione <ENTER> para sair. 0
Digite algo, pressione <ENTER> para sair. False
Digite algo, pressione <ENTER> para sair.
Estou no else do while.
```

No código acima, quando a variável teste é falsa, nenhum comando do bloco principal do while é executado, imprimindo apenas a instrução da cláusula else. Na segunda parte do exemplo, quando a variável teste é verdadeira, os comandos do laço são executados até que seja pressionada a tecla <ENTER>. Isso faz com que a variável teste tenha o valor de string vazia “” e a string vazia é avaliada como falsa, então o código da cláusula else é executado.

```
src/modelo10.txt

>>> teste = True
>>> mensagem = 'Digite algo, digite break para sair. '
>>> while teste:
...     teste = input(mensagem)
...     if teste == 'break':
...         break
...     else:
...         continue
...     # Nunca será executado por causa do break e continue.
...     print('Nunca passa aqui.')
... else:
...     print('Estou no else.')
...
Digite algo, digite break para sair. Comando continue faz
Digite algo, digite break para sair. com que o comando
Digite algo, digite break para sair. print seja ignorado
Digite algo, digite break para sair. Break
```

No exemplo anterior, sempre que a variável teste for diferente da palavra break, o comando continue será executado, voltando ao início do laço e interagindo com o usuário novamente. Quando o usuário escrever a palavra break, o comando break será executado, interrompendo o laço. Dessa forma, o comando print('Nunca passa ....') nunca será executado, não importa o valor da variável teste. É possível perceber que a cláusula else do comando while será executada em uma situação específica: quando o usuário pressionar <ENTER>. Nesse caso, a variável teste terá o valor string vazia, será avaliada como sendo diferente de break, executando o comando continue e, ao voltar ao início do laço, será avaliada como falsa na cláusula while, executando então o comando else do while.

Atividades práticas 

Comando de laço for

- Itera sobre elementos de uma sequência.
- Também possui cláusula else, executada quando a condição do for termina.
- Também pode ser interrompido com break.
- Também pode ignorar comandos com o continue.

O comando for é utilizado para iterar sobre ou percorrer uma sequência e iteradores, lembrando que entre os tipos sequenciais estão strings, listas e tuplas. Assim como o while e o if, o comando for possui uma cláusula else. Da mesma forma que o while, pode ser interrompido com o comando break e pode ter comandos ignorados, usando o comando continue.

```
src/modelo11.txt

>>> cervejas = ['APA', 'IPA', 'STOUT', 'PORTER']
>>> for cerveja in cervejas:
...     print(cerveja)
...
APA
IPA
STOUT
PORTER
>>> palavra = 'mustela'
>>> for letra in palavra:
...     print(letra)
... else:
...     print('Estou no else.')
...
m
u
s
t
e
l
a
Estou no else.
>>>
```

O exemplo mostra a maneira como o comando for percorre uma lista. Podemos ler o comando for da seguinte maneira: “Para cada elemento na sequência, repita:”. Nesse caso, o código imprime cada palavra que está na lista. No segundo caso, o comando imprime cada letra da palavra. Ao final, executa o comando na cláusula else.

```
src/modelo12.txt

>>> palavra = 'mustela'
>>> for letra in palavra:
...     if letra == 's':
...             continue
...     if letra == 'a':
...             break
...     print(letra)
... else:
...     print('Estou no else.')
...
m
u
t
e
l
>>>
```

Nesse segundo exemplo, quando a letra possuir o valor 's', reinicia o comando for, graças ao continue. Quando a letra é 'a', encerra o for.

Funções

Função range

- Gera um iterador de números, similar a uma sequência.
- Pode ser transformado em lista.
- Pode ser fornecido o número inicial e final da faixa desejada.
- Um terceiro argumento opcional serve como incremento.
- O uso mais comum é em laços for.

O propósito da função range é fornecer um iterador de números. Os iteradores são um tipo de dados da linguagem Python feitos para serem usados em laços for. Por enquanto, basta saber que eles se comportam como uma lista quando usados em um comando for. Podem, também, ser transformados em listas e tuplas. A função range aceita três parâmetros: o número inicial, o número final e o passo ou incremento. Normalmente, a função range é utilizada em comandos for.

```
src/modelo13.txt

>>> range(5)  # objeto (iterador) range.
range(0, 5)
>>> print(list(range(5)))  # objeto transformado em lista.
[0, 1, 2, 3, 4]
>>> print(list(range(1,5)))  # aceita limite inicial e final.
[1, 2, 3, 4]
>>> print(list(range(1,10,2)))  # terceiro argumento é o incremento.
[1, 3, 5, 7, 9]
>>> for elemento in range(5):
...     print(elemento)
...
0
1
2
3
4
>>>
```

Quando invocamos a função range, o retorno é um objeto iterador. Ao transformar em lista, os valores dos elementos da lista são os valores assumidos pelo iterador. O parâmetro que delimita o final é exclusivo, ou seja, é o valor - 1.

Comando pass

- Usado para definir algo vazio, mas que é necessário sintaticamente.
- Python precisa disso, pois os blocos são definidos por indentação.
- Usualmente usado para definir classes mínimas.
- Pode ser usado como marcador em funções e condicionais, para lembrar de implementar algo mais tarde.

Como os blocos da linguagem Python são delimitados pela indentação, precisamos de um comando especial para definir um bloco vazio, seja um if, for, while vazios ou uma classe vazia. Para isso, existe o comando pass, cujo uso mais comum é para definir classes mínimas. Pode ser usado também como marcador, para lembrar que é necessário implementar algo, de modo a não perder a linha de raciocínio no meio do desenvolvimento de um algoritmo.

src/modelo14.txt

```
>>> while True:  # Repetirá para sempre.
...     pass     # Pressionar Ctrl-C para parar.
...
^CTraceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyboardInterrupt
>>> class Minima:
...     pass
...
>>> def super_algoritimo():
...     pass  # TODO: implementar essa semana.
...
>>>
```

Nesse exemplo, criamos um laço while vazio e eterno, que só se encerra se as teclas CTRL+C forem pressionadas ao mesmo tempo. Em seguida, é definida uma classe mínima. E por fim um exemplo de uma função que ainda não foi implementada.

Funções definidas pelo usuário

- É usada a instrução def para definir funções, seguido pelo nome e seus argumentos entre parênteses.
- O corpo da função começa na linha seguinte e precisa ser indentado.
- A primeira linha pode conter a documentação, escrita usando strings multilinha, as docstrings (com aspas triplas).
- Usamos a palavra-chave return para fornecer o valor de retorno.
- Funções podem ser aninhadas, definindo uma função dentro da outra.
- Python não possui procedures, funções sem retorno definido retornam None, o valor nulo de Python.

As funções definidas pelo usuário utilizam a palavra reservada def seguida pelo nome da função e abre e fecha parênteses. Opcionalmente, podem ser definidos parâmetros, sempre entre os parênteses, e separados por vírgulas. Finalizando temos, após os parênteses, o caractere dois pontos (":"), que indica o fim da primeira linha da definição. Em seguida pode ser escrito o conteúdo ou corpo da função. A linguagem Python aceita comandos na mesma linha, separados por ponto e vírgula (";"), mas para manter uma boa legibilidade o ideal é evitar essa construção, especialmente em funções grandes. O modo mais usado é iniciar a função na linha seguinte. Se na primeira linha do corpo da função houver uma string de múltiplas linhas, ela serve como documentação, por isso são chamadas de docstrings.
A docstring deve ser edentada, assim como o restante do corpo da função. Para fornecer um valor de retorno, utiliza-se a palavra-chave return.

```
src/modelo15.txt

>>> def somar(a, b): soma = a+b; return soma
...
>>> somar(3, 7)
10
>>> def somar(x, y):
...     '''Soma dois valores e retorna o resultado'''
...     soma = x + y
...     return soma
...
>>> somar(10,20)
30
>>> somar(10.0,-20)
-10.0
>>> def so_imprime(valor):
...     print(valor)
...
>>> resultado = so_imprime('Qual o retorno?')
Qual o retorno?
>>> print(resultado)
None
>>>

Continua ...
```

Quando as funções, métodos ou classes possuem uma docstring, podemos usar a função help para consultar essa documentação.

```
src/modelo15.txt - continuação.

>>> help(somar)
Help on function somar in module __main__:

somar(x, y)
    Soma dois valores e retorna o resultado
(END)
```

Dica: para retornar ao interpretador, é necessário pressionar a tecla "q".

Passagem de parâmetros

- Podemos fornecer valores padrão para parâmetros.
- Na passagem de parâmetros, é possível indicar o nome do parâmetro, que são chamados de keyword arguments.
- Existe uma sintaxe para passar um número variável de parâmetros e keyword arguments.
- Os argumentos podem ser desempacotados na chamada de funções.

Os parâmetros das funções em Python podem ter valores padrão. Os parâmetros que não possuem valor padrão são obrigatórios. Quando executamos a função, podemos usar o nome do parâmetro para fornecer o seu valor. Caso o nome do parâmetro não seja fornecido, é necessário respeitar a ordem em que foram definidos. Caso o nome do parâmetro seja fornecido, a ordem de passagem de parâmetros pode ser alterada. Chamamos os parâmetros nomeados de "keyword arguments". Existe uma sintaxe para passagem de uma quantidade variável de parâmetros e quantidade variável de argumentos nomeados. Também é possível desempacotar um conjunto de valores na passagem de parâmetros durante a execução da função.

```
src/modelo16.txt

>>> def potenciacao(base, expoente=2):    # (1)
...     resultado = base ** expoente
...     return resultado
...
>>> potenciacao(2)    # (2)
4
>>> potenciacao(3, 3)    # (3)
27
>>> def somatorio(*args):    # (4)
...     resultado = 0
...     for numero in args:    # (5)
...         resultado += numero
...     return resultado
...
>>> somatorio(1,2)
3
>>> somatorio(1,2,4,8,16)
31
```

- **# (1):** quando definimos uma função, inicializamos o valor padrão de um argumento.
- **# (2):** na execução da função, não é necessário fornecer o valor do argumento que possui valor padrão.
- **# (3):** ao fornecer um argumento, o valor padrão é sobrescrito.
- **# (4):** a técnica de empacotamento e desempacotamento de tuplas permite definir funções que recebem número indefinido de argumentos. Usa-se um asterisco para definir um argumento que receberá uma tupla.
- **# (5):** podemos então percorrer a tupla para utilizar os valores dos argumentos.

```
#!/usr/bin/env python3
# src/modelo17.py

"""
>>> item = {'Calça jeans': 1}
>>> compras_cliente1 = lista_de_compras(item)
>>> compras_cliente1
```

```
[{'Calça jeans': 1}]
>>> item = {'Camiseta': 1}
>>> lista_de_compras(item, compras_cliente1)
[{'Calça jeans': 1}, {'Camiseta': 1}]
>>> item = {'Bermuda': 1}
>>> compras_cliente2 = lista_de_compras(item)
>>> compras_cliente2
[{'Calça jeans': 1}, {'Camiseta': 1}, {'Bermuda': 1}]
"""

def lista_de_compras(item, compras=[]):
    compras.append(item)
    return compras
```

Um cuidado especial precisa ser tomado quando definimos uma função que recebe um parâmetro lista com valor padrão. A lista é um tipo mutável, criada no momento da declaração da função e que continuará existindo enquanto a função existir. Nos exemplos, podemos perceber que a lista vai acumulando os itens, mesmo quando é esperado que uma nova lista seja criada. Os itens do cliente2 são adicionados na lista de compras do cliente1. O fato é que elas são a mesma lista.

```
#!/usr/bin/env python3
# src/modelo18.py

"""
>>> item = {'Calça jeans': 1}
>>> compras_cliente1 = lista_de_compras(item)
>>> compras_cliente1
[{'Calça jeans': 1}]
>>> item = {'Camiseta': 1}
>>> lista_de_compras(item, compras_cliente1)
[{'Calça jeans': 1}, {'Camiseta': 1}]
>>> item = {'Bermuda': 1}
>>> compras_cliente2 = lista_de_compras(item)
>>> compras_cliente2
[{'Bermuda': 1}]
"""

def lista_de_compras(item, compras=None):
    if compras is None:
        compras = []
    compras.append(item)
    return compras
```

Para evitar o comportamento anterior, usa-se a seguinte técnica:

- **# (1):** declaramos o argumento lista com o valor padrão None, o nulo da linguagem Python.
- **# (2):** verificamos se o valor do argumento é None; se for, inicializamos uma lista vazia.

Agora, quando é criada a lista de compras do cliente2, de fato é criada uma nova lista, como é o esperado.

```
src/modelo19.txt

>>> def insere_elementos_lista(*args, valores=None):
...     if valores is None:
...         valores = []
...     for elemento in args:
...         valores.append(elemento)
...     return valores
...
>>> meus_elementos = []
>>> # Sem nomear o parâmetro, todos são empacotados em args.
... insere_elementos_lista(1, 2, 3, 4, 5, meus_elementos)
[1, 2, 3, 4, 5, []]
>>> meus_elementos
[]
```

Outro cuidado que precisa ser tomado quando declaramos a tupla de argumentos (*args) é que todos os argumentos não nomeados explicitamente são empacotados. No exemplo, a lista meus_elementos é tratada como um argumento empacotado em *args e é adicionada como um item na lista valores.

```
src/modelo20.txt

>>> # A lista é mutável, então o conteúdo é mantido após o
... # término da função.
... insere_elementos_lista(1, 2, 3, 4, 5, valores=meus_elementos)
[1, 2, 3, 4, 5]
>>> meus_elementos
[1, 2, 3, 4, 5]
>>> insere_elementos_lista(6, 7, 8, 9, valores=meus_elementos)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> meus_elementos
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> # Para evitar isso, deixar o parâmetro vazio e atribuir o
... # resultado para a lista meus_elementos.
... meus_elementos = insere_elementos_lista(1, 2, 3, 4, 5)
>>> meus_elementos
[1, 2, 3, 4, 5]
>>> meus_elementos = insere_elementos_lista(6,7,8)
>>> meus_elementos
[6, 7, 8]
>>>
```

Quando as listas são alteradas dentro da função, elas permanecem alteradas após a execução da função. Uma das maneiras de evitar isso é deixar de informar o argumento lista e atribuir o resultado à lista que vai guardar o resultado da função.

src/modelo21.txt

```
>>> def listar_musica(
...   nome, genero='Rock',
...   subgenero='Clássico', artista='Led Zeppelin'):
...     print('Música: ', nome, ', ', genero, ', ',
...           subgenero, ', ', artista, sep='')
...
>>> listar_musica('Stairway to heaven')
Música: Stairway to heaven, Rock, Clássico, Led Zeppelin
>>> listar_musica(nome='Whole Lotta Love')    # (1)
Música: Whole Lotta Love, Rock, Clássico, Led Zeppelin
>>> listar_musica(
...     nome='Travelling Riverside Blues',
...     subgenero='Blues')
Música: Travelling Riverside Blues, Rock, Blues, Led Zeppelin
>>> listar_musica(
...     artista='Pink Floyd',
...     nome='Wish You Were Here')    # (2)
Música: Wish You Were Here, Rock, Clássico, Pink Floyd
>>> listar_musica('Time', 'Rock', 'Clássico', 'Pink Floyd')
Música: Time, Rock, Clássico, Pink Floyd
>>> listar_musica('Money', artista='Pink Floyd')    # (3)
Música: Money, Rock, Clássico, Pink Floyd
>>>
```

Quando for preciso, podemos utilizar o nome do argumento na execução da função.

- **# (1):** definimos a função com argumentos que possuem valor padrão, então podemos omitir alguns argumentos e usar o nome do argumento para informar um valor diferente do valor padrão.
- **# (2):** ao usar o nome do argumento, podemos usar qualquer ordem.
- **# (3):** só precisamos respeitar a ordem se for informado um argumento sem nome, passamos todos os argumentos sem nome que quisermos e depois podemos passar os argumentos nomeados.

src/modelo22.txt

```
>>> def monta_pizza(nome, *args, **kwargs):
...     print('Pizza: ', nome)
...     print('Ingredientes: ', end='')
...     for ingrediente in args:
...         print(ingrediente, ' ', end='')
...     print()  # Nova linha.
...     print('-' * 20)
...     print('Opcionais: ', end='')
...     chaves = sorted(kwargs.keys())
...     for chave in chaves:
```

```
...         print(chave, '=', kwargs[chave], ' ', end='')
...     print()  # Nova linha.
...
>>> monta_pizza('portuguesa', 'cebola', 'ovo',
...             'presunto', 'queijo', 'orégano',
...             borda='cheddar', massa='integral',
...             espessura='fina')
Pizza:  portuguesa
Ingredientes: cebola  ovo  presunto  queijo  orégano
--------------------
Opcionais: borda = cheddar  espessura = fina  massa = integral
>>>

... Continua.
```

O exemplo anterior mostra como pode ser feito o desempacotamento de número indefinido de argumentos posicionais na tupla *args, assim como um número indefinido de argumentos nomeados no dicionário **kwargs. No código da função, basta tratar o argumento args como tupla e o argumento kwargs como dicionário. Além disso, usar um asterisco na declaração de args e dois asteriscos na declaração de kwargs.

```
src/modelo22.txt - continuação.

>>> ingredientes = ('cebola', 'ovo', 'presunto',
...                 'queijo', 'orégano')
>>> opcoes = {'borda': 'cheddar', 'massa': 'integral',
...           'espessura': 'fina'}
>>> monta_pizza('portuguesa', *ingredientes, **opcoes)
Pizza:  portuguesa
Ingredientes: cebola  ovo  presunto  queijo  orégano
--------------------
Opcionais: borda = cheddar  espessura = fina  massa = integral
>>>
```

Considerando a mesma função definida no exemplo anterior, podemos desempacotar argumentos. Para isso, basta definir uma tupla e um dicionário. Ao executar a função, desempacota-se a tupla utilizando um asterisco e o dicionário usando dois asteriscos.

Escopo de variáveis

- Por padrão, as variáveis têm escopo local.
- O comando nonlocal muda o escopo para o nível exterior.
- O comando global afeta a variável globalmente, ou seja, o escopo do módulo.

O escopo, em linguagens de programação, refere-se à abrangência. O escopo de uma variável está relacionado à abrangência da variável. Na linguagem Python, o escopo das variáveis pode ser local, não local ou então global. Por padrão, o escopo de uma variável é local. Para afetar o escopo de uma variável, é preciso informar isso explicitamente. A palavra-

-chave nonlocal é usada para informar que a variável possui um escopo externo ao local onde foi definida. A palavra-chave global indica que a variável possui um escopo abrangente, no nível do módulo. A figura 2.1 ilustra superficialmente o escopo de um módulo com uma função e uma função aninhada.

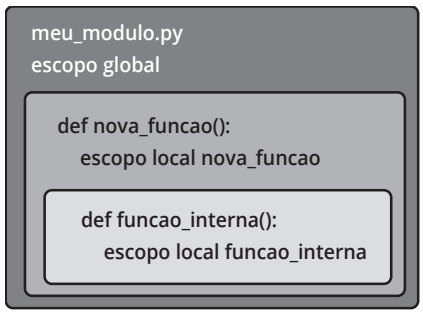

Figura 2.1
Ilustração do escopo de variáveis.

A função “nova_funcao” possui um escopo local e um escopo não local em relação à “funcao_interna”. A função “funcao_interna” possui seu escopo local e o módulo possui escopo global.

src/modelo24.txt

```
>>> def nova_funcao():
...     def funcao_interna_local():
...         abrangencia = 'local'
...         escopo_local = 'funcao_interna_local'
...     def funcao_interna_nonlocal():
...         nonlocal  abrangencia
...         abrangencia = 'primeiro nível'
...         escopo_local = 'funcao_interna_nonlocal'
...     def funcao_interna_global():
...         global  abrangencia
...         abrangencia = 'global'
...         escopo_local = 'funcao_interna_global'
...     abrangencia = 'inicial'    # 1
...     escopo_local = 'inicial'
...     funcao_interna_local()
...     print('Após função local:', abrangencia)    # 2
...     print('Após função local:', escopo_local)
...     funcao_interna_nonlocal()
...     print('Após função nonlocal:', abrangencia)    # 3
...     print('Após função nonlocal:', escopo_local)
...     funcao_interna_global()
...     print('Após função global:', abrangencia)    # 4
...     print('Após função global:', escopo_local)
...
>>> print(abrangencia)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'abrangencia' is not defined
>>> nova_funcao()
```

```
Após função local: inicial
Após função local: inicial
Após função nonlocal: primeiro nível
Após função nonlocal: inicial
Após função global: primeiro nível
Após função global: inicial
>>> print(abrangencia)
global
>>> print(escopo_local)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'escopo_local' is not defined
```

Nesse exemplo, a função “funcao_interna_local” altera uma variável chamada de abrangência. Como o escopo é local, assim que a função termina, o valor não está mais disponível. O mesmo ocorre com a variável escopo_local. A função “funcao_interna_nonlocal” utiliza a palavra-chave nonlocal, indicando que a variável pertence ao escopo externo à função, ou seja, o valor estará disponível após o encerramento da função, mas não estará disponível ao escopo global do módulo. Nesse caso, a variável escopo_local continua sendo local, pois o comando nonlocal não foi usado com ela.

A função “funcao_interna_global” utiliza a palavra-chave global, indicando que a variável pertence ao escopo global do módulo. Ou seja, o valor da variável estará disponível ao escopo do módulo; entretanto, a variável escopo_local não vai se comportar dessa forma.

- **# 1:** a variável “abrangencia” recebe um valor (“inicial”). Nesse ponto, seu escopo é local em relação à função “nova_funcao” e não local (nonlocal) em relação às outras, por enquanto. A variável escopo_local também está na mesma situação nesse momento.
- **# 2:** após a execução da função “funcao_interna_local”, o valor das variáveis ainda possui o valor “inicial”. O que aconteceu dentro da função não afetou o valor das variáveis.
- **# 3:** após a execução da função “funcao_interna_nonlocal”, a variável abrangencia é afetada. Foi usada a palavra-chave nonlocal, afetando assim a variável no escopo da “nova_funcao”. Mas a variável escopo_local continua se comportando como variável local.
- **# 4:** após a execução da função “funcao_interna_global”, a variável “abrangencia” ainda está com seu valor anterior, “primeiro nível”, pois seu escopo é local em relação à função “nova_funcao”. A palavra-chave global afetou a variável “abrangencia” em um escopo externo, o escopo do módulo. O valor alterado estará disponível ao fim da execução da função “nova_funcao”, como pode ser visto na penúltima linha do exemplo. Entretanto, a variável escopo_local não está disponível, pois foi definida no escopo interno da função “nova_funcao”.

Atividades práticas AVA

3
Orientação a objetos

objetivos

Assumindo que o aluno já tenha sido apresentado aos principais conceitos da programação orientada a objetos, nesta sessão começam a ser apresentados como tais conceitos são implementados na linguagem Python.

conceitos

Orientação a objetos; Classes e seus atributos; Herança; Métodos estáticos e privados.

Introdução

- Técnica de análise e/ou desenvolvimento.
- Organiza conceitos e comportamentos em entidades semelhantes.
- Favorece o uso de boas práticas de desenvolvimento de sistemas como: abstração e encapsulamento.
- Reutilização de código.
- Representação de conceitos próximos a realidade de usuários e analistas.
- Permite herança múltipla.
- Sobrescrita e chamada de métodos das classes hierarquicamente superiores.
- Métodos são declarados com um primeiro argumento explícito (“self”), representando o objeto (instância da classe). Esse argumento é implícito na chamada do método.
- Classes são objetos também.
- Tipos nativos podem ser estendidos.

A orientação a objetos é um modelo de análise, projeto e programação de sistemas de software baseado na composição e interação entre diversas unidades de software chamadas de objetos. Nesse contexto, existem duas técnicas ou etapas independentes, mas totalmente relacionadas, que podem ou não ser realizadas em conjunto. A primeira é a análise orientada a objetos, que descreve sistemas usando diagramas e documentos definidos de acordo com uma determinada metodologia, sendo UML a mais conhecida e difundida delas. A programação orientada a objetos é a outra técnica e é suportada pela grande maioria das linguagens de programação modernas.

A programação orientada a objetos é uma forma de organizar conceitos e comportamentos semelhantes em estruturas ou entidades, normalmente chamadas de classes, que podem ser depois reutilizadas. Entende-se que a programação orientada a objetos torna o desenvolvimento de software mais rápido e confiável, sendo acompanhada por uma série de recomendações, ou boas práticas, para apoiar o desenvolvimento de sistemas com qualidade.

Este curso não tem como objetivo descrever em detalhes o paradigma da análise ou programação orientada a objetos. Espera-se que o aluno já tenha sido apresentado a esses conceitos anteriormente, sendo capaz de entender o que são classes, atributos, métodos e herança, conforme sua implementação em Python seja apresentada. Diga-se de passagem que praticamente tudo é objeto em Python, incluindo funções e classes (exceto os operadores de adição, subtração etc.). Python também permite herança múltipla e os métodos das classes hierarquicamente superiores podem ser sobrecarregados. Isso tudo será abordado com o devido detalhe daqui para a frente.

Iniciamos com um diagrama de classes bem simples, em UML, que servirá de base para os exemplos que serão trabalhados a seguir.

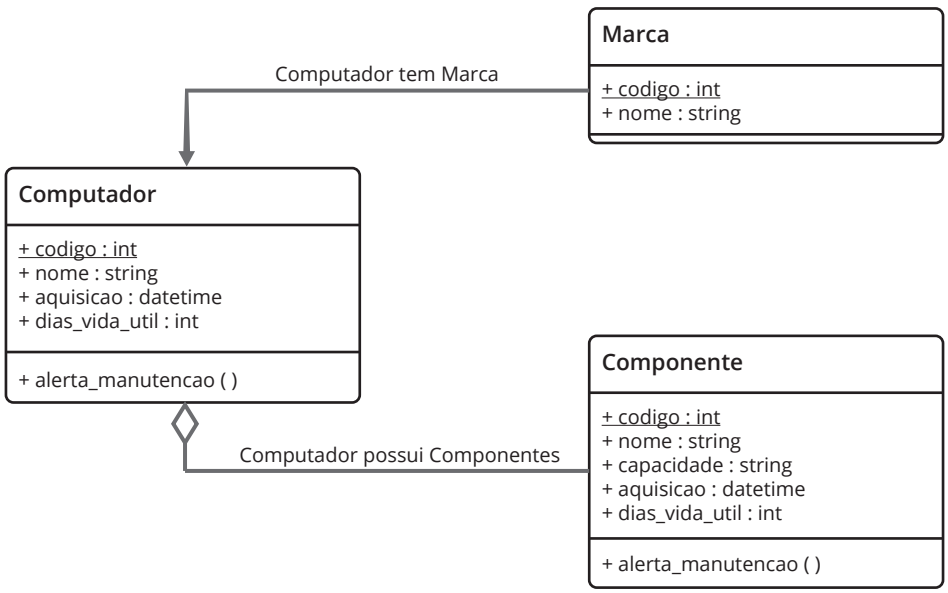

Figura 3.1
Diagrama de classes simples

Na figura temos dois relacionamentos: "Computador tem Marca" é um relacionamento chamado de associação, enquanto "Computador possui componentes" é um relacionamento chamado de agregação.

Classes em Python

As classes em Python são declaradas utilizando a palavra-chave class. A seguir, é apresentado o exemplo de uma classe Python que representa a classe Computador da figura 3.1.

```
#!/usr/bin/env python3
# modelo02.py

class Computador():
    def __init__(self, codigo, nome, aquisicao, vida, marca):
        self.codigo = codigo
        self.nome = nome
        self.aquisicao = aquisicao
        self.vida = vida
        self.marca = marca

    def alerta_manutencao(self):
        # TODO: calcular período de manutenção.
        pass
```

Em Python, os métodos precisam de um argumento explícito, que por convenção deve ser chamado de “self”. Esse argumento representa um objeto dessa classe. Quando um indivíduo ou instância dessa classe for criada, posteriormente será possível acessar os valores dele através desse nome “self”. Os tipos nativos do Python podem ser estendidos. Podemos, por exemplo, criar uma classe que se comporte como um inteiro, string ou dicionário.

Objetos em Python

Quando criamos uma variável dos tipos nativos, strings, inteiros, tuplas, listas, dicionários, conjuntos etc., estamos criando um objeto do tipo nativo em questão. O mesmo é válido em relação a classes, onde um objeto é considerado uma instância da classe.

No exemplo a seguir, ainda tomando como base a figura 3.1, temos a definição de duas classes, “computador” e “marca”. no final do exemplo, são criados dois objetos da classe “marca” e dois objetos da classe “computador”.

```
#!/usr/bin/env python3
# modelo03.py
class Computador():
    def __init__(self, codigo, nome, aquisicao, vida, marca):
        self.codigo = codigo
        self.nome = nome
        self.aquisicao = aquisicao
        self.vida = vida
        self.marca = marca
    def alerta_manutencao(self):
        # TODO: calcular período de manutenção.
        pass

class Marca():
    def __init__(self, codigo, nome):
        self.codigo = codigo
        self.nome = nome

if __name__ == '__main__':
    # Instancias(objetos) de marca.
    dall = Marca(1, 'Dall')
    lp = Marca(2, 'LP')
    # Instancias(objetos) de computador.
    vastra = Computador(1, 'Vastra', '10/01/2015', 365, dall)
    polvilion = Computador(2, 'Polvilion', '10/01/2015', 365, lp)

    print(type(dall))
    # <class '__main__.Marca'>
    print(type(polvilion))
    # <class '__main__.Computador'>
    print(isinstance(dall, Marca))
    # True
```

```
    print(isinstance(polvilion, Marca))
    # False
    print(isinstance(polvilion, Computador))
    # True
```

Nesse exemplo, além da criação dos objetos existe uma sutileza: estamos criando uma associação chamada “Computador tem Marca”. Primeiro um objeto da classe “marca” (dall) é passado para a classe “computador” (vastra).

Observe que na definição da classe “computador” essa associação foi criada através do comando: self.marca = marca.

Veja que foi utilizada a função type() para descobrir o tipo do objeto. Quando é necessário verificar se um objeto é de um determinado tipo, a função isinstance() pode ser usada. Quando foi testado se dall é uma instância da classe “marca” a função retornou True, retornando False para o teste se polvilion é uma instância da classe “marca”. Já quando foi testado se polvilion é uma instância da classe “computador”, a função retornou True.

Métodos

- Métodos são definidos como funções da Classe.
- Possuem um parâmetro explícito na definição.
- Recebem um parâmetro implícito na chamada, que é a referência da instância (objeto).

Os métodos dos objetos são definidos como funções de classe e possuem como primeiro parâmetro referência ao próprio objeto. Na chamada do método essa referência será passada implicitamente.

No exemplo a seguir, são apresentados os conceitos de métodos com mais detalhes.

```
#!/usr/bin/env python3
# src/modelo04.py

class Aluno:
    def __init__(self, matricula, nome):
        self.matricula = matricula
        self.nome = nome
        self.livros = []
    def empresta_livro(self, livro):
        self.livros.append(livro)

if __name__ == '__main__':
    # Aluno matriculado na instituição de ensino.
    # A classe aluno representa um aluno da instituição de ensino.
    print(type(Aluno.empresta_livro))
    # <class 'function'>
    jorge = Aluno(12345, 'Jorge da Capadócia')
    print(type(jorge.empresta_livro))
    # <class 'method'>
    Aluno.empresta_livro(jorge, 'Matemática essencial')
    print(jorge.livros)
```

```
    # ['Matemática essencial']
    jorge.empresta_livro('Pedagogia do ensino primário')
    print(jorge.livros)
    # ['Matemática essencial', 'Pedagogia do ensino primário']
```

Quando definimos um método, ele é uma função de classe, como vemos nesse trecho do exemplo:

```
print(type(Aluno.empresta_livro))
# <class 'function'>
```

Quando um objeto é criado, a referência ao método é reconhecida como um método do objeto, como pode ser visto no outro trecho do código:

```
print(type(jorge.empresta_livro))
# <class 'method'>
```

Os métodos possuem em sua definição um parâmetro explícito, que é uma referência ao próprio objeto, normalmente chamado self. Quando o método é chamado, esse parâmetro é passado implicitamente.

```
jorge.empresta_livro('Pedagogia do ensino primário')
```

Invariavelmente, quando uma função é definida com certo número de parâmetros, ela precisa ser chamada com o mesmo número de parâmetros. Mas por que isso não acontece com os métodos? Na verdade, isso acontece, sim. O que acontece com os métodos é equivalente ao seguinte trecho de código:

```
Aluno.empresta_livro(jorge, 'Matemática essencial')
```

Dessa forma, o comportamento padrão da linguagem é respeitado, usando uma sintaxe simplificada que facilita o trabalho do programador. O método é chamado e o primeiro parâmetro é a referência do objeto (jorge), enquanto o segundo parâmetro é o nome do livro (‘Matemática essencial’).

Herança em Python

Não custa lembrar que a herança é outro tipo de ligação ou relação entre classes. Essa relação também é conhecida como especialização ou generalização. Uma das classes é chamada de classe Pai, ou classe Base, enquanto que a outra é chamada de classe Filho ou classe Derivada. A classe Filho herda da classe Pai suas características e comportamentos. A classe Filho, por outro lado, sempre tem uma característica própria ou uma pequena mudança de comportamento.

Na figura 3.2, temos um diagrama representando uma hierarquia entre classes.

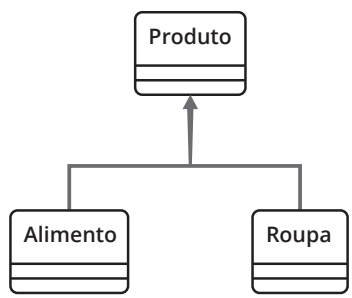

Figura 3.2
Diagrama simples com herança

O código a seguir mostra a implementação das classes da figura anterior.

```
#!/usr/bin/env python3
# src/modelo05.py

class Produto():
    def __init__(self, nome, imposto, custo):
        self.nome = nome
        self.imposto = imposto
        self.custo = custo
    def preco(self, quantidade):
        return (self.custo +
               (self.custo * self.imposto)) * quantidade

class Alimento(Produto):
    def __init__(self, nome, imposto, custo, validade):
        super().__init__(nome, imposto, custo)    # python3
        self.validade = validade

class Roupa(Produto):
    def __init__(self, nome, imposto, custo, tamanho):
        super().__init__(nome, imposto, custo)    # python3
        self.tamanho = tamanho

if __name__ == '__main__':
    bola = Produto('Noique', 0.5, 120.0)
    macarrao = Alimento('Espaguete', 0.4, 5.0, 30)
    camisa = Roupa('Calvo', 0.45, 30.0, 'G')
    print(bola.preco(1))
    # 180.0
    print(macarrao.preco(2))
    # 14.0
    print(camisa.preco(1))
    # 43.5
```

Os exemplos na string de testes, doctests, mostram a chamada do método “preco”, que reaproveita o código da classe superior, “Produto”. O código não precisou ser reimplementado nas classes derivadas, “Roupa” e “Alimento”.

A diferença entre as classes é que enquanto “Produto” é genérica (representando produtos em geral), “Alimento” e “Roupa” possuem atributos específicos, que são respectivamente validade e tamanho. Todos possuem nome, imposto e custo. Outro detalhe é a chamada do método da classe base na implementação das classes “Alimento” e “Roupa”. Em vez de inicializar os atributos diretamente, é utilizada a função super para chamar o método __init__ da classe “Produto”.

Atividades práticas AVA

Atributos de classe

- Os atributos da classe são compartilhados pelas instâncias (objetos).
- Podem ser alterados pelas instâncias.
- Se forem de tipos imutáveis não afetam a classe. (modelo06.py)
- Se forem mutáveis afetam a classe. (modelo07.py)

As classes também podem ter atributos, úteis para compartilhar valores padrão entre suas instâncias. Entretanto, é necessário entender como funcionam, especialmente quando é realizada uma atribuição. Em uma atribuição feita usando o nome da classe, suas instâncias terão acesso a esse valor. Caso seja feita uma atribuição em uma de suas instâncias, apenas o valor dessa instância será afetado. Por outro lado, se o atributo for um objeto mutável como uma lista, e o conteúdo desse objeto mutável for alterado, a classe e as outras instâncias terão acesso ao objeto alterado.

Na figura 3.3, o valor do atributo é compartilhado entre a Classe e os objetos. O mais comum é dizer que eles compartilham a mesma referência, ou referenciam o mesmo valor na memória.

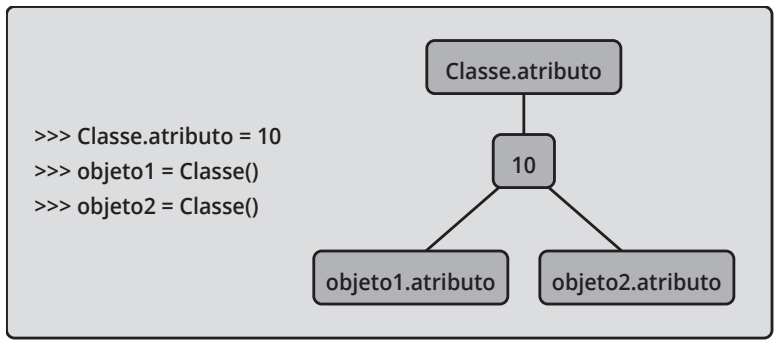

Figura 3.3 Objetos compartilham o atributo da classe.

Na figura 3.4, quando é realizada uma atribuição diretamente para o atributo do objeto, são criadas novas referências independentes. O atributo da Classe não é afetado.

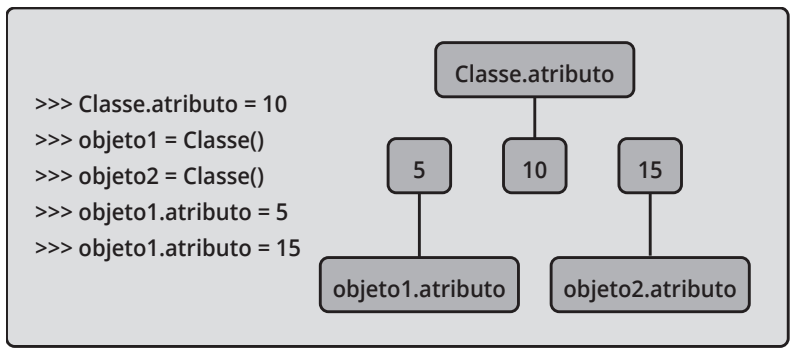

Figura 3.4 Atributos de instância possuem referências diferentes.

A seguir, alguns exemplos para esclarecer esses conceitos.

Exemplo de atributo de classe

Esse exemplo mostra o funcionamento de atributos de classe de tipos imutáveis. No caso, os atributos contador e limite, do tipo inteiro, são definidos na classe com o valor 0 (zero) para o contador e 3 para o limite. A explicação desse código está logo após o exemplo.

```
#!/usr/bin/env python3
# modelo06.py
class ObjetoContavel:
    contador = 0
    limite = 3

class ComentarioContavel(ObjetoContavel):
    def __init__(self, comentario):
        if self.contador >= self.limite:
            raise Exception
        else:
            self.__class__.contador += 1
        self.comentario = comentario


class PostagemContavel(ObjetoContavel):
    def __init__(self, postagem):
        if self.contador >= self.limite:
            raise Exception
        else:
            self.__class__.contador += 1
        self.postagem = postagem

if __name__ == '__main__':
    postagem = PostagemContavel('Eu acho python muito legal!')
    outra_postagem = PostagemContavel('Python é massa!')
    comentario = ComentarioContavel('Concordo com a sua opinião.')
    print(postagem.contador)
    # 2
    print(comentario.contador)
    # 1
    print(ObjetoContavel.contador)
    # 0
    print(PostagemContavel.contador)
    # 2
    print(ComentarioContavel.contador)
    # 1
    print(ObjetoContavel.limite)
    # 3
    print(postagem.limite)
    # 3
    print(comentario.limite)
    # 3
    postagem.limite = 5
    print(ObjetoContavel.limite)
    # 3
    print(ComentarioContavel.limite)
    # 3
```

```
print(PostagemContavel.limite)
# 3
outro_comentario = ComentarioContavel('Só, pode crer.')
print(ComentarioContavel.contador)
# 2
mais_um_comentario = ComentarioContavel('Certeza mano.')
print(ComentarioContavel.contador)
# 3
# Descomentar para ver a exceção. Comentar para executar
# os comandos abaixo. A exceção interrompe o programa.
# comentario_extra = ComentarioContavel('Tambem acho.')
# Traceback (most recent call last):
#     ...
# Exception
ObjetoContavel.limite = 10
print(postagem.limite)
# 5
print(comentario.limite)
# 10
print(ComentarioContavel.limite)
# 10
comentario_extra = ComentarioContavel('Tambem acho.')
print(comentario_extra.contador)
# 4
print(ObjetoContavel.contador)
# 0
print(ComentarioContavel.contador)
# 4
```

No início do código principal, dentro do if __name__ == '__main__', algumas instâncias de objetos são criadas, duas postagens e um comentário.

```
postagem = PostagemContavel('Eu acho python muito legal!')
outra_postagem = PostagemContavel('Python é massa!')
comentario = ComentarioContavel('Concordo com a sua opinião.')
```

Quando as instâncias são criadas, o código do método __init__ dos objetos é executado. Ele verifica se o limite de objetos foi atingido e, nesse caso, emite uma exceção. Senão, apenas soma o atributo contador. É importante reparar que o acesso ao atributo contador foi realizado de uma forma diferente: para alterar o atributo contador da classe é necessário acessar a referência da classe __class__. De outra forma seria criado um atributo contador na instância, e não é isso o que queremos.

```
def __init__(self, comentario):
    if self.contador >= self.limite:
        raise Exception
    else:
        self.__class__.contador += 1
```

Usando a referência da classe __class__, conseguimos um resultado interessante. Todas as instâncias de PostagemContavel compartilham o valor de contador dessa classe. No exemplo, é impresso o valor de postagem.contador com o valor 2 e da classe PostagemContavel.contador, que também é 2. A mesma coisa acontece com ComentárioContavel e comentário, só que o valor é 1. Isso acontece porque cada vez que criamos uma instância de PostagemContavel o contador dela é incrementado. Quando é criada uma instância da classe ComentarioContavel, é o contador dela que é incrementado.

E ObjetoContador não sofreu alteração. Outra forma de fazer isso seria usando o nome da classe diretamente: PostagemContavel.contador += 1 ou ComentarioContavel.contador += 1, teriam o mesmo efeito que self.__class__.contador += 1. Após os comandos existe uma linha comentada mostrando a saída esperada.

```
print(postagem.contador)
# 2
print(comentario.contador)
# 1
print(ObjetoContavel.contador)
# 0
print(PostagemContavel.contador)
# 2
print(ComentarioContavel.contador)
# 1
```

Agora, se fizermos alguma alteração em um atributo de classe usando a referência da instância, as outras instâncias e a classe não serão afetadas. Na verdade o que acontece é que criamos um novo atributo na instância e o mecanismo da linguagem Python percebe que existe um atributo na instância e para de procurar pelo nome.

Quando o mecanismo não encontra o atributo na instância, ele "sobe" na hierarquia para ver se existe um atributo com o mesmo nome na classe. Nas três linhas a seguir o limite tem o mesmo valor. Alteramos o valor do atributo limite na instância na linha seguinte, para 5. Mesmo assim o valor do atributo limite nas três classes continua o mesmo, 3. Por isso usamos a referência __class__ no método __init__, como foi explicado.

```
print(ObjetoContavel.limite)
# 3
print(postagem.limite)
# 3
print(comentario.limite)
# 3
postagem.limite = 5
print(ObjetoContavel.limite)
# 3
print(ComentarioContavel.limite)
# 3
print(PostagemContavel.limite)
# 3
```

Criamos mais alguns comentários, o contador vai sendo incrementado e quando o contador chega ao limite, uma exceção é lançada. Alteramos o valor do limite na classe ObjetoContavel. O valor do atributo na instância postagem não sofre alteração, pois criamos um atributo de instância que "sobrescreveu" o atributo da classe, de acordo com o que foi explicado anteriormente. Mas as instâncias de comentário ComentarioContavel e PostagemContavel enxergam essa mudança, tanto é que no exemplo imprimem o valor 10.

```
outro_comentario = ComentarioContavel('Só, pode crer.')
print(ComentarioContavel.contador)
# 2
mais_um_comentario = ComentarioContavel('Certeza mano.')
print(ComentarioContavel.contador)
# 3
# Descomentar para ver a exceção. Comentar para executar
# os comandos abaixo. A exceção interrompe o programa.
# comentario_extra = ComentarioContavel('Tambem acho.')
# Traceback (most recent call last):
#     ...
# Exception
ObjetoContavel.limite = 10
print(postagem.limite)
# 5
print(comentario.limite)
# 10
print(ComentarioContavel.limite)
# 10
```

Agora que o limite foi alterado, podemos criar mais um comentário sem problema. O contador de ComentarioContavel vale 4 agora e o contador de ObjetoContavel não sofreu alteração, como era esperado.

```
comentario_extra = ComentarioContavel('Tambem acho.')
print(comentario_extra.contador)
# 4
print(ObjetoContavel.contador)
# 0
print(ComentarioContavel.contador)
# 4
```

O comportamento dos atributos de classe é um pouco diferente quando se trata de um tipo mutável. Quando lidamos com tipos imutáveis, as atribuições mudam a referência ao objeto na memória. A figura 3.5 ilustra a inicialização de uma variável string chamada descrição (lembre-se de que strings são imutáveis).

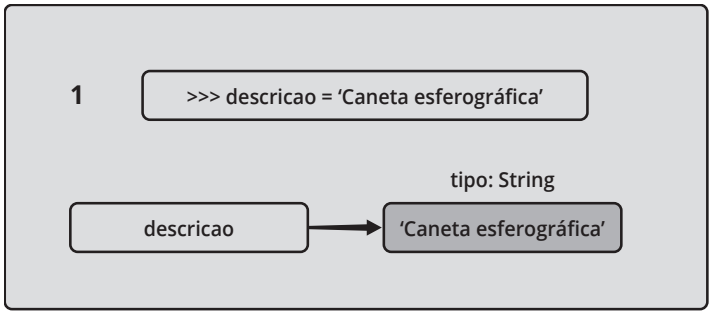

Figura 3.5
Inicialização de uma string.

Os tipos imutáveis não possuem métodos para alterar seu conteúdo, então o que normalmente é realizado é uma nova atribuição. A figura 3.6 ilustra o que acontece quando realizamos uma nova atribuição. A referência antiga de descrição que apontava para o objeto string "Caneta esferográfica" é eliminada. Uma nova referência é criada para o objeto string "Cola bastão". Se não houver nenhuma outra referência para o objeto string "Caneta esferográfica", no momento apropriado o coletor de lixo ("garbage collector") da linguagem Python liberará a memória ocupada pelo objeto "Caneta esferográfica".

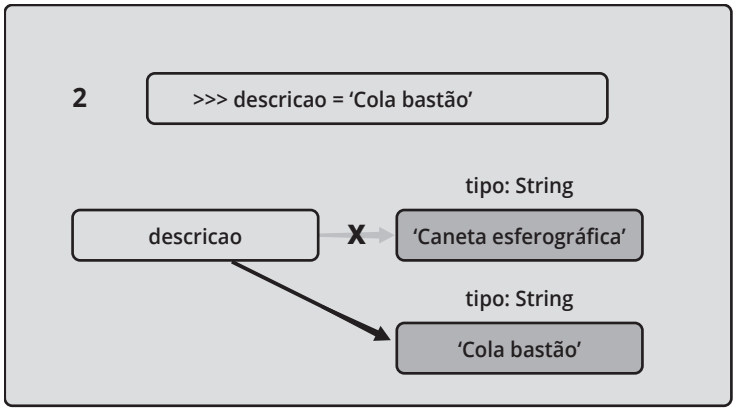

Figura 3.6
Nova atribuição de string.

Já os objetos de tipos mutáveis, como listas, por exemplo, possuem métodos que permitem alterar seu conteúdo. A figura 3.7 ilustra a inicialização de uma lista vazia e a adição de um elemento usando o método append.

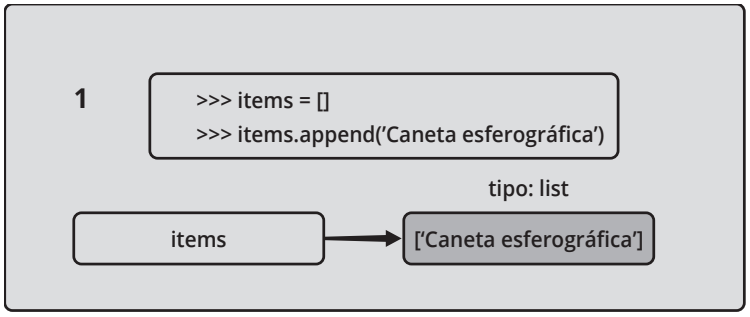

Figura 3.7
Inicialização de uma lista.

Sempre que usarmos métodos como o append, atuaremos sobre a própria lista, e o resultado dessas operações pode ser visto na figura 3.8.

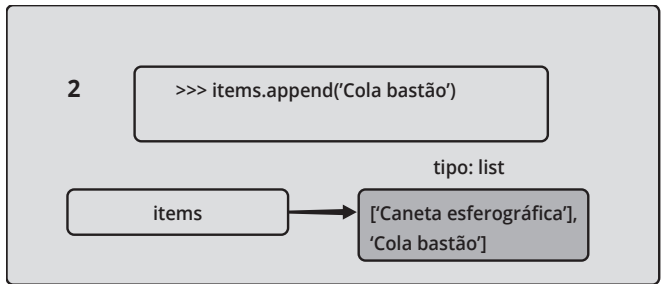

Figura 3.8
Alteração do conteúdo de uma lista.

No exemplo a seguir, vemos o que acontece quando usamos atributos de classe com tipos mutáveis.

```
#!/usr/bin/env python3
# modelo07.py

class Receita:
    etapas = []
    def __init__(self, etapas=None):
        self.etapas.extend(etapas)

class Alimento(Receita):
    def __init__(self, etapas=None):
        super().__init__(etapas)

class Remedio(Receita):
    def __init__(self, etapas=None):
        super().__init__(etapas)

if __name__ == '__main__':
    panetone = Alimento(['Assar em forno pré-aquecido'])
    melol = Remedio(['Adicionar princípio ativo'])
    print(panetone.etapas)
    # ['Assar em forno pré-aquecido',
    # 'Adicionar princípio ativo']
    print(melol.etapas)
    # ['Assar em forno pré-aquecido',
    # 'Adicionar princípio ativo']
    print(Receita.etapas)
    # ['Assar em forno pré-aquecido',
    # 'Adicionar princípio ativo']
    bolo = Alimento(['Cobrir com morangos frescos'])
    aipoglos = Remedio(['Centrifugar por 30 minutos'])
    print(Receita.etapas)
    # ['Assar em forno pré-aquecido',
    # 'Adicionar princípio ativo',
    # 'Cobrir com morangos frescos',
    # 'Centrifugar por 30 minutos']
```

O exemplo implementa a classe base “Receita” e as classes “Alimento” e “Remedio”, que herdam de receita. O importante aqui é observar o comportamento do atributo Receita. etapas. Inicialmente ele é vazio.

```
    etapas = []
```

Quando criamos o objeto do tipo Alimento, panetone, é adicionado um elemento em etapas, “Assar em forno pré-aquecido”.

```
panetone = Alimento(['Assar em forno pré-aquecido'])
```

Quando criamos o objeto do tipo Remedio, melol, é adicionado um elemento em etapas, “Adicionar princípio ativo”.

```
melol = Remedio(['Adicionar princípio ativo'])
```

Acontece que o atributo etapas é mutável e acaba sendo compartilhado entre todas as classes. Além disso, se forem criados novos objetos Alimento ou Remedio, os elementos serão adicionados na lista “etapas” e ficarão repetidos. No exemplo, vemos isso acontecer quando criamos os objetos bolo e aipoglos.

```
    bolo = Alimento(['Cobrir com morangos frescos'])
    aipoglos = Remedio(['Centrifugar por 30 minutos'])
```

Isso é um indício forte de que o atributo “etapas” não deveria ser um atributo de classe.

Métodos estáticos

- Fazem parte da classe, mas não alteram o estado da classe nem da instância.
- Podem ser sobrescritos em subclasses.
- Não precisam da palavra-chave self na definição.
- Economizam recursos, pois são instanciados apenas para a classe.

Métodos estáticos são um tipo especial de método que não altera o estado da Classe ou Instância. Em outras palavras, não altera nenhum atributo da Classe nem da Instância. Poderiam ser escritos como uma função no módulo, justamente por não influenciar o estado da Classe ou da Instância. A maior vantagem de se criar um método estático é a organização, já que sua definição fica próxima ao local onde é usado, facilitando a legibilidade do código.

Também ajuda a localizar a definição, pois para usar o método é necessário escrever o nome da classe primeiro (Classe.metodo_estatico()). Por fim, permite que o método seja sobrescrito em uma subclasse.

Vejamos um exemplo.

```
#!/usr/bin/env python3
# src/modelo08.py

class Aluno:
    def __init__(self, nome):
        self.nome = nome

class Disciplina:
    limite_vagas = 20
    def __init__(self, nome):
```

```
        self.nome = nome
        self.total_matriculas = 0
        self.matriculados = []

    def matricular(self, aluno):
        if self.verifica_vagas(self.total_matriculas,
                               self.limite_vagas):
            self.matriculados.append(aluno)
            self.total_matriculas += 1
        else:
            print('Não há mais vagas em: {}'.format(self.nome))

    @staticmethod
    def verifica_vagas(total_matriculas, limite_vagas):
        return limite_vagas > total_matriculas
if __name__ == '__main__':
    calculo = Disciplina('Cálculo')
    marcelino = Aluno('Marcelino')
    calculo.matricular(marcelino)
    print(calculo.total_matriculas)
    # 1
    print(Disciplina.verifica_vagas(20, 20))
    # False
```

No exemplo, o método estático foi utilizado diretamente na string doctest, passando valores para o método sem a necessidade de criar uma instância, assim como seria feito com uma função comum.

```
>>> print(Disciplina.verifica_vagas(20, 20))
False
```

Para definir o método como estático, foi utilizado um decorador padrão da linguagem, @staticmethod. Os métodos estáticos têm uma pequena vantagem sobre métodos comuns, pois não precisam da palavra-chave self na definição. Além disso, métodos são objetos que precisam ser instanciados e isso tem um preço. Não é muito caro, mas os métodos estáticos são instanciados apenas uma vez, para a Classe, e não para cada objeto instanciado da Classe. Dessa forma, recursos computacionais são economizados.

Métodos privados

- Python não possui métodos privados “reais”.
- Um método cujo nome inicia por sublinhado (“_”) deve ser tratado como não público. Detalhe de implementação, sujeito a mudanças.
- Métodos cujo nome iniciam com dois sublinhados são “escondidos” por uma mudança no nome (name mangling), evitando conflito de nomes na herança.
- _nomedaclasse__metodo

Métodos ou atributos privados são visíveis apenas pelo próprio objeto, dentro do escopo da classe. A linguagem Python não implementa métodos ou atributos privados como algumas outras linguagens. Por convenção, atributos e métodos que têm o nome iniciado por um caracter sublinhado ("_") devem ser considerados detalhe de implementação. Isso significa que o desenvolvedor pode renomear ou até apagar esse método em algum momento. Por isso não deve ser usado como um atributo ou método comum.

Quando um método ou atributo começa com dois caracteres sublinhados ("_"), ele será escondido. Essa característica é chamada de "name mangling", uma mudança de nome criada para evitar conflitos de nome na herança. É adicionado um prefixo no nome do atributo ou método, _nomedaclasse__atributo ou _nomedaclasse__metodo.

Esse conceito será explicado logo após o exemplo de métodos privados.

Um dos motivos de utilização de métodos ou atributos privados em algumas linguagens não são válidos em Python. Para criar uma validação, por exemplo, a linguagem Python permite que sejam criadas propriedades

Exemplo de métodos privados

O exemplo mostra uma das utilizações da técnica de métodos privados para proteger a implementação de um método que será sobrescrito através da herança.

```
#!/usr/bin/env python3
# src/modelo09.py

class Material:
    def __init__(self, historico):
        self.historico = []
        self.__atualiza(historico)
    def atualiza(self, historico):
        for historia in historico:
            self.historico.append(historia)
    __atualiza = atualiza

class Escritorio(Material):
    def atualiza(self, mapa):
        for item in mapa.items():
            self.historico.append(item)

if __name__ == '__main__':
    martelo = Material(['compra', 'uso', 'concerto'])
    print(martelo.historico)
    # ['compra', 'uso', 'concerto']
    cartucho = Escritorio(['compra', 'descarte'])
    hist = cartucho.historico
    print(sorted(hist) == ['compra', 'descarte'])
    # True
    cartucho.atualiza({'recarga': 5, 'nivel': 2})
    hist = cartucho.historico
    print(sorted(hist, key=str))
    # [('nivel', 2), ('recarga', 5), 'compra', 'descarte']
```

Na definição da classe Material, o método __init__ tem uma chamada para o método __atualiza(). O método __atualiza() é uma cópia do método atualiza definido na classe Material. O método atualiza() insere elementos na lista self.historico utilizando o parâmetro historico, assumindo que o parâmetro é um objeto iterável (pode ser uma lista, uma tupla, ou até um dicionário ou uma string).

É importante alertar que se forem passados um dicionário ou uma string, o comportamento pode não ser o esperado. No caso de passar um dicionário, apenas as chaves serão adicionadas. Já no caso de passar uma string, serão adicionadas as letras da string no atributo self.historico. Então, a classe Escritorio, que herda suas características de Material, implementa uma versão do método atualiza() que aceita um dicionário como parâmetro. Nesse exemplo, destacamos que o método __init__, que é herdado pela classe Escritorio da classe Material, manteve sua funcionalidade intacta, já que utiliza a cópia protegida de atualiza(). Nesse trecho do exemplo, podemos perceber que Escritorio foi chamada passando uma lista como parâmetro.

```
    cartucho = Escritorio(['compra', 'descarte'])
```

Se o método self.atualiza() fosse chamado diretamente no método __init__, um erro ocorreria ao criar um objeto da classe Escritorio passando uma lista como parâmetro.

Atividades práticas 

4
Recursos especiais do Python

objetivos

Muito da popularidade do Python deriva de recursos específicos da linguagem. Nesta sessão, são apresentados outros conceitos relacionados com atributos e métodos de classes e objetos. Discute-se também a forma como podemos organizar o código Python em módulos e como eles são utilizados e reutilizados.

conceitos

Propriedades; Getters; Setters; Atributos especiais; Métodos especiais; Iteradores; Geradores e módulos..

Propriedades

- Equivalente aos getters e setters de outras linguagens.
- Devem ser usadas apenas quando necessário.
- Não precisam ser criados apenas para ler ou alterar um valor. Python já cuida disso através de métodos especiais.
- Úteis para implementar validação, cache ou conversões.
- Permite a implementação futura, sem alterar o uso da classe.

Uma das perguntas mais comuns feitas por programadores que já têm alguma experiência com outras linguagens de programação orientadas a objetos é sobre a criação de getters e setters. Para aqueles que não conhecem, getters são métodos que retornam o valor de um atributo privado e setters são métodos que alteram o valor desse atributo privado.

Criar esses métodos praticamente torna esses atributos públicos. A única diferença é quando ocorre alguma computação ao retornar ou alterar o valor. Se o valor for apenas retornado diretamente ou alterado diretamente, na prática o atributo é acessado sem problemas, como se fosse público.

Em outras linguagens de programação, é quase que obrigatória a criação de getters e setters, sendo considerada uma boa prática criá-los junto com uma nova classe para evitar trabalho futuro. Mas isso é pelo fato de essas linguagens não terem um mecanismo que possibilite que uma mudança em atributo privado seja transparente a um programador usando uma classe.

A linguagem Python adotou uma abordagem diferente. Em vez de exigir que o programador utilize getters e setters sempre que estiver em dúvida se vai ou não precisar de getters e setters, simplesmente permite que o programador use getters e setters quando for realmente necessário. Ainda assim, permitindo que um programador usando uma classe nem perceba que isso aconteceu. Para isso, são usadas propriedades – mais conhecidas pelo nome em inglês, Python Properties. Podemos dizer que a grande vantagem obtida com o uso de Properties é que não precisamos criar código especulando sobre o futuro, quando for necessário usamos Properties, caso contrário não criamos getters e setters.

Exemplo de propriedades

No primeiro exemplo a classe é implementada sem usar propriedades. Vamos imaginar que após a primeira implementação tenhamos percebido que a classe não deve aceitar código 0 (zero) e a vida útil deve ter um valor positivo.

```
#!/usr/bin/env python3
# src/modelo01.py
class Computador():
    def __init__(self, codigo, nome, aquisicao, vida, marca):
        self.codigo = codigo
        self.nome = nome
        self.aquisicao = aquisicao
        self.vida = vida
        self.marca = marca

    def alerta_manutencao(self):
        # TODO: calcular período de manutenção.
        pass

if __name__ == '__main__':
    eicir = Computador(1, 'Eicir Um', '20/10/2013', 600, 'Eicir')
    eicir.codigo = 0
    eicir.vida = -10
    print(eicir.codigo, eicir.vida)
    # 0 -10
```

Para resolver essa situação, criamos as propriedades codigo e vida, que farão a validação no método setter. O que acontece na prática em Python é que o método setter é chamado quando realizamos uma operação de atribuição (“=”), e o getter é chamado sempre que precisamos recuperar o valor do atributo.

```
#!/usr/bin/env python3
# src/modelo02.py

class Computador():
    def __init__(self, codigo, nome, aquisicao, vida, marca):
        self._codigo = codigo
        self.nome = nome
        self.aquisicao = aquisicao
        self._vida = vida
        self.marca = marca
```

```
    @property
    def codigo(self):
        return self._codigo

    @codigo.setter
    def codigo(self, codigo):
        if codigo > 0:
            self._codigo = codigo
        else:
            raise ValueError('Código precisa ser maior que 0 (zero).')

    @property
    def vida(self):
        return self._vida

    @vida.setter
    def vida(self, vida):
        if vida < 0:
            raise ValueError('Vida útil não pode ser negativa.')
        else:
            self._vida = vida

    def alerta_manutencao(self):
        # TODO: calcular período de manutenção.
        pass

if __name__ == '__main__':
    eicir = Computador(1, 'Eicir Um', '20/10/2013', 600, 'Eicir')

    # Comentar a próxima linha para executar o resto do código.
    # Descomentar para ver a exceção.
    eicir.codigo = 0
    # Traceback (most recent call last):
    # ...
    # ValueError: Código precisa ser maior que 0 (zero).

    # Comentar a próxima linha para executar o resto do código.
    # Descomentar para ver a exceção.
    eicir.vida = -10
    # Traceback (most recent call last):
    # ...
    # ValueError: Vida útil não pode ser negativa.
    print(eicir.codigo, eicir.vida)
    # 1 600
```

Quando tentamos atribuir um valor inválido às propriedades vida e codigo, os métodos set_vida e set_codigo são executados e realizam a validação, lançando uma exceção.

Atributos especiais

- Os objetos em python possuem diversos atributos especiais.
- __doc__: atributo com o conteúdo da string de documentação.
- __name__: nome do objeto. Um módulo chamado pela linha de comando terá o nome '__main__'.
- __module__: nome do módulo de um objeto.
- __globals__: dicionário com as variáveis globais do objeto.
- __dict__: dicionário com atributos arbitrários do objeto.

A linguagem Python possui alguns métodos e atributos especiais. O nome desses atributos e métodos é iniciado e terminado com dois caracteres sublinhado. Um desses métodos já é conhecido, o __init__, método executado durante a inicialização dos objetos. Assim como o __init__, outros métodos especiais são executados em momentos específicos. Nesta sessão serão abordados alguns atributos especiais mais relevantes.

Um dos atributos especiais mais importantes é o atributo que armazena a string de documentação de um objeto, seja ele um módulo, classe, função, método ou qualquer outro objeto. O nome é __doc__ e é usado pela própria linguagem, editores de código ou ambientes de desenvolvimento. Raramente precisamos utilizá-lo em nossos programas, a não ser que estejamos criando um editor de código, ambiente de desenvolvimento ou um programa semelhante.

Existe um atributo interessante que é o __main__ e que armazena o nome do objeto. Normalmente utilizamos esse atributo para criar módulos. Ao importar um módulo utilizando o comando import, todos os comandos do módulo são interpretados, inclusive qualquer chamada de função acaba sendo executada. Mas isso não é interessante quando estamos utilizando o módulo em outro programa. Quando executamos um módulo, por exemplo: "python exemplo10.py", o atributo __main__ possui um valor especial, "__main__". Isso nos permite diferenciar se o módulo está sendo importado ou se está sendo executado. Para tanto, basta criarmos um bloco if com as instruções que não devem ser executadas ao importar o módulo.

Há um atributo especial que mostra o nome de um módulo e que se chama __module__. O atributo __dict__ armazena o espaço de nomes ("namespace") de um objeto – armazena os atributos que podem ser alterados ou escritos ("writeable").

Entre outros atributos, vale ainda mencionar o atributo __globals__, que armazena todos os objetos globais. Ele é uma referência para o espaço de nomes ("namespace") do módulo. Normalmente não é utilizado diretamente, assim como o __doc__, sendo mais usado por desenvolvedores de editores de código ou ambientes de desenvolvimento. É interessante mencioná-lo, pois ajuda a entender o funcionamento da linguagem. É um dos lugares onde o interpretador procurará um nome de variável ou objeto.

Exemplo de atributos especiais

Não é muito comum utilizar atributos especiais em programas comuns, já que normalmente não há necessidade de utilizá-los diretamente (sua existência é aproveitada automaticamente). Mesmo assim, é importante saber que eles existem e o exemplo a seguir, modelo03.py, mostra exatamente isso.

```
#!/usr/bin/env python3
# src/modelo03.py

variavel_global = 25

def soma(a, b=5):
    """
    Soma dois valores.
    >>> soma(2, 2)
    29
    """
    b = b + variavel_global
    return a + b

if __name__ == '__main__':
    print(soma.__code__)
    # <code object soma at 0x...>
    print(soma.__defaults__)
    # (5,)
    soma.dicionario = 'Irei para o __dict__.'
    print(soma.__dict__)
    # {'dicionario': 'Irei para o __dict__.'}
    print(soma.__doc__)
    #
    #    Soma dois valores.
    #
    #    >>> soma(2, 2)
    #    29
    print('variavel_global' in soma.__globals__)
    # True
    print(soma.__module__ == __name__)
    # True
    print(soma.__name__)
    # 'soma'
```

No exemplo, criamos um atributo para o objeto função chamado dicionário apenas para mostrar como ele é armazenado no atributo __dict__ do objeto função.

Ao acessar o atributo __doc__ no interpretador, é exibida a string como ela está armazenada. Para exibir a string, podemos usar a função print(), que vai imprimir as quebras de linha. Mas a melhor maneira mesmo é utilizar a função help(soma), que exibirá a string de documentação usando a ajuda padrão da linguagem. Além disso, ferramentas de geração de documentação utilizam esse atributo para gerar a documentação de código da linguagem Python. Se a função help() for utilizada sem parâmetro, abrirá a interface de ajuda padrão da linguagem que permite obter ajuda sobre todos os módulos, classes e métodos da biblioteca padrão.

Voltando ao exemplo, em seguida é realizada uma verificação para a "variavel_global", definida no espaço de nomes do módulo. Utilizamos o atributo __globals__ da função soma e, como esse atributo é um dicionário, utilizamos o operador in.

Se for necessário, também é possível acessar a documentação oficial através do endereço http://www.python.org/doc

Outra verificação realizada é para ver se o nome do módulo onde a função soma foi definida é o mesmo nome do módulo atual. Para isso foi utilizado o atributo __module__ da função e o atributo __name__ do módulo. No caso o resultado é verdadeiro.

Por fim, é exibido o nome da função soma, que é obviamente soma, usando o atributo __name__ da função para isso.

Métodos especiais

A linguagem Python também possui métodos especiais que seguem o padrão de nomenclatura dos atributos especiais. Dois caracteres sublinhado no início e mais dois no fim do nome do método. Esses métodos permitem uma flexibilidade muito grande para customização das classes. Alguns métodos permitem que sejam implementados comportamentos semelhantes a strings, inteiros ou booleanos, por exemplo. Esses comportamentos são obtidos através da sobrescrita de métodos especiais.

- Métodos especiais permitem a sobrescrita de operações envolvendo objetos em Python.
- **__new__:** chamado para criar uma nova instância da classe.
- **__init__:** chamado quando uma instância é criada.
- **__repr__:** chamada para obter uma representação "formal" do objeto.
- **__str__:** chamada para obter uma representação "informal" do objeto.
- Entre muitos outros, estão também métodos usados em comparações.

Os métodos especiais são executados pela linguagem em momentos específicos. Ao sobrescrevê-los a implementação customizada é que é executada. Um dos primeiros métodos a serem executados ao criar uma instância de uma classe é o __new__. Em seguida, o método __init__ é executado e frequentemente sobrescrevemos o método __init__. Raramente precisamos sobrescrever o método __new__.

Outros dois métodos frequentemente sobrescritos são o __repr__, que é chamado para obter uma representação formal do objeto. A representação formal seria uma expressão que permite recriar o objeto com o mesmo valor. Se isso não for possível, uma string na forma <... descrição útil do objeto...> deve ser retornada. O outro é o método __str__, que é executado quando é necessário obter uma representação informal do objeto. A representação informal é aquela que aparece quando usamos a função print() passando o nome do objeto como parâmetro.

Muitos outros métodos especiais estão disponíveis e são chamados quando dois objetos são comparados, quando são utilizados operadores de adição, subtração, divisão, multiplicação, enfim, em uma gama grande de operações que podem ser sobrescritos para customizar o comportamento de objetos.

Exemplo de métodos especiais

No exemplo a seguir, modelo04.py, são apresentados apenas alguns métodos especiais para mostrar a flexibilidade da linguagem. O modelo04.py cria uma classe que aceita strings contendo números inteiros válidos. Implementa operações para tratar a string como um inteiro tanto em comparações como em aritmética.

```
#!/usr/bin/env python3
# src/modelo04.py
class StringInteiro(str):
    def __new__(cls, str_inteiro):
        valor = int(str_inteiro)
        return super(StringInteiro, cls).__new__(cls, str_inteiro)

    def __init__(self, str_inteiro):
        return super(StringInteiro, self).__init__()

    def __repr__(self):
        return '{}({})'.format('StringInteiro', self)

    def __str__(self):
        return self

    def __lt__(self, other):
        return int(self) < int(other)

    def __add__(self, other):
        return str(int(self) + int(other))

if __name__ == '__main__':
    string_dez = StringInteiro('10')
    string_dois = StringInteiro('2')
    print(repr(string_dez))
    # StringInteiro(10)
    teste = eval(repr(string_dez))
    print(teste)
    # 10
    print(string_dez)
    # 10
    print(string_dez < string_dois)
    # False
    print('10' < '2')
    # True
    print(10 < 2)
    # False
    print(string_dez + string_dois)
    # 12
```

São criadas duas StringInteiro e em seguida verificamos que a representação da StringInteiro funciona utilizando a função nativa da linguagem Python eval. Essa função recebe uma expressão válida em Python e interpreta seu conteúdo retornando o resultado da expressão.

```
string_dez = StringInteiro('10')
string_dois = StringInteiro('2')
print(repr(string_dez))
# StringInteiro(10)
teste = eval(repr(string_dez))
print(teste)
# 10
print(string_dez)
# 10
```

A representação formal do objeto é StringInteiro(10). Essa expressão é válida e pode ser passada para a função eval. Assim, criamos o objeto teste, que possui o mesmo valor. Quando imprimimos os objetos teste e string_dez, temos a representação informal, que é o número inteiro 10. Esse comportamento foi implementado nos métodos __repr__ e __str__.

```
    def __repr__(self):
        return '{}({})'.format('StringInteiro', self)

    def __str__(self):
        return self
```

Em seguida, são realizadas diversas comparações. No exemplo fica clara a diferença de comportamento entre a StringInteiro e a string padrão da linguagem Python.

```
print(string_dez < string_dois)
# False
print('10' < '2')
# True
print(10 < 2)
# False
```

Enquanto a comparação da string_dez e da string_dois é avaliada como falsa, a comparação entre '10' e '2' é avaliada como verdadeira. E como era esperado, a comparação entre o inteiro 10 e o inteiro 2 é avaliada como falsa, o mesmo comportamento da StringInteiro. Esse comportamento é implementado no método __lt__ (abreviação) – do nome em inglês do sinal <, que é "less than", ou seja, lt.

```
    def __lt__(self, other):
        return int(self) < int(other)
```

O último comando da string de doctest mostra a operação matemática de soma. Ao utilizar a operação de adição com strings padrão, o que ocorreria seria a concatenação das strings. Já a StringInteiro se comporta como números inteiros.

```
print(string_dez + string_dois)
# 12
```

Iteradores

- A maioria dos conteiners em Python podem ser usados em laços for.
- O for cria um iterador, usando a função iter().
- O iterador define o método __next__(), que retorna um elemento por vez.
- Ao final, uma exceção StopIteration indica que o for deve terminar.

Os tipos de dados que podem de certa forma armazenar conteúdo em Python são chamados de containers. Alguns exemplos dessas estruturas são listas, dicionários, tuplas e strings. Esses tipos podem ter seus elementos percorridos através de um laço for.

O termo mais comumente usado é "iterar sobre", ou seja, laço for itera sobre uma lista. O que acontece na prática é que a função iter() é chamada passando um desses objetos como parâmetro. Isso é feito pelo laço for, criando um objeto chamado iterador. A principal característica dos iteradores é que eles implementam o método especial __next__. Esse método retorna um elemento por vez e, quando não existem mais elementos a serem retornados, uma exceção StopIteration indica que o laço for deve ser encerrado. Deve também implementar um método __iter__, que retorna o objeto iterador.

Quando classes são criadas, para que possam ser utilizadas em laços for é necessário implementar métodos __iter__ e __next__. Quando listas são usadas, por exemplo, elas já têm esses métodos implementados e por isso podem ser usadas diretamente. Os exemplos a seguir mostram como os iteradores funcionam e como podem ser implementados.

Exemplo de iteradores

Nesse primeiro exemplo, é apresentada a forma como o iterador funciona nos bastidores de um laço for. Para demostrar isso, é criado um objeto iterador a partir de uma lista. Em seguida, os elementos da lista são obtidos um a um até que a exceção ocorra.

```
#!/usr/bin/env python3
# src/modelo05.py

if __name__ == '__main__':
    sistemas = ['linux', 'windows', 'mac os']
    sistemas_iterador = iter(sistemas)
    sistema = next(sistemas_iterador)
    print(sistema)
    # linux
    sistema = next(sistemas_iterador)
    print(sistema)
    # windows
    sistema = next(sistemas_iterador)
    print(sistema)
    # mac os
    sistema = next(sistemas_iterador)
    # Traceback (most recent call last):
    #    ...
    # StopIteration
```

O objeto iterador é criado passando a lista como parâmetro para a função iter().

```
sistemas_iterador = iter(sistemas)
```

Nesse momento é que o método __iter__ da lista é executado. Então, para obter os elementos da lista, é executada a função next(), só que agora o objeto iterador criado é o parâmetro da função.

```
sistema = next(sistemas_iterador)
```

Quando essa instrução é executada, o método __next__ da lista é executado. Podemos repetir essa instrução quantas vezes forem necessárias, até que o último elemento seja retornado. A próxima chamada da função next() gerará uma exceção indicando que acabaram os elementos. No caso do laço for, essa exceção é tratada e ele apenas encerra o laço.

No exemplo a seguir, temos a implementação dos métodos __iter__ e __next__ para a classe Dobra.

```
#!/usr/bin/env python3
# src/modelo06.py

class Dobra:
    def __init__(self, numeros):
        self.indice = 0
        self.numeros = numeros
        self.tamanho = len(self.numeros)

    def __iter__(self):
        return self

    def __next__(self):
            if self.indice == self.tamanho:
                raise StopIteration
            proximo = self.numeros[self.indice] * 2
            self.indice += 1
            return proximo


if __name__ == '__main__':
    dobra = Dobra([1,2,3,4])
    for numero in dobra:
        print(numero)
    # 2
    # 4
    # 6
    # 8
    iterador = Dobra([3,6,9])
    next(iterador)
    # 6
    next(iterador)
    # 12
```

```
next(iterador)
# 18
next(iterador)
# Traceback (most recent call last):
#     ...
# StopIteration
```

A classe Dobra implementa um iterador. Para controlar a execução do iterador, são inicializados alguns atributos no método __init__.

```
def __init__(self, numeros):
    self.indice = 0
    self.numeros = numeros
    self.tamanho = len(self.numeros)
```

O índice servirá para indicar a posição do elemento que deve ser retornado. O atributo numeros é uma referência à lista de números que possui os elementos. E o tamanho armazena o tamanho da lista.

```
def __iter__(self):
    return self
```

O método __iter__ retorna a referência do próprio objeto, self.

```
def __next__(self):
        if self.indice == self.tamanho:
            raise StopIteration
        proximo = self.numeros[self.indice] * 2
        self.indice += 1
        return proximo
```

O método __next__ implementa o funcionamento do iterador. Quando o índice atingir o fim da lista, é gerada a exceção StopIteration, interrompendo a execução – e faz o cálculo do dobro do número, incrementa o índice e retorna o resultado do cálculo. O if principal mostra um exemplo do uso do iterador Dobra.

Atividades práticas 

Geradores

- Implementa iteradores de forma mais fácil.
- São criados como funções, mas usam yield em vez de return.
- Métodos __iter__ e __next__ são criados automaticamente.
- O estado de execução e valores das variáveis locais são controlados automaticamente.

Existe uma maneira ainda mais simples para criar um iterador: são os geradores ou “generators” (que é o termo original em inglês). Na verdade, um generator é uma função que gera uma sequência de valores.

Para criar um generator, basta declarar uma função e retornar elementos usando a palavra-chave yield (em vez de utilizar a palavra-chave return). Ou seja, ao utilizar a palavra-chave yield, a função se torna um generator e seu estado de execução é mantido, retornando um elemento de cada vez, até que não existam elementos a retornar. Na verdade, os métodos __iter__ e __next__ são criados automaticamente, assim como o estado de execução e valores das variáveis locais, que são também automaticamente controlados.

O exemplo apresentado a seguir, modelo07.py, implementa uma nova versão de dobra, que em modelo06.py era uma classe com dois métodos e alguns atributos para controlar o estado. Agora é criada uma função que obtém os mesmos resultados do exemplo anterior, escrevendo menos linhas de código.

```
#!/usr/bin/env python3
# src/modelo07.py

def dobra(numeros):
    for numero in numeros:
        yield numero * 2


if __name__ == '__main__':
    # Implementa a função geradora dobra que multiplica o valor dos elementos
    # por dois.

    for numero in dobra([1, 2, 3, 4]):
        print(numero)
    # 2
    # 4
    # 6
    # 8
    gerador = dobra([3,6,9])
    print(next(gerador))
    # 6
    print(next(gerador))
    # 12
    print(next(gerador))
    # 18
    print(next(gerador))
    # Traceback (most recent call last):
    #     ...
    # StopIteration
```

Tudo ficou mais simples. Três linhas de código para ter o mesmo resultado.

```
def dobra(numeros):
    for numero in numeros:
        yield numero * 2
```

Cada vez que o yield é chamado, o valor calculado é retornado, assim como ocorre com o return. A diferença é que o estado de execução fica salvo e quando a função for chamada novamente, o cálculo é realizado e outro elemento é retornado.

```
>>> for numero in dobra([1, 2, 3, 4]):
...    print(numero)
```

Ao final, quando não houver mais elementos, a exceção StopIteration é lançada e o laço for é encerrado.

Expressões com geradores

- Geradores simples podem ser escritos com sintaxe similar à compreensão de listas.
- Usam-se parênteses em vez de colchetes.
- Normalmente usadas como parâmetros de funções.
- São mais simples que geradores.
- Porém, são menos flexíveis.
- Mais amigáveis ao uso de memória do que compreensão de listas equivalentes.

As expressões geradoras (“generator expression”) são escritas de forma similar à compreensão de listas vista na sessão 2. A diferença é que utilizam parênteses em vez de colchetes. O uso mais comum é utilizá-las como parâmetros de função. São mais simples e compactas do que os geradores (“generators”), mas são menos versáteis e menos flexíveis. São mais amigáveis ao uso de memória do que compreensão de listas equivalentes.

Seque um exemplo de expressão geradora.

Exemplo de expressão com geradores

O exemplo implementa uma classe Aluno para auxiliar a demonstração. O if principal no fim do programa mostra o uso de expressões geradoras.

```
#!/usr/bin/env python3
# src/modelo08.py

class Aluno:
    def __init__(self, codigo=0, nome='', media=0):
        self.codigo = codigo
        self.nome = nome
        self.media = media

    def situacao(self):
        if self.media < 4:
            return 'Reprovado'
        elif self.media < 7:
            return 'Recuperação'
        else:
            return 'Aprovado'

if __name__ == '__main__':
    # Implementa a classe exemplo alunos para ser usada como exemplo
    # de uso de expressões de geradores. Além das expressões que criam
    # os geradores codigos e situacoes, a função zip é um gerador também.
```

```
    alunos = [Aluno(1, 'Pedro', 8),
              Aluno(2, 'Laura', 9),
              Aluno(3, 'Estudapoco', 5),
              Aluno(4, 'Mataula', 3)]
    codigos = (aluno.codigo for aluno in alunos)
    print(type(codigos))
    # <class 'generator'>
    situacoes = (aluno.situacao() for aluno in alunos)
    print(type(situacoes))
    # <class 'generator'>
    resultados = {cod: sit for cod, sit in zip(codigos, situacoes)}
    print(resultados)
    # {1: 'Aprovado', 2: 'Aprovado', 3: 'Recuperação', 4: 'Reprovado'}
```

É criada uma lista de alunos que será usada nos exemplos de expressões geradoras. Então, são criados dois geradores utilizando expressões geradoras.

```
codigos = (aluno.codigo for aluno in alunos)
print(type(codigos))
<class 'generator'>
situacoes = (aluno.situacao() for aluno in alunos)
print(type(situacoes))
<class 'generator'>
```

Para confirmar que foram criados geradores, foi impresso o tipo das variáveis codigos e situacoes usando a função type. Elas são do tipo “class generator”.

```
resultados = {codigo: situacao
    for codigo, situacao in zip(codigos, situacoes)}
print(resultados)
{1: 'Aprovado', 2: 'Aprovado', 3: 'Recuperação', 4: 'Reprovado'}
```

Por fim, criamos um dicionário que utiliza os códigos como chave, associados a situação. O dicionário é criado utilizando uma expressão geradora, essa sintaxe é chamada também de compreensão de dicionários.

Uma dúvida sempre surge: quando usar compreensão de listas ou expressões geradoras? Uma das maneiras mais fáceis de decidir é pensar se uma lista será necessária. Após obter o resultado, será necessário usar um método de lista (algo como lista.append ou lista.insert)? Será necessário acessar elementos pelo índice ou usar fatiamento de listas? Enfim, se for necessário ter uma lista, deve ser usada uma compreensão de listas.

Módulos

- São arquivos Python que contêm definições e comandos.
- Facilitam a reutilização e organização de código.
- O nome do arquivo é o nome do módulo, mais o sufixo “.py”
- O nome de um módulo pode ser acessado pela variável global __name__.

Os módulos da linguagem Python são arquivos com a extensão “.py”, que podem conter instruções, variáveis, comandos, funções ou classes. Para manter uma boa organização, recomenda-se que em um módulo fiquem definidas coisas similares dentro de um contexto.

Deixar todo o código de um sistema grande em um único módulo dificulta a manutenção e reutilização do código.

Existe uma variável global muito utilizada em módulos, chamada __name__, que armazena o nome do módulo. Um módulo da linguagem Python é um objeto e, portanto, possui propriedades.

Vejamos como isso acontece na prática. Vamos implementar um módulo simples que imprima a variável __name__.

```
# exercicio01.py
    print(__name__)
```

Tendo feito isto, execute o mesmo conforme o comando:

```
$ python exercicio01.py
```

Em seguida, abra o interpretador e execute o comando import, conforme indicado a seguir:

```
$ python
>>> import exercicio01
```

Qual a diferença entre as duas execuções do mesmo módulo?

Sempre que um módulo é importado ou quando é passado como parâmetro na linha de comando do interpretador, ele é interpretado. Todas as instruções do módulo são interpretadas, ou seja, são realizadas as análises léxica e sintática, é gerado o bytecode, que é então executado pela máquina virtual da linguagem Python.

Isso tudo linha a linha, instrução por instrução. Mas nem sempre é necessário executar tudo, já que em módulos com muitas classes ou funções podemos estar interessados em usar apenas uma função ou classe específica. Nesse caso, é possível usar uma técnica específica para que a parte principal de um módulo só seja executada quando o módulo for chamado por linha de comando.

- Para rodar um módulo, usamos o comando: python nome_do_modulo.py <argumentos>.
- Quando executado dessa forma, o nome do módulo será __main__, ou seja, a variável __name__ será __main__.
- Para controlar o que é executado quando o módulo é importado, é verificado se a variável __name__ é igual a __main__.
- Quando o módulo for importado por outro, a variável __name__ será o nome do módulo e o bloco do teste não será executado.

Como vimos acima, para rodar um módulo pela linha de comando basta passar o nome do módulo como parâmetro da linha de comando para o interpretador:

```
python nome_do_modulo.py <argumentos do módulo>
```

Na linha de comando acima, os <argumentos do módulo> são opcionais. Ao executar um módulo dessa maneira, o nome dele, internamente, será __main__. Esse nome fica armazenado na variável global __name__. Então, para evitar que uma parte do código seja executada, criamos uma condição que verifica se essa variável, __name__, é igual a __main__.

Quando o módulo for importado, a variável __name__ conterá o nome do módulo, que no exemplo anterior era “nome_do_modulo”.

Esses conceitos podem ser melhor explorados no exemplo a seguir, modelo09.py, onde temos um módulo que define duas funções que verificam se um número é primo usando dois algoritmos diferentes: primo1 e primo2. Logo no início temos a definição da exceção PrimoInvalido. Assim, podemos dizer que esse módulo tem três símbolos, já que cada variável, classe ou função é considerado um símbolo.

```
#!/usr/bin/env python3
# src/modelo09.py

class PrimoInvalido(Exception):
    pass

def primo1(numero):
    if numero <= 0:
        raise PrimoInvalido
    # Número um não é primo.
    if numero == 1:
        return False
    # Número 2 é o único par primo.
    if numero == 2:
        return True
    for valor in range(2, numero, 1):
        if (numero % valor) == 0:
            return False
    return True

def primo2(numero):
    if numero == 0:
        raise PrimoInvalido
    # Número um não é primo.
    if numero == 1:
        return False
    # Número 2 é o único par primo.
    if numero == 2:
        return True
    # Os outros números pares não são primos.
    if (numero % 2) == 0:
        return False
    for valor in range(3, numero // 2, 2):
        if (numero % valor) == 0:
            return False
    return True

if __name__ == '__main__':
    print(primo1(0))
    # Traceback (most recent call last):
    #  ...
    # modelo09.PrimoInvalido
    print(primo1(1))
    # False
    print(primo1(2))
    # True
    print(primo1(4))
    # False
    print(primo1(5))
```

```
    # True
    print(primo2(0))
    # Traceback (most recent call last):
    #  ...
    # modelo09.PrimoInvalido
    print(primo2(1))
    # False
    print(primo2(2))
    # True
    print(primo2(4))
    # False
    print(primo2(5))
    # True
```

O último comando if do programa utiliza a variável global __name__ para diferenciar o modo de execução do módulo. Os comandos dentro do bloco if são executados somente quando o módulo é executado diretamente pela linha de comando ("python modelo09.py"). Entretanto, se importarmos o módulo exemplo09.py em outro módulo qualquer, esse trecho do if não será executado (a variável __name__ será '__modelo09__').

Vale reforçar que as instruções/comandos de um programa Python são interpretados apenas uma vez, no momento da sua execução ou importação. As definições de um módulo fazem parte da sua própria tabela de símbolos. É importante ter isso em mente, pois módulos podem importar outros módulos, permitindo assim o reaproveitamento de código.

- As definições e comandos são interpretados apenas uma vez, quando o módulo é executado ou importado.
- Cada módulo tem sua própria tabela de símbolos.
- Módulos podem importar outros módulos.
- Módulos importados são alocados na tabela de símbolos do módulo onde foram importados.

Quando um ou mais módulos são importados, eles são alocados na tabela de símbolos do módulo principal, que importou os outros módulos. Existem várias formas de importar um módulo, conforme veremos mais adiante.

```
#!/usr/bin/env python3
# src/modelo10.py

import modelo09
from modelo09 import primo2

print(modelo09.primo1(13))
print(primo2(17))

if __name__ == '__main__':
    print(modelo09.primo1(0))
    # Traceback (most recent call last):
    # ...
    # modelo09.PrimoInvalido
```

```
print(modelo09.primo1(1))
# False
print(primo2(2))
# True
print(primo2(4))
# False
```

Para importar o módulo, usamos o comando import modulo. Quando o módulo é importado desse jeito, para acessar algum elemento usamos o nome do módulo e o nome do elemento, separado por um ponto.

```
modelo09.primo1(13)
```

Quando é importado um elemento, podemos usar o nome do elemento diretamente. Só é preciso tomar cuidado para não importar elementos que tenham o nome igual, pois isso vai provocar um conflito silencioso (o último elemento importado sobrescreve o anterior).

```
print(primo2(17))
```

Importando vários elementos

- É possível importar mais elementos na mesma linha.
- Também podem ser importados todos os elementos de uma só vez.
- Entretanto, é recomendado importar apenas o que é necessário. Um elemento por linha de preferência.
- Quando usar o interpretador em modo interativo, essas recomendações podem ser ignoradas.

Vários elementos de um mesmo módulo podem ser importados em uma única linha, basta separar os nomes com vírgulas.

```
>>> from modelo09 import primo1, primo2
```

Para importar todos os elementos de um módulo não há necessidade de listar um por um, basta usar o carácter curinga asterisco (“*”).

```
>>> from modelo09 import *
```

No primeiro caso, apenas as funções primo1 e primo2 são importadas para o espaço de nomes do módulo que fez a importação. Já no segundo, tudo o que foi declarado em modelo09.py será importado, não ficando claro quais são os elementos que serão de fato utilizados.

Módulos são a base para a melhor organização de um sistema, permitindo que o código seja dividido por diferentes arquivos para melhorar a documentação e manutenção do sistema. Na verdade, o Python já vem com uma biblioteca considerável de módulos padrão. Módulos podem ser também organizados em pacotes. Vamos explorar isso melhor na próxima sessão.

Mas antes disso é hora de colocar em prática o que já foi abordado até aqui.

Atividades práticas AVA

5
Bibliotecas do Python

objetivos

Permitir que o aluno comece a conhecer o universo de módulos e bibliotecas já disponíveis na linguagem, permitindo que programas e sistemas sejam desenvolvidos de forma mais rápida e eficiente.

conceitos

Módulos Nativos; Pacotes; Serialização; JSON; Acesso à internet e bancos de dados..

Baterias inclusas

- Python já vem com diversas bibliotecas (módulos e pacotes) em sua distribuição padrão, dando origem à frase "Python vem com baterias (pilhas) inclusas".
- Além das bibliotecas padrão, Python possui uma gama enorme de bibliotecas de terceiros.
- As bibliotecas de terceiros podem ser encontradas no índice de pacotes python, pypi: https://pypi.python.org.

A linguagem Python possui diversos módulos já prontos, com inúmeras funcionalidades disponíveis. Como veremos a seguir, os módulos podem ser organizados em pacotes e formam um conjunto de bibliotecas muito úteis. Daí se dizer que Python já vem com as baterias (pilhas) inclusas, fazendo analogia aos brinquedos movidos a pilha que vinham com o aviso "baterias não incluídas". A criança abria o presente, mas não conseguia começar a brincar com ele porque era necessário comprar as pilhas, se sentindo obviamente frustrada.
No caso da linguagem Python, ela já está pronta para começar a brincar!

Como muitas pessoas e empresas que utilizam a linguagem acreditam na filosofia de colaboração e compartilhamento, muitas bibliotecas de terceiros estão disponíveis publicamente. Um dos melhores lugares para procurar bibliotecas de terceiros é o índice de pacotes Python. Do termo em inglês "Python Package Index", temos a sigla pypi e o endereço é https://pypi.python.org.

Encontrando módulos e bibliotecas

- Ao importar um módulo usando o comando import, o interpretador inicia uma procura.
- Começa a busca verificando a lista de módulos nativos.
  Ver: https://docs.python.org/3/library/
- Prossegue buscando na lista de diretórios em sys.path.
- Se nada for encontrado, ocorre uma exceção.

Como já foi visto na sessão anterior, é preciso importar as funcionalidades que estão definidas externamente para que possam ser utilizadas em um programa ou módulo qualquer sendo desenvolvido.

Para que tudo funcione a contento, é importante entender como o interpretador Python funciona ao encontrar um comando import. O primeiro passo será procurar o arquivo com o nome indicado no comando. Essa busca segue etapas pré-determinadas, começando por uma busca na lista de módulos nativos da linguagem. Se o módulo não for encontrado, a busca continua pela lista de diretórios armazenada em sys.path. A propósito, o módulo sys está sempre disponível e permite o acesso a variáveis usadas ou mantidas pelo interpretador Pyhton.

A lista de módulos nativos pode ser encontrada na documentação oficial: https://docs.python.org/3/library/

O código apresentado em modelo01.py a seguir imprime de uma maneira amigável o conteúdo da lista sys.path.

```
#!/usr/bin/env python3
# src/modelo01.py

import sys
from pprint import pprint as pp
pp(sys.path)
```

Consultando sys.path

Logo a seguir são apresentadas duas formas distintas de execução do modelo01.py. Na primeira, o módulo é executado diretamente pela linha de comando, enquanto que na segunda o módulo é importado de dentro do interpretador.

Primeira forma:

```
$ python3 modelo01.py
['/home/.../src',
 '/usr/lib/python3.4',
 ...
 '/usr/lib/python3/dist-packages']
```

Segunda forma:

```
$ python3
Python 3.4.2 (default, Oct 8 2014, 10:45:20)
[GCC 4.9.1] on linux
Type "help", "copyright", "credits" or "license" for ...
>>> import modelo01
['',
 '/usr/lib/python3.4',
 ...
 '/usr/lib/python3/dist-packages']
>>>
```

A grande diferença entre as duas execuções é que na primeira a lista sys.path tem como elemento inicial o diretório "/home/.../src", que é o local onde está o próprio módulo sendo executado. Já na segunda forma de execução, de dentro do interpretador, o primeiro elemento da lista sys.path é uma string vazia. Isso acontece porque nenhum script foi passado para o interpretador. O efeito na prática é quase o mesmo, pois no segundo caso são procurados módulos no diretório atual, que é o mesmo que consta na lista sys.path da primeira execução.

A função dir

Função nativa dir:

- Exibe os nomes definidos em um módulo bem como métodos e atributos de uma classe/objeto.
- Sem argumentos, retorna os nomes já definidos.
- Não mostra variáveis e funções nativas por padrão. Para isso, precisa importar o módulo builtins.

Quando um elemento é importado de outro módulo, ele se torna parte da tabela de símbolos atual. Uma das formas de consultar essa tabela de símbolos é usando a função nativa dir. Essa função exibe os nomes de todos os elementos definidos no módulo e os nomes de todos os elementos importados de outros módulos. A função dir também pode ser usada para listar os métodos e propriedades de um objeto, bastando passar o objeto como parâmetro (p.e. dir(list)).

A função dir normalmente não mostra as variáveis e funções nativas (até porque a lista pode ser muito grande), mas havendo necessidade, basta importar o módulo builtins e rodar novamente a função passando o módulo como parâmetro.

O modelo02.py a seguir mostra exemplos da chamada da função dir.

```
#!/usr/bin/env python3
# src/modelo02.py

def main():
    print(dir())
    # ['__builtins__', '__cached__', '__doc__', ... '__spec__']
    import zipfile
    print(dir(zipfile))
    # ['BZIP2_VERSION', 'BadZipFile', 'BadZipfile', ..., 'zlib']
    import builtins
    print(dir(builtins))
    # ['ArithmeticError', 'AssertionError', ..., 'vars', 'zip']
    meu_sapato = 'Havaianas'
    print(dir(meu_sapato))
    # ['__add__', '__class__', ... 'translate', 'upper', 'zfill']
    dir()
    # ['__builtins__', ... 'builtins', 'meu_sapato', 'zipfile']

if __name__ == '__main__':
    main()
```

Na primeira chamada da função dir, nada foi declarado ou importado, então são exibidas apenas algumas variáveis globais.

```
>>> print(dir())
['__builtins__', '__cached__', '__doc__',... '__spec__']
```

Em seguida, o módulo zipfile é importado e a função é chamada, passando o módulo como parâmetro. Dessa forma, são listados os elementos definidos no módulo.

```
>>> import zipfile
>>> print(dir(zipfile))
['BZIP2_VERSION', 'BadZipFile', 'BadZipfile',..., 'zlib']
```

De forma análoga é importado o módulo builtins – e seus elementos são apresentados.

```
>>> import builtins
>>> print(dir(builtins))
['ArithmeticError', 'AssertionError', ..., 'vars', 'zip']
```

Agora uma variável string meu_sapato é criada e os seus elementos são exibidos da mesma forma, chamando a função dir e passando a variável como parâmetro.

```
>>> meu_sapato = 'Havaianas'
>>> dir(meu_sapato)
['__add__', '__class__', ... 'translate', 'upper', 'zfill']
```

Após todas essas definições e importações, se a função dir for executada sem parâmetros, todos os elementos definidos são listados.

```
>>> dir()
['__builtins__', ... 'builtins', 'meu_sapato', 'zipfile']
```

Normalmente, a função dir é utilizada com o interpretador no formato interativo, sendo muito útil para conhecer os métodos e atributos de algum objeto ou lembrar o nome correto de algum método.

Pacotes

- Pacotes são um mecanismo interessante para organizar módulos.
- Um pacote nada mais é do que um diretório contendo módulos ou subdiretórios.
- Esse diretório precisa de um arquivo chamado __init__.py para se tornar um pacote
- Ao importar um pacote, ele é procurado na lista de caminhos sys.path
- Normalmente não “executamos” um pacote.

A boa estruturação de um projeto pode facilitar muito a sua manutenção. A linguagem Python possui um mecanismo de organização muito interessante, que são os pacotes. Um pacote é, na verdade, uma pasta (diretório), que pode conter módulos e subpastas (subdiretórios). Fica a critério do desenvolvedor decidir qual a melhor maneira para estruturar os módulos e pastas para o seu projeto.

Para que a linguagem Python trate um diretório como pacote, é preciso que exista um arquivo chamado __init__.py nesse diretório. Esse arquivo pode ser vazio, mas se houver alguma instrução ou declaração, ela será executada quando o pacote for importado. Assim como os módulos, ao importar um pacote ele será procurado na lista sys.path.

Uma característica importante dos pacotes é que eles não são executados. Para utilizar um pacote, criamos outro script que importa o pacote e implementa a lógica necessária para utilizar as funcionalidades do pacote.

Existe um software livre chamado Cookiecutter, que utiliza alguns modelos padrão de projetos e cria a estrutura básica para projetos variados com arquivos e pastas padrão. Ele está disponível em https://github.com/audreyr/cookiecutter

Estrutura de um pacote

Um pacote tem uma estrutura hierárquica. Um diretório contendo módulos e outros diretórios:

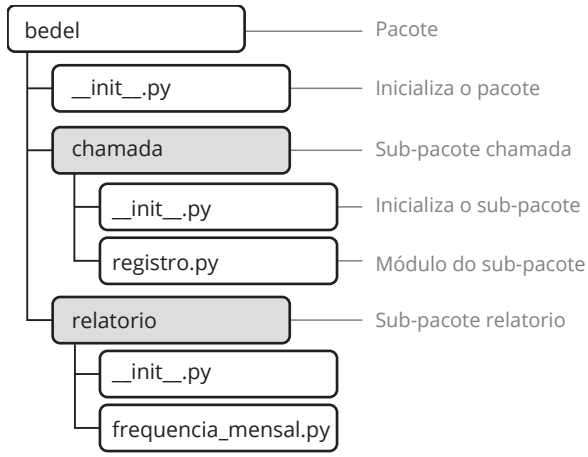

Figura 5.1 Estrutura hierárquica de um pacote.

Esse exemplo foi criado para ilustrar a estrutura de um pacote, com dois subpacotes: chamada e relatorio. O módulo do pacote relatorio consome os dados do módulo registro.py do pacote chamada. No momento, basta entender que o pacote bedel possui dois subpacotes, que por sua vez possuem módulos.

Exemplo de uso de pacote

A utilização de um pacote é similar à utilização de um módulo. Basta seguir a hierarquia de pastas utilizando o ponto (".") para indicar a estrutura correspondente.

```
#!/usr/bin/env python3
# src/modelo03.py

from bedel.relatorio import frequencia_mensal

def main():
    frequencia_mensal.imprimir()

if __name__ == '__main__':
    main()
```

Esse trecho de código acima, modelo03.py, o módulo frequencia_mensal do subpacote relatorio, pacote bedel, é importado para que se possa usar a função imprimir.

```
#!/usr/bin/env python3
# src/bedel/chamada/registro.py

import ...
```

O módulo registro.py define algumas funções que emulam a coleta de dados que serão usados nos relatórios. O ideal é que os subpacotes sejam independentes entre si. Um script ou módulo de um nível anterior aos subpacotes é que deve fazer a integração das funcionalidades. Para fins didáticos, essa independência não foi implementada aqui, sendo possível ver como funciona o uso de importação relativa entre subpacotes.

```
#!/usr/bin/env python3
# src/bedel/relatorio/frequencia_mensal.py

from ..chamada.registro import registros_de_frequencia
```

O relatório de fluxos precisa dos dados de utilização, que são obtidos através de uma função que está em outro subpacote. Por conta disso, foi utilizada a importação relativa, usando ponto-ponto (“..”) para indicar o diretório superior. Isso viabiliza encontrar o pacote chamada e seu módulo registro para enfim importar a função registros_de_frequencia.

Importar subpacotes com *

- from pacote import * não importa todos os subpacotes automaticamente.
- A variável __all__ no arquivo __init__.py define quem será importado com *.
- É recomendado evitar isso: o ideal é usar o comando from pacote import subpacote.

Também é possível importar vários elementos com o carácter coringa (“*”). Entretanto, é necessário que o desenvolvedor que criou o pacote inicialize uma variável com o nome dos subpacotes ou módulos que devem ser importados com o coringa. A variável chama-se __all__ e deve ser definida no arquivo __init__.py. É recomendado evitar o uso do coringa (“*”) em scripts. Um dos motivos é que não fica claro o que está sendo importado e, pior do que isso, pode haver conflito de nomes entre variáveis declaradas em módulos diferentes. Mas esse mecanismo pode ser útil para escrever códigos rápidos no interpretador.

Exemplo com a variável __all__

A variável __all__ é uma lista de strings com os nomes dos módulos ou subpacotes que devem ser importados quando o carácter coringa “*” for usado.

```
#!/usr/bin/env python3
# bedel/__init__.py

__all__ = ['chamada', 'relatorio']
```

A variável __all__ precisa ser definida no arquivo __init__.py do pacote. Nesse exemplo, foram definidos os subpacotes chamada e relatorio.

```
#!/usr/bin/env python3
# src/modelo03.py

from bedel.relatorio import frequencia_mensal
from bedel import *

escopo_global = dir()
atributos_do_modulo = dir(frequencia_mensal)

def main():
    frequencia_mensal.imprimir()
    # []
    print(atributos_do_modulo)
    # ['__builtins__', ..., 'imprimir', 'registros_de_frequencia']
```

```
    print(escopo_global)
    # ['__builtins__', ..., 'chamada', 'frequencia_mensal', 'relatorio']

if __name__ == '__main__':
    main()
```

Quando o caractere "*" é usado nesse exemplo, os subpacotes chamada e relatorio são importados, pois foram declarados na variável __all__ do __init__.py do pacote bedel. O módulo frequencia_mensal também consta na saída da chamada da função dir, pois foi importado explicitamente.

Lidando com arquivos

- A função open retorna um objeto arquivo.
- Entre os parâmetros aceitos estão: nome do arquivo, modo de abertura (leitura, escrita, acréscimo, binário, entre outros) e encoding (codificação usada para codificar e decodificar o arquivo, o padrão depende da plataforma).
- O objeto arquivo possui diversos métodos para leitura e escrita.
- Recomenda-se usar a palavra reservada with para lidar com arquivos.
- O arquivo é fechado automaticamente quando o with termina.

Outra função nativa do Python é a open, que retorna um objeto do tipo arquivo. Entre os parâmetros aceitos estão: nome do arquivo, modo de abertura (leitura, escrita, acréscimo, binário, entre outros) e encoding (codificação usada para codificar e decodificar o conteúdo do arquivo). Esses objetos possuem diversos métodos para ler e escrever informações.

Após o uso do arquivo, é importante fechá-lo para evitar que ele seja corrompido. É mais comum do que se imagina estar com um arquivo aberto e em uma queda de energia ter todos os dados perdidos. Uma maneira prática para evitar isso é utilizar a palavra reservada with em conjunto com a função open. Quando um bloco with termina, o arquivo é fechado automaticamente. Vejamos isso através de um exemplo prático.

O modelo04.py apresenta diversas operações com arquivos na string de documentação doctest.

```
#!/usr/bin/env python3
# src/modelo04.py

def main():
    arquivo = open('citacoes.txt')
    conteudo = arquivo.read()
    print(conteudo)
    # Algo só é impossível até que alguém duvide e resolva provar ao contrário. ...
    # Não sabendo que era impossível, ele foi lá e fez. - Jean Cocteau
    # O degrau da escada não foi inventado para repousar, mas apenas para ...
    # Daqui a alguns anos estará mais arrependido pelas coisas que não ...
    conteudo = arquivo.read()
    print(conteudo)
    #

    # seek(0) posiciona o cursor no início do arquivo.
```

```
    arquivo.seek(0)
    conteudo = arquivo.readline()
    print(conteudo)
    # Algo só é impossível até que alguém duvide ... - Albert Einstein

    arquivo.seek(0)
    autores = []
    for linha in arquivo:
        # separa a string linha usando "-" como separador.
        autor = linha.split('-')[1]
        autores.append(autor)
        print(linha, end='')
    # Algo só é impossível até que alguém duvide e resolva provar o contrário. ...
    # Não sabendo que era impossível, ele foi lá e fez. - Jean Cocteau
    # O degrau da escada não foi inventado para repousar, mas apenas para ...
    # Daqui a alguns anos estará mais arrependido pelas coisas que não ...

    # modo de escrita. (sobrescreve)
    saida = open('autores.out', 'w')
    saida.write(autores[0])
    saida.close()

    # Descomentar a próxima linha para ver a exceção.
    # saida.readlines()
    # Traceback (most recent call last):
    # ...
    # ValueError: I/O operation on closed file.

    # Modo 'a' abre para adição. (não sobrescreve).
    with open('autores.out', 'a') as saida:
        # Código do primeiro autor (índice 0) já foi escrito no arquivo.
        # Percorremos o for do segundo (índice 1) em diante.
        for autor in autores[1:]:
            saida.write(autor)
    saida.closed
    # True

if __name__ == '__main__':
    main()
```

O arquivo citacoes.txt é aberto no modo de leitura. O conteúdo é lido todo de uma só vez para a variável conteudo utilizando o método read. Tomar cuidado com o método read, pois se o arquivo for muito grande, a operação pode demorar.

```
>>> arquivo = open('citacoes.txt')
>>> conteudo = arquivo.read()
```

Após a leitura do arquivo, o cursor de leitura fica posicionado no fim do arquivo. Esse cursor é uma espécie de ponteiro que marca uma posição no arquivo. Se for feita uma nova tentativa de leitura quando o cursor está no final, nada é lido.

```
>>> conteudo = arquivo.read()
>>> print(conteudo)
<BLANKLINE>
>>> # seek(0) posiciona o cursor no início do arquivo.
>>> arquivo.seek(0)
0
>>> conteudo = arquivo.readline()
```

Para reposicionar o cursor, existe um método chamado seek. No exemplo, o cursor é reposicionado na primeira posição do arquivo e o método readline é usado para ler apenas uma linha do arquivo.

```
>>> for linha in arquivo:
```

Outra maneira de percorrer o arquivo é utilizando o comando for. A cada iteração do for, uma linha do arquivo é lida.

```
>>> saida = open('autores.out', 'w')
>>> saida.write(autores[0])
```

Para escrever informações em um arquivo, deve ser indicado que arquivo será aberto em modo de escrita ('w'). Esse modo sobrescreve o conteúdo anterior. Para gravar informações no arquivo, usamos o método write.

```
>>> saida.close()
>>> saida.readlines()
Traceback (most recent call last):
 ...
ValueError: I/O operation on closed file.
```

Se for realizada uma tentativa de leitura ou escrita em um arquivo fechado, uma exceção será disparada.

Serializando dados com JSON

- A sigla JSON significa JavaScript Object Notation.
- É usado para armazenar e transmitir informações com base no formato texto.
- Muito mais simples que XML.
- Sua sintaxe permite armazenar diversos tipos de dados.
- Python possui uma biblioteca chamada json, que permite serializar e deserializar dados no formato json.

O termo serializar é usado para o ato de salvar objetos em arquivos de uma forma que possibilite a obtenção desses objetos novamente. Ao deserializar um objeto, ele volta a ficar disponível e pode ser manipulado em um programa. Um objeto pode ser serializado usando diversos formatos, como por exemplo, em modo binário, texto ou até mesmo XML. Um dos formatos que vêm se tornando cada vez mais populares é o formato JSON. É na verdade a abreviação ou sigla derivada de "JavaScript Object Notation". Uma tradução livre para português seria Notação de Objetos em JavaScript (NOJS). Esse formato tem sido usado para armazenar e transmitir informações com base no formato texto, especialmente em serviços de internet, pois é reconhecido pela linguagem javascript.

É um formato muito mais simples que XML, e a sintaxe permite armazenar diversos tipos de dados. A linguagem Python possui uma biblioteca que permite a serialização e deserialização de objetos usando o formato JSON. A seguir um exemplo de arquivo nesse formato.

```
{'google_sp': {
     "complemento": "de 3253 ao fim - lado \u00edmpar",
     "bairro": "Itaim Bibi", "cidade": "S\u00e3o Paulo",
     "logradouro": "Avenida Brigadeiro Faria Lima",
     "estado_info": {"area_km2": "248.222,362",
     "codigo_ibge": "35", "nome": "S\u00e3o Paulo"},
     "cep": "04538133",
     "cidade_info": {"area_km2": "1521,101",
     "codigo_ibge": "3550308"}, "estado": "SP"}}
```

No exemplo a seguir, modelo05.py, é gerado o arquivo local.json. Os dados de um dicionário Python são usados para demonstrar o uso da biblioteca json.

```
#!/usr/bin/env python3
# src/modelo05.py

"""
API de CEPs.

https://viacep.com.br/ws/04538133/json/
https://viacep.com.br/ws/22220000/json/
"""

import json

ceps = {"google_sp": {
      "cep": "04538-133",
      "logradouro": "Avenida Brigadeiro Faria Lima",
      "complemento": "de 3253 ao fim - lado ímpar",
      "bairro": "Itaim Bibi",
      "localidade": "São Paulo",
      "uf": "SP",
      "ibge": "3550308",
      "gia": "1004"
    }}

msf_brasil = {
  "cep": "22220-000",
  "logradouro": "Rua do Catete",
  "complemento": "até 195/196",
  "bairro": "Catete",
  "localidade": "Rio de Janeiro",
  "uf": "RJ",
  "ibge": "3304557",
  "gia": ""
}
```

```
def serializa(ceps_json):
    with open('local.json', 'w') as arquivo:
        json.dump(ceps_json, arquivo)

def deserializa():
    with open('local.json') as arquivo:
        return json.load(arquivo)

if __name__ == '__main__':
    google_sp = json.dumps(ceps['google_sp'])
    print(google_sp)
    # '{...}'
    serializa(ceps)
    novo_ceps = deserializa()
    print('msf_brasil' in novo_ceps)
    # False
    ceps['msf_brasil'] = msf_brasil
    serializa(ceps)
    novo_ceps = deserializa()
    print(novo_ceps['msf_brasil']['logradouro'])
    # Rua do Catete
    print(novo_ceps['msf_brasil']['bairro'])
    # Catete
    print(novo_ceps['msf_brasil']['cidade'])
    # Rio de Janeiro
    print(novo_ceps['msf_brasil']['estado'])
    # RJ
    print(novo_ceps['msf_brasil']['complemento'])
    # até 195/196
```

Os dados do dicionário representam dados consultados em um serviço aberto de CEPs. O exemplo transforma os dados do dicionário em uma string usando o método json.dumps:

```
google_sp = json.dumps(ceps['google_sp'])
```

É possível reparar que o formato do json é muito similar ao formato do Python. Em seguida, o exemplo cria um novo elemento no dicionário ceps, usando um dicionário definido anteriormente, msf_brasil, com o CEP do escritório de uma ONG no Brasil e salva usando a função serializa. Lembre-se de que os dicionários podem armazenar nos seus itens qualquer objeto, inclusive outros dicionários. Existe restrição apenas para as chaves que devem ser objetos imutáveis.

```
ceps['msf_brasil'] = msf_brasil
serializa(ceps)
```

A função serializa utiliza o método json.dump para salvar o conteúdo do dicionário no arquivo.

```
def serializa(ceps_json):
    with open('local.json', 'w') as arquivo:
        json.dump(ceps_json, arquivo)
```

Nesse momento, os dados já estão salvos no arquivo local.json. Então, é criado outro dicionário carregando os dados com a função deserializa. O exemplo imprime alguns dados do CEP msf_brasil.

```
    print(novo_ceps['msf_brasil']['logradouro'])
    # Rua do Catete
    print(novo_ceps['msf_brasil']['bairro'])
    # Catete
    print(novo_ceps['msf_brasil']['localidade'])
    # Rio de Janeiro
    print(novo_ceps['msf_brasil']['uf'])
    # RJ
    print(novo_ceps['msf_brasil']['complemento'])
    # até 195/196
```

A função deserializa faz uso do método json.load para carregar os dados do arquivo.

```
def deserializa():
    with open('local.json') as arquivo:
        return json.load(arquivo)
```

Atividades práticas 

Interagindo com o ambiente

- Bibliotecas como os e shutil facilitam a interação com o Sistema Operacional.
- Precisam ser importadas: import os, import shutil.
- São muito vastas e o uso das funções dir() e help() ajuda a conhecê-las melhor.
- >>> dir(os)
- >>> dir(os.path)
- >>> help(os)
- >>> help(os.path)
- >>> help(shutil)

Uma das tarefas mais comuns para programadores é interagir com o Sistema Operacional. Tarefas simples como pesquisar conteúdo de diretórios, descobrir se um diretório ou arquivo existe, copiar ou mover arquivos são consideradas básicas, mas podem variar de acordo com o Sistema Operacional (o comando correspondente pode ser diferente).

A linguagem Python possui algumas bibliotecas com recursos para facilitar a interação com o Sistema Operacional. Duas das mais famosas são os e shutil. A biblioteca os, por exemplo, utiliza os mesmos métodos independentemente do Sistema Operacional onde a linguagem Python foi instalada.

Como são bibliotecas muito vastas, é importante lembrar de usar as funções dir() e help() para conhecê-las melhor. Para usar essas funções, é necessário importar as bibliotecas e passar os módulos, seus métodos ou propriedades como parâmetro para as funções.

Em modelo06.py são criados alguns diretórios, que são depois percorridos e no final são removidos.

```
#!/usr/bin/env python3
# src/modelo06.py

import os
import shutil
from random import shuffle

def main():
    os.mkdir('dir_n1')
    os.chdir('dir_n1')
    print(os.getcwd())
    # /home/.../dir_n1
    gera_conteudo('arquivo.txt')
    os.mkdir('sub_dir1_n2')
    gera_conteudo('sub_dir1_n2/arquivo1.txt')
    gera_conteudo('sub_dir1_n2/arquivo2.txt')
    os.mkdir('sub_dir1_n2/sub_dir1_n3')
    gera_conteudo('sub_dir1_n2/sub_dir1_n3/arquivo1.txt')
    for topo, dirs, arqs in os.walk('sub_dir1_n2'):
        print(topo, dirs, arqs)
    # sub_dir1_n2 ['sub_dir1_n3'] ['arquivo1.txt', 'arquivo2.txt']
    # sub_dir1_n2/sub_dir1_n3 [] ['arquivo1.txt']
    # Descomentar a próxima linha para ver a exceção.
    # os.rmdir('sub_dir1_n2/sub_dir1_n3')
    # Traceback (most recent call last):
    #  ...
    # OSError: [Errno 39] Directory not empty: 'sub_dir1_n2/sub_dir1_n3'
    shutil.rmtree('sub_dir1_n2')
    os.chdir('..')
    shutil.rmtree('dir_n1')


def gera_conteudo(nome_arquivo):
    numeros = list(range(70))
    numeros = [str(numero) for numero in numeros]
    with open(nome_arquivo, 'w') as arquivo:
        for indice in range(30):
            shuffle(numeros)
            arquivo.write(''.join(numeros))


if __name__ == '__main__':
    main()
```

O exemplo começa com a criação do diretório, mudança para o diretório e acesso ao diretório atual. Os métodos usados foram, respectivamente, os.mkdir, os.chdir, os.getcwd.

```
os.mkdir('dir_n1')
os.chdir('dir_n1')
print(os.getcwd())
```

O próximo trecho interessante do exemplo é a navegação pelo diretório. O método os.walk percorre um diretório e para cada diretório encontrado ele retorna as seguintes informações:

- O nome do diretório sendo percorrido, variável topo do exemplo.
- Os nomes diretórios que ele contém, variável dirs do exemplo.
- Os arquivos que estão no diretório sendo percorrido, variável arqs do exemplo.

```
for topo, dirs, arqs in os.walk('sub_dir1_n2'):
    print(topo, dirs, arqs)
sub_dir1_n2 ['sub_dir1_n3'] ['arquivo1.txt', 'arquivo2.txt']
sub_dir1_n2/sub_dir1_n3 [] ['arquivo1.txt']
```

Por fim, no exemplo, os diretórios são removidos. A primeira tentativa é feita com o método os.rmdir, mas ele não permite que seja removido um diretório que não está vazio. Em seguida é utilizado o método shutil.rmtree, que remove o diretório e seu conteúdo sem qualquer confirmação prévia. Tenha cuidado ao utilizar esse método para não remover algum diretório importante sem querer.


```
for topo, dirs, arqs in os.walk('sub_dir1_n2'):
    print(topo, dirs, arqs)
sub_dir1_n2 ['sub_dir1_n3'] ['arquivo1.txt', 'arquivo2.txt']
sub_dir1_n2/sub_dir1_n3 [] ['arquivo1.txt']
```


Outros recursos interessantes estão disponíveis no módulo glob, que permite obter listas de nomes em diretórios utilizando caracteres coringa como "?" e "*". Arquivos e diretórios que possuem o nome começado por ponto (".") não constam nos resultados onde são usados "?" e "*". É necessário utilizar o caractere ponto "." para obter arquivos ou diretórios que tenham o nome começado por ponto.

Voltando ao módulo sys, a variável sys.argv é inicializada com os parâmetros de linha de comando, sendo o primeiro elemento o nome do script, seguido pelos parâmetros que eventualmente tenham sido passados. Já sys.stdin, sys.sdtout e sys.stderr são objetos do tipo arquivo, que lidam com os dispositivos padrão de entrada ou saída do computador.

Quando é utilizada a função print, o texto que é passado como parâmetro é impresso no terminal, então é dito que a saída padrão é o terminal (sys.stdout). Além da saída padrão, temos a saída de erros, que pode ser o terminal também (sys.stderr).

- O módulo glob permite criar listas a partir de pesquisas em diretórios usando o carácter coringa '*'.
- Para receber argumentos de linha de comando, usamos a propriedade argv do módulo sys.
- O primeiro argumento de sys é o nome do script.
- sys.stderr escreve na saída de erro padrão do Sistema Operacional e mostra a mensagem mesmo quando a saída padrão foi redirecionada.

Em modelo07.py são apresentados exemplos de uso do glob, sys.argv e sys.stderr, considerando que o script seja executado no diretório de exemplos e exercícios da apostila (os resultados podem variar dependendo do conteúdo do diretório).

```
#!/usr/bin/env python3
# src/modelo07.py

import glob
import sys

def lista_dir(coringa):
    return sorted(glob.glob(coringa))

def main(argumentos):
    if argumentos is None:
        argumentos = sys.argv
    if len(argumentos) < 2:
        msg = 'Número inválido de argumentos. Abortando.\n'
        sys.stderr.write(msg)
        sys.exit(1)
    return lista_dir(argumentos[1])

if __name__ == "__main__":
    print(lista_dir('*.py'))
    # ['__init__.py', 'modelo01.py', 'modelo02.py', ...]
    print(lista_dir('*.out'))
    # ['autores.out']
    print(lista_dir('*.txt'))
    # ['citacoes.txt', ...]
    print(main(['modelo07.py', '*.json']))
    # ['modelo07.json']
    print(main(['*.py']))
    # Traceback (most recent call last):
    #  ...
    # SystemExit: 1
```

O exemplo implementa uma função que utiliza o módulo glob para retornar uma lista de nomes, de acordo com o parâmetro passado. No exemplo são exibidos todos os arquivos com extensão py, out e txt.

```
    print(lista_dir('*.py'))
    # ['__init__.py', 'modelo01.py', 'modelo02.py', ...]
    print(lista_dir('*.out'))
    # ['autores.out']
    print(lista_dir('*.txt'))
    # ['citacoes.txt']
```

A função principal, main, é implementada esperando uma lista com no mínimo dois elementos. O primeiro é o nome do script e o restante são padrões para pesquisa no diretório.

```
    print(main(['modelo07.py', '*.json']))
    # ['modelo07.json']
    print(main(['*.py']))
    # Traceback (most recent call last):
    #  ...
    # SystemExit: 1
```

Quando o número de elementos da lista é menor do que o esperado, uma mensagem de erro é escrita na saída de erros, utilizando sys.stderr, e o programa é encerrado utilizando sys.exit.

```
def main(argumentos):
    if argumentos is None:
        argumentos = sys.argv
    if len(argumentos) < 2:
        msg = 'Número inválido de argumentos. Abortando.\n'
        sys.stderr.write(msg)
        sys.exit(1)
    return lista_dir(argumentos[1])
```

Como exemplo de execução, o exemplo pode ser chamado passando um padrão como parâmetro. É necessário proteger o padrão, pois o shell do Linux captura o padrão e pode processar antes de ser passado para o programa exemplo.

```
$ python3 exemplo13.py '*.json'
['exemplo11.json', 'exercicio10-df.json']
```

Quando for necessário, o resultado da execução pode ser direcionado para um arquivo. Para exibir o conteúdo de arquivos, é utilizado o comando cat do Linux.

```
$ python3 exemplo13.py '*.json' > /tmp/teste.txt
$ cat /tmp/teste.txt
['exemplo11.json', 'exercicio10-df.json']
```

Módulo de expressões regulares

- O módulo re é usado quando é necessário usar expressões regulares para processamento de strings.
- Útil para serviços mais complexos de busca e correspondência usando strings.
- Para coisas simples é recomendado usar os métodos de strings, mais fáceis de ler e depurar.

Expressões regulares servem para identificar cadeias de caractere que correspondem a um certo padrão. Editores de texto e linguagens de programação fazem uso de expressões regulares para busca, substituição, filtro de informação e validação de textos específicos, como números de telefone, CEP, CPF, RG, formatos de data, entre outros.

A linguagem Python possui o módulo re, que implementa os recursos para realizar as tarefas mais comuns com expressões regulares. Expressões regulares são úteis e recomendadas para realizar serviços complexos de busca e correspondência usando strings. Entretanto, deve ser tomado o cuidado de não exagerar no seu uso. Em alguns casos, os métodos de string realizam as mesmas tarefas e são mais fáceis de ler e depurar.

O exemplo a seguir, modelo08.py, implementa três funções que utilizam expressões regulares para realizar operações de validação.

```
#!/usr/bin/env python3
# src/modelo08.py

import re

def verifica_cep(cep):
```

```
    expressao = '\d{5}-\d{3}'
    return re.match(expressao, cep) is not None

def verifica_hora(hora):
    expressao = '([01]\d|2[0-4]):[0-5]\d'
    return re.match(expressao, hora) is not None

def verifica_nome_identificador(identificador):
    expressao = '[a-zA-Z_][a-zA-Z0-9_]*'
    return re.match(expressao, identificador) is not None

if __name__ == '__main__':
    print(verifica_cep('80000-123'))
    # True
    print(verifica_cep('123-123'))
    # False
    print(verifica_hora('11:02'))
    # True
    print(verifica_hora('25:03'))
    # False
    print(verifica_nome_identificador('var5_valida3'))
    # True
    print(verifica_nome_identificador('2_var_invalida'))
    # False
```

Uma das funções de validação implementa a verificação de CEP. A regra é que o CEP deve ter cinco dígitos, um traço e mais três dígitos para ser válido.

```
    print(verifica_cep('80000-123'))
    # True
    print(verifica_cep('123-123'))
    # False
```

A expressão utilizada na função utiliza o padrão \d para indicar que o texto deve ter um dígito. Seguindo o dígito, temos um modificador de quantidade entre chaves. Esse número indica quantos dígitos o texto deve ter. Antes do traço são 5 e depois são 3, portanto, \d{5} e \d{3}.

```
def verifica_cep(cep):
    expressao = '\d{5}-\d{3}'
    return re.match(expressao, cep) is not None
```

O método match retorna um objeto match caso exista correspondência. Caso contrário, retorna nulo (None).

A regra para a função que valida hora tem uma particularidade. Não basta apenas verificar se o texto possui dois dígitos seguidos de dois pontos (":") e mais dois dígitos. O dia só tem 24 horas e uma hora só tem 60 minutos. Portanto, a expressão deve prever esses detalhes.

```
    print(verifica_hora('11:02'))
    # True
    print(verifica_hora('25:03'))
    # False
```

Para atender a regra de validação de hora, é necessário criar uma condição. Caso a parte do texto que corresponde à hora comece com os dígitos zero (“0”) ou um (“1”), o próximo caractere pode ser qualquer dígito. Para tanto, temos a seguinte expressão: [01]\d.

Se a parte do texto que corresponde à hora começar com o dígito dois (“2”), o próximo caractere deve estar entre zero (“0”) e quatro (“4”), condição que é refletida na expressão 2[0-4]. Para agrupar essas duas condições, é necessário usar parênteses, indicando ainda que apenas uma dessas alternativas deve ser considerada. Para tanto, faz-se uso do caractere pipe (“|”). Depois do caractere “dois pontos”, temos a parte dos minutos, cujo primeiro dígito pode variar entre zero (“0”) e cinco (“5”), seguido de qualquer dígito. Nesse caso, a expressão que reflete essa condição é [0-5]\d.

```
def verifica_hora(hora):
    expressao = '([01]\d|2[0-4]):[0-5]\d'
    return re.match(expressao, hora) is not None
```

A última expressão valida um nome de variável de acordo com as regras da linguagem Python. Uma variável em Python deve ter o nome começado por uma letra ou pelo caractere sublinhado (“_”). Em seguida, pode ser qualquer letra, dígito ou o carácter sublinhado (“_”). A versão atual da linguagem, python3, aceita também caracteres acentuados nos nomes de identificador (as versões anteriores não aceitavam).

No exemplo, a expressão utilizada não aceita acentos.

```
    print(verifica_nome_identificador('var5_valida3'))
    # True
    print(verifica_nome_identificador('2_var_invalida'))
    # False
```

A regra para validar a primeira letra é definida usando colchetes e as opções que o caractere pode assumir. [a-zA-Z_] indica que a primeira letra pode ser qualquer uma das opções de “a” até “Z” ou “_”. Para validar o restante, são utilizados os colchetes também, mas agora além das opções de “a” até “Z” e o sublinhado, também podem ocorrer os dígitos de “0” até “9”. Note é que o modificador asterisco é utilizado, indicando que o padrão anterior pode ocorrer zero (“0”) ou mais vezes.

```
def verifica_nome_identificador(identificador):
    expressao = '[a-zA-Z_][a-zA-Z0-9_]*'
    return re.match(expressao, identificador) is not None
```

Acesso à internet

- Python tem várias bibliotecas para uso de rede e internet.
- Algumas delas são: socket, http, smtplib e urllib, webbrowser, cgi, wsgiref, ftplib, poplib, imaplib.
- Além dessas bibliotecas padrão, existem bibliotecas de terceiros ainda mais específicas.

A quantidade de bibliotecas de acesso à rede e internet disponíveis na linguagem Python a tornou uma das preferidas de especialistas em segurança. Ela é muito utilizada para criação de scripts rápidos e poderosos para acessar serviços e criar servidores com poucas linhas de código. Entre as bibliotecas mais conhecidas estão socket, http, smtplib, urllib, webbrowser, cgi, wsgiref, ftplib, poplib, imaplib. A seguir, uma tabela com breve descrição de cada módulo.

Biblioteca	Descrição
socket	Esse módulo prove acesso a interface de sockets do Sistema Operacional.
webbrowser	Interface de alto nível para exibição de documentos web.
cgi	Define utilitários usados por scripts CGI escritos em Python.
wsgiref	Implementa a especificação WSGI, que pode ser usada para dar suporte WSGI a servidores web ou frameworks.
urllib	Pacote que contém diversos módulos para lidar com URLs.
http	Pacote que contém diversos módulos para trabalhar com o protocolo HTTP.
ftplib	Módulo contendo a classe cliente FTP, que pode ser usada para automatizar tarefas de acesso a servidores ftp.
poplib	Módulo que define a classe POP3, que encapsula uma conexão em um servidor POP3.
imaplib	Módulo que define três classes as quais encapsulam a conexão com servidores IMAP4.
smtplib	Esse módulo define um objeto cliente de sessão SMTP, que pode ser usado para enviar e-mail para qualquer máquina da internet com serviço SMTP habilitado.

Tabela 5.1
Tabela com descrição dos módulos.

Além dessas bibliotecas, existem muitas bibliotecas de terceiros para as mais variadas tarefas envolvendo redes e processamento distribuído.

O programa modelo09.py implementa uma função que consulta o serviço de CEPs usando um serviço disponível na internet. Vale lembrar que esses serviços podem sofrer alterações ou deixar de funcionar.

```
#!/usr/bin/env python3
# src/modelo09.py

from urllib.request import urlopen
from json import loads

def consulta_cep(cep):
    url_template = 'https://viacep.com.br/ws/{cep}/json/'
    url = url_template.format(cep=cep)
    local_json = urlopen(url).read()
    local_str = local_json.decode()
    local = loads(local_str)
    return local

def formata_cep(local):
    return 'Endereço: {logradouro} - ' \
        'Cidade: {localidade} - ' \
        'Estado: {uf}.'.format(**local)


if __name__ == '__main__':
```

```
    local = consulta_cep('22290906')
    ordenados = [item for item in sorted(local.items())]
    filtra_info =  [item for item in ordenados
                    if 'info' not in item[0]]
    for campo in filtra_info:
        print(campo[0], campo[1])
    # bairro Botafogo
    # cep 22290-906
    # complemento 116
    # gia
    # ibge 3304557
    # localidade Rio de Janeiro
    # logradouro Rua Lauro Muller
    # uf RJ
    print(formata_cep(local))
    # Endereço: Rua Lauro Muller, 116 - Cidade: Rio de Janeiro ...
```

Para obter o resultado da consulta, é utilizada a função urllib.urlopen(). O conteúdo obtido pela internet precisa ser convertido em string, por isso é usado o método decode(). O serviço de consulta retorna as informações no formato json, que através da função json.loads() é convertido para uma estrutura de dados da linguagem Python, no caso um dicionário.

```
def consulta_cep(cep):
    url_template = 'https://viacep.com.br/ws/{cep}/json/'
    url = url_template.format(cep=cep)
    local_json = urlopen(url).read()
    local_str = local_json.decode()
    local = loads(local_str)
    return local
```

O exemplo16.py utiliza a função consulta_cep e depois imprime o resultado, mas filtra algumas informações adicionais para exibir apenas uma parte específica do endereço.

```
    local = consulta_cep('22290906')
    ordenados = [item for item in sorted(local.items())]
    filtra_info =  [item for item in ordenados
                    if 'info' not in item[0]]
    for campo in filtra_info:
        print(campo[0], campo[1])
    # bairro Botafogo
    # cep 22290-906
    # complemento 116
    # gia
    # ibge 3304557
    # localidade Rio de Janeiro
    # logradouro Rua Lauro Muller
    # uf RJ
```

Logo em seguida os dados do endereço são impressos de uma maneira mais agradável após a execução da função formata_cep.

```
    print(formata_cep(local))
    # Endereço: Rua Lauro Muller, 116 - Cidade: Rio de Janeiro …
```

Essa função apenas formata o dicionário, chamado de local, para imprimir apenas o nome da rua com número, nome da cidade e estado.

```
def formata_cep(local):
    return 'Endereço: {logradouro} - ' \
        'Cidade: {localidade} - ' \
        'Estado: {uf}.'.format(**local)
```

Acesso a bancos de dados

- PEP 249: Python Database API Specification v2.0.
- Especifica uma interface na tentativa de padronizar bibliotecas de acesso a bancos de dados.
- É esperado que isso facilite o aprendizado das bibliotecas, aumente a portabilidade de código e aumente o alcance da conectividade da linguagem Python com bancos de dados.

A linguagem Python possui uma especificação especialmente escrita para quem deseja implementar bibliotecas de acesso a bancos de dados. Essa especificação está descrita através de uma proposta de melhoria, a PEP-0249, conhecida também como Python Database API Specification 2.0 (API de banco de dados versão 2).

Essa especificação substitui a proposta anterior (PEP-0248), dando continuidade a um movimento de padronização entre diferentes fabricantes de bancos de dados. Seguindo essas PEPs, a implementação das diversas bibliotecas de acesso a diferentes bancos de dados se tornam muito semelhantes, facilitando o aprendizado e o uso.

Existem várias formas de acesso a bancos de dados, sendo uma das mais comuns aquela que se baseia no padrão conhecido como ODBC. A linguagem Python possui vários módulos para acesso ODBC a bancos de dados. A linguagem Java possui um padrão próprio chamado de JDBC, que pode ser utilizado também em Python através do interpretador Jython.

Outro padrão conhecido é o ADO, da Microsoft, suportado também em Python através do módulo adodbapi. Existem também diversas bibliotecas de terceiros, prometendo ganhos de performance em função da sua especialização. O resultado disso é que o funcionamento das respectivas bibliotecas difere um pouco. Isso pode exigir uma consulta à documentação da biblioteca para garantir o seu uso da forma adequada, mas o importante é que todos os principais bancos de dados disponíveis no mercado contam com bibliotecas que permitem a conexão em programas Python.

- A maioria dos bancos de dados suportam ODBC. Existem vários módulos para acesso ODBC.
- Bancos de dados Java com suporte a JDBC podem ser acessados com jython.
- Python também suporta ADO, interface de alto nível para bancos de dados da Microsoft.
- Além disso, existem diversas bibliotecas de terceiros para acesso a bancos de dados específicos, tais como IBM DB2, Firebird (e Interbase), Informix, Ingres, MySQL, Oracle, PostgreSQL, SAP DB (também conhecido como “MaxDB”), Microsoft SQL Server, Sybase etc.

O termo banco de dados normalmente remete a bancos de dados relacionais, conhecidos por fazer uso de uma linguagem chamada SQL. Existem, contudo, bancos de dados que não implementam a arquitetura relacional e que estão se tornando cada vez mas comuns em função da necessidade de processamento de quantidades absurdas de informação via aplicações web. Esses bancos são chamados de bancos de dados não somente relacionais ("Not Only SQL"), ou ainda, bancos de dados NO-SQL.

Python também possui interfaces para a maioria dos bancos de dados não relacionais em uso no mercado. Entre eles temos o ZODB, orientado a objetos, e que é considerado bastante poderoso, que é escrito em Python. Outros exemplos de bancos de dados não relacionais suportados pela linguagem Python são: Cassandra, Riak, MongoDB, CouchDB e Redis.

- Python também tem adaptadores para utilização de bancos de dados NOSQL.
- ZODB: banco de dados orientado a objetos escrito em Python.
- Outros exemplos de bancos NOSQL bastante utilizados: Cassandra, Riak, MongoDB, CouchDB e Redis.

Para demostrar o uso da api de banco de dados em Python, será utilizado o banco de dados sqlite. Não faz parte do escopo do curso ensinar conceitos relacionados com banco de dados ou com a linguagem SQL. Esses são explorados em outros cursos da grade da ESR.

Mais detalhes sobre as interfaces de banco de dados da linguagem Python podem ser obtidas no wiki oficial: https://wiki.python.org/moin/DatabaseInterfaces

Sqlite

- Banco de dados simples que oferece armazenamento local de dados.
- Utilizado para aplicações individuais e dispositivos.
- Enfatiza economia, eficiência, confiabilidade, independência e simplicidade.
- É autocontido, não possui servidor, não precisa de configuração e tem suporte a transações.

O banco de dados Sqlite é um banco de dados simples que armazena os dados em um arquivo local. Foi projetado para ser usado em aplicações individuais e para dispositivos como telefones celulares. Sua ênfase é na economia, eficiência, confiabilidade, independência e simplicidade.

O Sqlitep possui suporte a transações, mas não possui servidor e nem precisa de configurações.

Para utilizar o Sqlite com a linguagem Python, basta importar o módulo correspondente (sqlite3) e criar uma conexão. Feito isso, basta usar comandos padrão SQL.

```
# /usr/bin/env python3
# src/modelo10.py

# Quando é utilizado um outro banco de dados é necessário
# importar a biblioteca específica do banco.
# No caso do Postgresql seria:
# import psycopg2
import sqlite3
from datetime import date

def conectar_banco(nome_bd):
```

```
    ''' Para o Postgresql usamos o método da biblioteca
    psycopg2. Dessa forma:
    return psycopg2.connect(nome_bd)
    '''
    return sqlite3.connect(nome_bd)

def criar_cursor(conexao):
    ''' Isso é padronizado, o uso no psycopg2 é idêntico.'''
    return conexao.cursor()

def criar_tabela_computador(cursor):
    '''Existem poucas diferenças entre o SQL do sqlite e do
    Postgresql, nesse caso o comando é idêntico.'''
    comando = 'CREATE TABLE IF NOT EXISTS' \
              '    computador (' \
              '        codigo INTEGER PRIMARY KEY,' \
              '        nome VARCHAR,' \
              '        aquisicao DATE,' \
              '        vida INTEGER,' \
              '        marca VARCHAR' \
              '    )'
    executar_comando(cursor, comando)


def executar_comando(cursor, comando, parametros=None):
    ''' Isso é padronizado, o uso no psycopg2 é idêntico.'''
    if parametros:
        cursor.execute(comando, parametros)
    else:
        cursor.execute(comando)
    return cursor


if __name__ == '__main__':
    conexao = conectar_banco(':memory:')
    print(type(conexao))
    # <class 'sqlite3.Connection'>
    cursor = criar_cursor(conexao)
    print(type(cursor))
    # <class 'sqlite3.Cursor'>
    criar_tabela_computador(cursor)
```

Uma característica interessante do Sqlite é que ele pode ser criado em memória sem a necessidade de armazenar um banco no computador. Isso é especialmente útil para testes e demonstrações. A primeira providência é importar a biblioteca sqlite3.

```
import sqlite3
from datetime import date
```

```
def conectar_banco(nome_bd):
    ''' Para o Postgresql usamos o método da biblioteca
    psycopg2. Dessa forma:
    return psycopg2.connect(nome_bd)
    '''
    return sqlite3.connect(nome_bd)
```

Para criar uma conexão, usamos o método connect, que recebe o nome do banco. Como as bibliotecas são padronizadas, os métodos são idênticos.

```
conexao = conectar_banco(':memory:')
```

A função conectar_banco recebe o nome do banco, nesse caso a palavra :memory:, que permite o uso de um banco para demonstração ou testes. Após o encerramento do programa, esse banco e os dados contidos nele serão perdidos. Com outros bancos de dados, o que muda é o nome do banco. No Postgresql, por exemplo, seria um identificador do banco conhecido como DSN: Database Server Name, conforme demonstrado a seguir:

```
config_banco = "dbname=rnp user=aluno password=rnpesr"
conexao = conectar_banco(config_banco)
```

Já a chamada do comando para conectar ao banco é idêntica.

```
def criar_cursor(conexao):
    ''' Isso é padronizado, o uso no postgresql é idêntico.'''
    return conexao.cursor()
```

Para interagir com o banco, precisamos de um cursor. O cursor é uma estrutura de controle, uma área de memória que permite a manipulação dos dados de um banco de dados. O método cursor da conexão cria essa estrutura para que seja possível lidar com os dados do banco.

```
def criar_tabela_computador(cursor):
    '''Existem poucas diferenças entre o SQL do sqlite e do
    Postgresql, nesse caso o comando é idêntico.'''
    comando = 'CREATE TABLE IF NOT EXISTS' \
              '    computador (' \
              '        codigo INTEGER PRIMARY KEY,' \
              '        nome VARCHAR,' \
              '        aquisicao DATE,' \
              '        vida INTEGER,' \
              '        marca VARCHAR' \
              '    )'
    executar_comando(cursor, comando)


def executar_comando(cursor, comando, parametros=None):
    ''' Isso é padronizado, o uso no psycopg2 é idêntico.'''
    if parametros:
        cursor.execute(comando, parametros)
    else:
        cursor.execute(comando)
    return cursor
```

A função executar_comando foi preparada para facilitar o uso do cursor. No trecho de código anterior, a função é usada pela outra função, criar_tabela_computador, que executa um comando SQL para criar uma tabela chamada computador. O método execute também segue a padronização da linguagem Python para bancos de dados e seu uso praticamente idêntico em diferentes bancos de dados.

Manipulando dados no SQLite

Uma vez criado o banco, podemos criar tabelas e inserir registros.

- Os comandos são enviados ao banco de dados através de um objeto do tipo cursor.
- É possível passar parâmetros para os comandos SQL.
- O cursor possui métodos para recuperar os registros retornados por consultas SQL. Alguns exemplos são fetchone() e fetchall().

Modelo11.py utiliza as funções definidas em modelo10.py para conectar ao banco e criar o cursor. No modelo11 são implementadas consultas para seleção, inclusão e atualização de dados no banco Sqlite.

```
# /usr/bin/env python3
# src/modelo11.py

from datetime import date
from modelo10 import conectar_banco
from modelo10 import criar_cursor
from modelo10 import criar_tabela_computador
from modelo10 import executar_comando

def inserir_registro_computador(
        cursor, codigo, nome, aquisicao, vida, marca):
    '''No psycopg2 o comando SQL INSERT seria assim:
    comando = 'INSERT INTO computador(' \
              '        codigo, nome, aquisicao, vida, marca' \
              '    ) VALUES (%s, %s, %s, %s, %s)'
    '''
    comando = 'INSERT INTO computador(' \
              '        codigo, nome, aquisicao, vida, marca' \
              '    ) VALUES (?, ?, ?, ?, ?)'
    parametros = (codigo, nome, aquisicao, vida, marca)
    executar_comando(cursor, comando, parametros)

def main():
    conexao = conectar_banco(':memory:')
    cursor = criar_cursor(conexao)
    criar_tabela_computador(cursor)
    inserir_registro_computador(
        cursor, 1, 'Vastra', date(2015, 1, 10), 365, 'dall')
    inserir_registro_computador(
        cursor, 2, 'Polvilion', date(2015, 1, 10), 365, 'lp')
```

```
    comando_sql = "SELECT * FROM computador"
    registros = executar_comando(cursor, comando_sql)
    registro = registros.fetchone()
    print(registro)
    # (1, 'Vastra', '2015-01-10', 365, 'dall')
    #
    # Como o sqlite não tem tipo data campos data retornam como
    # string, o retorno no Postgresql seria:
    # (1, 'Vastra', datetime.date(2015, 1, 10), 365, 'dall')

    registro = registros.fetchone()
    print(registro)
    # (2, 'Polvilion', '2015-01-10', 365, 'lp')
    comando_sql = "UPDATE computador " \
                  "SET vida = ? where codigo = ?"

    # Na biblioteca do Postgresql, psycopg2, existe uma pequena
    # diferença, o marcador de substituição é o %s. Então o
    # comando SQL UPDATE acima deveria ser escrito assim:
    # comando_sql = "UPDATE computador " \
    #               "SET vida = %s where codigo = %s"

    parametros = (600, 1,)
    registros = executar_comando(
        cursor, comando_sql, parametros)

    print(registros.rowcount)
    # 1
    parametros = (400, 2,)
    registros = executar_comando(
        cursor, comando_sql, parametros)

    print(registros.rowcount)
    # 1
    comando_sql = "SELECT * FROM computador"
    registros = executar_comando(cursor, comando_sql)
    for registro in registros.fetchall():
        print(registro)
    # (1, 'Vastra', '2015-01-10', 600, 'dall')
    # (2, 'Polvilion', '2015-01-10', 400, 'lp')

    # Por causa do tipo data a saída no Postgresql seria assim:
    # (1, 'Vastra', datetime.date(2015, 1, 10), 600, 'dall')
    # (2, 'Polvilion', datetime.date(2015, 1, 10), 400, 'lp')
if __name__ == '__main__':
    main()
```

Em modelo11.py, os comandos que criam o banco e a tabela precisam ser executados novamente, pois o banco é criado na memória do computador e deixa de existir depois que o programa é executado.

```
def inserir_registro_computador(
        cursor, codigo, nome, aquisicao, vida, marca):
    '''No psycopg2 o comando SQL INSERT seria assim:
    comando = 'INSERT INTO computador(' \
              '        codigo, nome, aquisicao, vida, marca' \
              '    ) VALUES (%s, %s, %s, %s, %s)'
    '''
    comando = 'INSERT INTO computador(' \
              '        codigo, nome, aquisicao, vida, marca' \
              '    ) VALUES (?, ?, ?, ?, ?)'
    parametros = (codigo, nome, aquisicao, vida, marca)
    executar_comando(cursor, comando, parametros)
```

Dica: é muito importante usar essa forma, com marcas para substituição, para evitar possíveis enganos ou tentativas de manipulação indevida de dados no banco. Existe um ataque muito comum em sistemas na internet chamado injeção de SQL, que pode ser evitado com práticas como essa, utilizando marcas para substituição.

A tabela computador armazenará dados de computador. Para inserir os dados na tabela, foi criada a função inserir_registro_computador, que usa o comando SQL INSERT. As marcas de interrogação representam valores de parâmetros que serão substituídos pelo método execute.

```
>>> inserir_registro_computador(
...     cursor, 1, 'Vastra', date(2015, 1, 10), 365, 'dall')
>>> inserir_registro_computador(
...     cursor, 2, 'Polvilion', date(2015, 1, 10), 365, 'lp')
```

Para usar a função inserir_registro_computador, basta passar os parâmetros que o comando INSERT será executado.

```
>>> comando_sql = "SELECT * FROM computador"
>>> registros = executar_comando(cursor, comando_sql)
>>> registro = registros.fetchone()
```

Para recuperar os dados gravados no banco, é realizada uma consulta. As consultas em bancos de dados SQL utilizam o comando SELECT. A função executar_comando nos retorna um cursor e através do método fetchone() recuperamos um registro do banco.

```
>>> print(registro)
(1, 'Vastra', '2015-01-10', 365, 'dall')
>>> registro = registros.fetchone()
>>> print(registro)
(2, 'Polvilion', '2015-01-10', 365, 'lp')
```

O método fetchone() nos retorna o registro em forma de uma tupla. Executando novamente o mesmo método, ele vai nos retornar o próximo registro do banco de dados.

```
>>> comando_sql = "UPDATE computador " \
                  "SET vida = ? where codigo = ?"
>>> parametros = (600, 1,)
>>> registros = executar_comando(
...     cursor, comando_sql, parametros)
...
>>> print(registros.rowcount)
1
```

```
>>> parametros = (400, 2,)
>>> registros = executar_comando(
...     cursor, comando_sql, parametros)
...
>>> print(registros.rowcount)
1
```

Quando é necessário efetuar uma alteração em um registro do banco de dados, o comando SQL utilizado é o UPDATE. Nesse caso, o cursor exibe, através da variável rowcount, o número de linhas afetadas pelo comando UPDATE.

```
>>> comando_sql = "SELECT * FROM computador"
>>> registros = executar_comando(cursor, comando_sql)
>>> for registro in registros.fetchall():
...     print(registro)
(1, 'Vastra', '2015-01-10', 600, 'dall')
(2, 'Polvilion', '2015-01-10', 400, 'lp')
```

Outra forma de recuperar os registros obtidos do banco de dados é com o método fetchall(), que retorna uma lista com as tuplas dos registros de uma única vez.

Mapeando dados para objetos

- Técnica de desenvolvimento usada para representar dados relacionais em termos de Programação Orientada a Objetos.
- Classes representam tabelas do banco de dados e objetos (instâncias) representam registros.
- Evita a necessidade de escrita de comandos SQL, gerados automaticamente pela própria biblioteca.

Quando se trata de lidar com bancos de dados em qualquer linguagem, uma das tarefas mais repetitivas é criar os comandos SQL para salvar, alterar e recuperar os dados do banco de dados. Outro desafio é a necessidade de lidar com as diferenças entre os bancos de dados. Para ajudar a lidar com essas tarefas, o mais comum é usar uma biblioteca que faça esse trabalho de gerar os comandos SQL e que cuide das diferenças entre os diversos bancos de dados. As bibliotecas que fazem esse serviço são chamadas de bibliotecas de mapeamento objeto relacional ou pela sigla em inglês, ORM.

O mapeamento objeto relacional é uma representação de tabelas e registros de bancos de dados relacionais como objetos. Os objetos são manipulados usando programação orientada a objetos e uma biblioteca ORM cuida para que estes sejam corretamente armazenados e recuperados dos bancos de dados.

As bibliotecas ORM possuem características em comum: classes representam as tabelas do banco de dados e as instâncias das classes, os objetos, representam os registros.

A maior parte das operações de banco de dados (inclusão, consulta, alteração e remoção) podem ser realizadas sem a necessidade de usar comandos SQL. As bibliotecas cuidam das respectivas “traduções”, mas elas também permitem que comandos SQL sejam executados sempre que for necessário.

A linguagem Python possui algumas bibliotecas de mapeamento objeto relacional. Entre elas destacamos:

- **SQLAlchemy:** http://www.sqlalchemy.org/
- **Django ORM:** https://docs.djangoproject.com/en/1.8/#the-model-layer
- **Peewee:** http://docs.peewee-orm.com/en/latest/
- **PonyORM:** http://ponyorm.com/
- **SQLObject:** http://www.sqlobject.org/

SQLAlchemy é uma das mais famosas, é baseada em uma bibliotecas ORM muito conhecido por programadores da linguagem Java, o Hibernate. Já o Django ORM faz parte do framework web Django, que será explorado nas próximas sessões.

Peewee é uma biblioteca simples e pequena, mas é expressiva e expansível através de addons. PonyORM é uma biblioteca moderna que permite consultas ao banco de dados usando generators que são traduzidos em SQL. Um dos seus diferenciais é um editor gráfico para os modelos do banco. Finalmente, SQLObject é um ORM que fornece uma interface para o banco tratando classes como tabelas, registros como instâncias e colunas como atributos. Inclui também uma linguagem de consultas baseada em objetos Python que é uma abstração do SQL, tornando as aplicações mais independentes de bancos de dados.

Atividades práticas AVA

6

Introdução ao Django

objetivos

Apresentar o framework Django, contando um pouco da sua história e características. Mostrar ao aluno como instalar e configurar o Django, assim como criar e configurar um projeto, suas aplicações e banco de dados com base nesse framework.

conceitos

Framework; MTV; Model; View; Template; URL; Estrutura de projetos Python; Métodos customizados.

Histórico

- Os primeiros sistemas para internet não possuíam tantos recursos e técnicas.
- Desenvolvedores identificaram tarefas repetitivas entre sistemas.
- As tarefas repetitivas começaram a ser organizadas e compartilhadas através de frameworks.
- O Django cresceu a partir de aplicações reais escritas para produzir e manter diversas aplicações em um ambiente ditado por prazos jornalísticos.
- Iniciado em 2003 para suprir demandas de funcionalidades e aplicações inteiras em dias ou até horas.
- Em 2005, o Django foi disponibilizado como software livre (licença BSD).
- Nomeado como homenagem a Django Reinhardt, um guitarrista de jazz.

Os primeiros sistemas desenvolvidos para internet não possuíam a variedade de recursos que existem hoje, até porque eram desenvolvidos com as técnicas e recursos existentes na época. Com o passar do tempo, desenvolvedores começaram a perceber que algumas partes de um sistema eram constantemente desenvolvidas novamente para outros sistemas. Começaram a perceber que essas partes "comuns" poderiam ser organizadas e compartilhadas entre sistemas. Assim começaram a surgir os diversos frameworks para desenvolvimento web, sendo Django um dos mais populares para a linguagem Python.

Os desenvolvedores que trabalhavam para o jornal Lawrence Journal-World, do grupo World Online, começaram a usar Python e com ele automatizar tarefas relacionadas com a publicação de notícias e todo o gerenciamento do site do jornal na internet. Foram desenvolvendo o conjunto de rotinas e funcionalidades que acabaram sendo organizadas em um framework batizado de Django.

O nome Django é uma homenagem ao guitarrista de jazz Django Reinhardt.

É um framework projetado para economizar tempo e preconiza o princípio "Don't repeat Yourself" (não seja repetitivo), estimulando que os desenvolvedores aproveitem ao máximo o código já existente, evitando reescrever trechos de programas semelhantes inúmeras vezes. O Django, por já fazer uso de várias tecnologias livres, foi disponibilizado também como software livre em 2005 (licença BSD).

Conceitos

- Dividido em três partes conceituais, Model, View e Template: MVT.
- Model define a estrutura do banco de dados.
- View seleciona e filtra quais dados serão apresentados.
- Template cuida da apresentação dos dados.
- Em comparação com o padrão MVC, a equipe responsável pelo Django considera que o controle está distribuído por várias partes do framework.

O Django segue o padrão de desenvolvimento chamado MTV, Model-View-Template. Similar ao modelo mais famoso, chamado de MVC, Model View Controller. Ambos os conceitos seguem a premissa de separar claramente o desenvolvimento em camadas. Isso é importante para evitar misturar o código que é responsável por mostrar os dados do código que é responsável por acessar dados ou do código responsável por implementar as regras de negócio.

No caso do Django, os conceitos do MVC estão distribuídos de outra forma. O modelo, assim como no MVC, representa a estrutura do banco de dados. A diferença é que no Django a camada chamada View é responsável por selecionar e filtrar os dados que devem ser apresentados, delegando a apresentação/exibição destes para a camada Template. Faltaria então a camada Controller do MVC. No caso do Django, os desenvolvedores do framework consideram que as várias partes do framework representam o Controller.

Figura 6.1 Django request-response cycle by: Ryan Nevius (http://www.ryannevius.com).

Organização do Django

Um sistema desenvolvido em Django é organizado em um projeto. O projeto é separado por aplicações. Cada aplicação possui, entre outros recursos, views, templates e models. São nessas três camadas que a maior parte do desenvolvimento ocorre.

- O desenvolvimento é realizado em um projeto.
- O projeto é separado por aplicações.
- As aplicações possuem views, templates e models.
- Um sistema de URLs direciona as requisições para views que utilizam templates para apresentar dados definidos nos models, usando o ORM do Django.

Outra parte importante do Django é o sistema de URLs. Esse sistema é responsável por associar uma URL para uma view. A view prepara os dados que devem ser apresentados e pode fazer uma seleção e filtro no banco de dados. Os dados então são encaminhados ao sistema de templates e uma resposta é enviada ao navegador do usuário.

URL
É o endereço de um recurso disponível em uma rede de computadores (desde uma rede local até a internet) e sua sigla é uma abreviação de "Uniform Resource Locator" (em inglês) que pode ser traduzido para Localizador Padrão de Recursos.

Ambiente de desenvolvimento

- Editor de texto simples.
- Ambiente virtual da linguagem Python.
- Virtualenv cria uma instalação de Python isolada.
- Permite que bibliotecas sejam instaladas em ambientes virtuais de acordo com os projetos sem que entrem em conflito.

Para trabalhar com Django é necessário, além do Python instalado, um editor de texto simples. Existem, entretanto, ferramentas muito úteis para aumentar a produtividade dos desenvolvedores. Uma delas é o virtualenv, integrada à versão 3 da linguagem Python, que isola a instalação do Django entre projetos e também do Sistema Operacional. Dessa forma, os projetos podem ter versões diferentes da mesma biblioteca sem que uma entre em conflito com a outra.

Os comandos usados para instalar e criar um ambiente virtual com a versão mais nova do Django, em sistemas Debian GNU/Linux ou Ubuntu, são os seguintes:

```
$ sudo apt install python3-pip
$ sudo apt install python3-venv
$ cd ~
$ mkdir curso
$ cd curso
$ python3 -m venv django
$ source django/bin/activate
(django) $ pip install django
```

Primeiro são instalados os pacotes do Sistema Operacional com os pacotes de módulos do Python necessários para criar o ambiente virtual. O comando cd ~ entra no diretório pessoal do usuário (p.e. /home/aluno). No exemplo anterior, mkdir curso cria o subdiretório curso, que deve ser o diretório a partir do qual deve ser criado o ambiente virtual. Temos uma instalação de Python isolada com o comando python3 -m venv django.

Feito isso, é preciso iniciar o ambiente, que é ativado através do comando source django/bin/activate. O sinal de que tudo funcionou corretamente é o aparecimento do nome do ambiente virtual (django) no canto esquerdo do terminal. Finalmente, com o comando pip install django, o Django é instalado dentro desse ambiente virtual. É possível visualizar as versões das bibliotecas instaladas com o comando:

```
(django) $ pip freeze
```

As versões podem ser "congeladas" com o comando:

```
(django) $ pip freeze > requirements.txt
```

Para instalar os pacotes usando as versões congeladas, seja em outro ambiente virtual, computador ou até mesmo outro servidor, basta usar o comando:

```
(django-servidor) $ pip install -r requitements.txtCriando o projeto
```

Para iniciar o desenvolvimento com Django, é necessário gerar algum código e algumas configurações iniciais que incluem configurações do banco, opções do Django e configurações específicas para a aplicação.

- Projeto baseado no tutorial do Django.
- Sistema de reservas. Cadastra clientes e reservas dos clientes.
- Teste para ver se o Django está instalado:
  - $ python3 -c "import django; print(django.get_version())"
- Criando o projeto:
  - $ django-admin startproject umas_e_ostras

Nesta sessão, será iniciado um sistema para controle de reservas em um estabelecimento, através de um projeto que será chamado de umas_e_ostras.

Para criar o projeto, é usado uma ferramenta auxiliar chamada de django-admin.
O comando a seguir criará o projeto:

```
$ django-admin startproject umas_e_ostrasEstrutura inicial do projeto
```

Ao iniciar um projeto, são gerados alguns arquivos de configuração e a estrutura básica de diretórios.

Projeto inicial com código básico e arquivos de configuração para instância do Django.

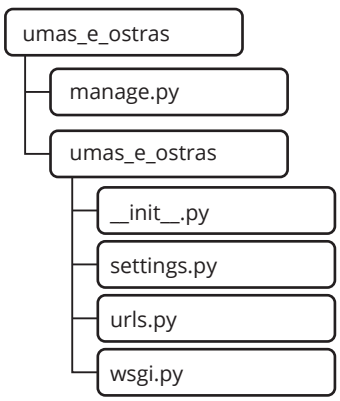

Figura 6.2
Estrutura básica de diretórios.

A partir dessa estrutura apresentada é que o projeto será desenvolvido. Vamos entender melhor essa estrutura e os arquivos gerados, já que cada um deles têm uma função específica.

- **Diretório raiz umas_e_ostras:** contém o projeto, o nome não faz diferença.
- **manage.py:** utilitário para interagir com o projeto.
- **Diretório interno umas_e_ostras:** pacote do projeto.
- **umas_e_ostras/__init__.py:** arquivo que indica que o diretório é um pacote.
- **umas_e_ostras/settings.py:** arquivo de configuração.
- **umas_e_ostras/urls.py:** declaração de URLs para o sistema de URLs do Django.
- **umas_e_ostras/wsgi.py:** ponto de entrada para servidores web WSGI hospedarem o site.

O diretório raiz, chamado de umas_e_ostras, armazena o projeto. O nome não faz diferença, já que reflete o nome escolhido pelo desenvolvedor na hora da sua criação. Se for necessário, ele pode ser renomeado.

O arquivo manage.py é um utilitário do Django que faz uma camada sobre o django-admin. Antes de delegar alguma tarefa ao django-admin, ele coloca o pacote do projeto no caminho de busca da linguagem Python, sys.path. Ele configura a variável de ambiente DJANGO_SETTINGS_MODULE, que aponta para o arquivo de configuração settings.py. Além disso, o manage.py executa o método django.setup(), que inicializa várias partes importantes do Django.

- O diretório interno umas_e_ostras é o pacote do projeto. O arquivo __init__.py indica que o diretório é um pacote, conforme já visto anteriormente.
- O arquivo settings.py contém as configurações do projeto, incluindo a configuração do banco de dados e aplicações utilizadas no projeto.
- O sistema de URLs utiliza as definições do arquivo urls.py para associar as URLs às views que responderão as requisições das URLs.

Por fim, o arquivo wsgi.py é o ponto de entrada para servidores web WSGI hospedarem o site.

Configurando o banco

- No arquivo umas_e_ostras/settings.py, o banco é configurado.
- Por padrão, a configuração usa o sqlite3 e um banco chamado db.sqlite3.
- Existem muitas opções de banco, entre elas: ‘django.db.backends.postgresql_psycopg2’, ‘django.db.backends.mysql’.
- É necessário adicionar outras opções para esses bancos: USER, PASSWORD, HOST.

Para configurar o banco de dados, é necessário editar o arquivo settings.py e alterar a variável DATABASES. Por padrão, está configurado o banco de dados sqlite3, abordado na sessão 4. Entre as razões disso, temos justamente a facilidade para iniciar o desenvolvimento sem a preocupação de configuração de bancos de dados mais sofisticados.

Conforme já visto na sessão 4, o Pyhton pode trabalhar com inúmeros bancos de dados. Isso é verdade também para o Django, bastando fazer as configurações corretas. Para outros bancos de dados, provavelmente será necessário usar a opção engine e passar parâmetros adicionais, tais como USER, PASSWORD e HOST.

Aplicações

- Projetos Django são divididos em aplicações.
- Aplicações são independentes entre si e podem ser compartilhadas entre projetos.
- Django possui aplicações padrão que são instaladas junto com o framework.
- Novas aplicações são criadas para resolver tarefas específicas de acordo com os requisitos do sistema.

Os projetos Django são divididos em aplicações. As aplicações são independentes entre si e podem ser compartilhadas com outros projetos. Algumas delas fazem parte do Django e são instaladas junto com o framework. Assim, ao desenvolver um sistema com Django, são usadas aplicações padrão do framework e também criadas novas aplicações para resolver tarefas específicas de acordo com os requisitos do sistema.

É importante tomar algumas providências antes de iniciar o desenvolvimento de aplicações. Entre elas, é fortemente recomendado configurar corretamente o fuso horário local. Isso é feito utilizando a variável TIME_ZONE, que será referenciada por todas as visões e modelos do sistema.

É preciso também habilitar o uso das diversas aplicações padrão disponíveis no Django, e também aplicações distribuídas por terceiros. É bem verdade que algumas aplicações padrão do Django já vêm configuradas para uso. Tudo isso pode ser ajustado através da variável INSTALLED_APPS.

- Antes das aplicações, é recomendado alterar o fuso horário: TIME_ZONE.
- É utilizada a variável INSTALLED_APPS para configurar as aplicações a serem usadas no projeto.
- Por padrão, são habilitadas algumas aplicações pré-instaladas.

Aplicações padrão do Django

As aplicações pré-configuradas do arquivo settings.py servem para facilitar o trabalho de desenvolvimento de um projeto web. Caso não sejam necessárias para um projeto específico, podem ser desabilitadas. São elas:

- **django.contrib.admin:** Site admin. Será detalhada na próxima sessão.
- **django.contrib.auth:** Sistema de autenticação.
- **django.contrib.contenttypes:** Framework para tipos de conteúdos.
- **django.contrib.sessions:** Framework de sessões.
- **django.contrib.messages:** Framework de mensagens.
- **django.contrib.staticfiles:** Framework para gerenciamento de arquivos estáticos.

Uma aplicação particularmente útil é a django.contrib.admin, que implementa uma interface completa para administração de um site, incluindo a manutenção das tabelas do banco com base nas definições do modelo correspondente. Essa aplicação será detalhada na próxima sessão.

Configurando o projeto umas_e_ostras

Antes de ser iniciado o desenvolvimento do projeto, algumas alterações podem ser feitas para adequá-lo às características da região:

- Editar o arquivo umas_e_ostras/settings.py
- Alterar a variável do código de linguagem:
- LANGUAGE_CODE = 'pt-BR'
- Alterar a variável do fuso horário:
- TIME_ZONE = 'America/Sao_Paulo'
- Executar o comando:
- $ python3 manage.py migrate

Uma lista de locais com os possíveis valores de TIME_ZONE pode ser encontrada no endereço: http://en.wikipedia.org/wiki/List_of_tz_database_time_zones

O código do idioma e o fuso horário são as duas variáveis que customizam o projeto para o Brasil. Caso seja necessário, o banco de dados também pode ser configurado, mas neste curso será mantido, por questões de praticidade, o uso do banco sqlite.

Após as configurações, é necessário inicializar o banco com tabelas que são usadas pelas aplicações. Isso é feito com o comando:

```
$ python3 manage.py migrate
```

Servidor de desenvolvimento

- Iniciando o servidor: $ python3 manage.py runserver
- Com IPV6 ::1 porta 8000: $ python3 manage.py runserver -6
- Servidor simples e leve. Não deve ser usado em produção.
- Mudando a porta padrão: $ python3 manage.py runserver 8080
- Permitindo acesso na mesma rede: $ python3 manage.py runserver 0.0.0.0:8000
- O servidor de desenvolvimento é reiniciado automaticamente quando mudanças são salvas, mas precisa ser manualmente reiniciado quando novos arquivos são adicionados.

O Django vem com um servidor de desenvolvimento embutido. Para iniciá-lo, basta utilizar o utilitário de gerenciamento do Django, manage.py, com a opção runserver:

```
$ python3 manage.py runserver
```

O servidor é útil para experimentar o sistema antes de colocá-lo em produção e também dispensa a necessidade de configurar um servidor web mais sofisticado. Por outro lado, é um servidor leve e simples que não deve ser usado em produção.

O servidor utiliza por padrão o endereço localhost: 127.0.0.1 na porta 8000. Para utilizar outra porta, basta informá-la como parâmetro adicional.

```
$ python3 manage.py runserver 8080
```

Pode ser interessante informar também o endereço IP para que sejam aceitas conexões pela rede. Útil para mostrar o sistema funcionando para alguém em outra máquina da rede.

```
$ python3 manage.py runserver 0.0.0.0:8000
```

Para rodar com um endereço IPV6, basta informar o endereço entre colchetes e a porta vem em seguida.

```
$ python3 manage.py runserver [2001:0db8:1234:5678::9]:7000
```

O servidor de desenvolvimento é reiniciado automaticamente quando mudanças são salvas, exceto quando novos arquivos são adicionados. Nesse caso, o servidor deve ser reiniciado manualmente (parar com CTRL-C e iniciar novamente com o comando manage.py runserver).

Criando os modelos

Antes de criar os modelos, criamos uma aplicação:

- $ python3 manage.py startapp reservas

Uma estrutura de arquivos e diretórios será criada:

- admin.py, __init__.py, migrations, models.py, tests.py, views.py
- A filosofia dos modelos é definir o modelo de dados em um lugar e derivar outras coisas a partir desse lugar.

Os modelos do banco de dados só têm sentido dentro de aplicações. Por isso, é preciso primeiro criar uma aplicação para depois criar o modelo do banco a ser utilizado. Seguindo essa lógica, para criar uma aplicação, utilizamos novamente o utilitário manage.py, com a opção startapp e o nome da aplicação. A seguir, o comando para criar a aplicação reservas:

```
$ python3 manage.py startapp reservas
```

É criada uma aplicação com arquivos onde serão implementadas as views (views.py), os modelos (models.py), testes (tests.py) e onde será configurada a interface administrativa do site (admin.py). Também há um diretório chamado migrations, onde serão armazenados históricos de mudanças no modelo.

Modelo da app reservas

- As informações do modelo são utilizadas para várias tarefas do Django.
- Uma das tarefas é criar o esquema do banco de dados (comandos CREATE TABLE).
- Com esses dados é criada uma API para acessar os objetos de cliente e Reserva.
- Para isso é necessário habilitar a aplicação reservas.

A aplicação de reservas terá duas tabelas. A tabela de clientes armazenará o nome, endereço, telefone, e-mail e a data de registro para cada um deles. A tabela de reservas armazenará a data em que a reserva foi feita, a data do evento, quantas pessoas vão ao evento e de qual cliente é a reserva. Isso tudo é feito através do arquivo models.py.

```
from django.db import models

class Cliente(models.Model):
    nome = models.CharField(max_length=200)
    endereco = models.CharField(max_length=200)
    telefone = models.CharField(max_length=20)
    email = models.CharField(max_length=200)
    registrado_em = models.DateTimeField('data do registro')
```

```
class Reserva(models.Model):
    data_reserva = models.DateTimeField('data da reserva')
    data_evento = models.DateTimeField('data do evento')
    pessoas = models.IntegerField(default=0)
    cliente = models.ForeignKey(Cliente)
```

Esse modelo representa a estrutura do banco que será criada a seguir. A figura 6.3 ilustra a estrutura do banco em um diagrama.

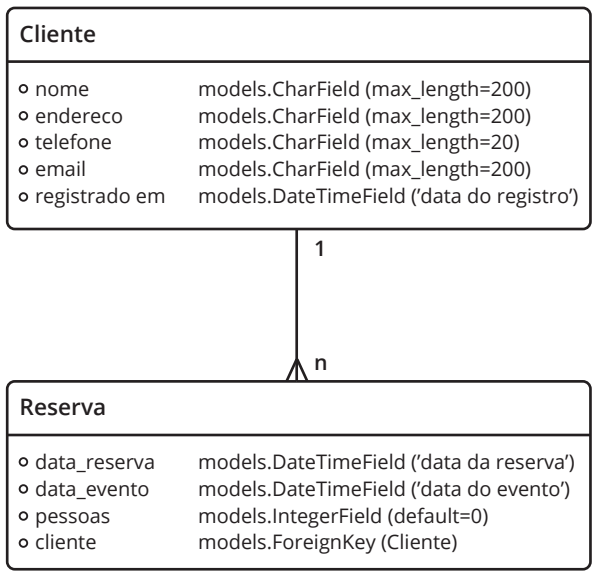

Figura 6.3 Modelo do banco Umas e Ostras.

O modelo do banco é muito importante para fornecer informações ao Django. A partir do modelo, são criadas as tabelas, relacionamentos, sequências, regras de restrição e demais estruturas para armazenar e recuperar dados do banco.

Além disso, é habilitada toda uma API de acesso aos objetos de Cliente e Reserva armazenados no banco. Mas antes de mais nada, é necessário habilitar a aplicação e criar suas tabelas no banco.

Habilitando a aplicação

De forma análoga às aplicações padrão que já vem instaladas com o Django, é necessário habilitar a aplicação no arquivo de configuração settings.py. Na variável INSTALLED_APPS, deve ser adicionada uma nova linha com o nome da classe que configura a aplicação "reservas.apps.ReservasConfig".

```
INSTALLED_APPS = (
    'reservas.apps.ReservasConfig',
    'django.contrib.admin',
    ...
    'django.contrib.staticfiles',
)
```

Aplicando as mudanças

Para informar ao Django que foram feitas mudanças no modelo, rodamos o utilitário de gerenciamento do Django com a opção makemigrations.

```
$ python3 manage.py makemigrations reservas
```

Esse comando gera um arquivo no diretório migrations com as mudanças feitas no modelo. E será usado para aplicar as mudanças. O arquivo gerado pode ser lido e alterado, sendo que seu nome é gerado automaticamente e segue uma numeração sequencial. Na primeira vez que o comando é executado, será gerado um arquivo 'reservas/migrations/0001_initial.py'.

Para ver as mudanças que serão aplicadas ao banco, é usado o comando sqlmigrate.

```
$ python3 manage.py sqlmigrate reservas 0001
```

Não é necessário rodar o comando sqlmigrate todas as vezes. Para aplicar as mudanças no banco, basta rodar o makemigrations e em seguida o comando migrate:

```
$ python3 manage.py migrate
```

Atividades Práticas AVA

Utilizando a API

- É possível usar a API do Django em um shell.
  - $ python3 manage.py shell
- Esse comando já prepara o ambiente configurado pelo arquivo settings.py
- Dessa forma, a conexão ao banco, fuso horário e aplicações são configuradas. E os comandos executados no shell utilizam essas configurações.

O utilitário de gerenciamento do Django possui uma opção que abre um shell interativo já utilizando as configurações do arquivo settings.py e com o caminho do projeto disponível para acessar os arquivos do sistema.

Esse shell nos permite explorar a API do banco de dados para conhecer alguns métodos do ORM do Django.

Explorando a API

Nada melhor do que trabalhar diretamente no computador para assimilar os conhecimentos sendo passados. Assim, vamos carregar o shell e prosseguir com os exemplos a partir dele.

```
$ python3 manage.py shell
```

Importante notar que os exemplos apresentados a seguir precisam ser executados dentro de um shell. A tentativa de executá-los fora do shell vai gerar erros. Veja o código a seguir e vá executando os comandos no seu computador, comparando os resultados obtidos.

```
# /usr/bin/env python3
# modelo01.py
“””
Esse exemplo precisa ser executado manualmente dentro do
shell do Django, iniciado com:
```

```
$ python3 manage.py shell

>>> from reservas.models import Cliente, Reserva
>>> Cliente.objects.all()
[]

O arquivo de configuração habilita o uso do fuso horário
por padrão. Isso faz com que as datas esperem por informação
de fuso horário. É recomendado usar timezone.now(), que já
retorna a data com informação de fuso.

>>> from django.utils import timezone

>>> cliente = Cliente(
...     nome = 'Robson dos Santos',
...     endereco = 'Rua da Vila Algemiro, 20, Bairro do Pescador',
...     telefone = '(51) 4334-4562',
...     email = 'robson@dossantos.fut.br',
...     registrado_em = timezone.now()
)
>>> cliente.save()
>>> cliente.id
1
>>> cliente.nome
'Robson dos Santos'
>>> cliente.registrado_em
datetime.datetime(2015, 4, 20, 0, 19, 30, 974174, tzinfo=<UTC>)

>>> cliente.email = 'robson@santos.fut.br'
>>> cliente.save()
>>> Cliente.objects.all()
[<Cliente: Cliente object>]
""""
```

Os primeiros comandos do modelo01.py importam as classes do modelo Cliente e Reserva.

```
>>> from reservas.models import Cliente, Reserva
>>> Cliente.objects.all()
[]
```

Com o método all() do atributo objects são recuperados todos os registros ou instâncias de cliente armazenados no banco de dados. Como ainda não existe nenhum registro no banco, a resposta é uma lista vazia.

Não custa lembrar que ao iniciar o shell as configurações do projeto são habilitadas. Assim, o fuso horário definido no arquivo settings.py passa a ser a referência, permitindo usar o módulo timezone para lidar com datas.

```
>>> from django.utils import timezone

>>> cliente = Cliente(
```

```
...     nome = 'Robson dos Santos',
...     endereco = 'Rua da Vila Algemiro, 20, Bairro do Pescador',
...     telefone = '(51) 4334-4562',
...     email = 'robson@dossantos.fut.br',
...     registrado_em = timezone.now()
)
```

Para criar uma instância de cliente, passamos os nomes dos campos para a classe Cliente. O campo da data registrado_em é preenchido com a data e hora atual usando a função now(), do módulo timezone. Essa função já retorna a data e hora atual com informações levando em consideração o fuso.

```
>>> cliente.save()
>>> cliente.id
1
>>> cliente.nome
'Robson dos Santos'
>>> cliente.registrado_em
datetime.datetime(2015, 4, 20, 0, 19, 30, 974174, tzinfo=<UTC>)
```

Ao salvar o objeto cliente no banco, a chave primária ("id") é automaticamente preenchida. Esse campo não precisou ser informado no nosso modelo, pois o Django gera uma chave primária automaticamente quando ela não é informada no modelo. Para acessar os valores das colunas, usamos os atributos da instância de cliente.

É possível reparar que a hora da data é calculada baseada na hora do sistema, no fuso horário configurado no arquivo settings.py e na diferença para o fuso horário de referência, UTC. Assim, no caso do Brasil, a diferença é de três horas a menos (eventualmente desconsiderando o horário de verão). No exemplo, o comando foi executado no dia 19 de abril de 2015, às 21:19, sendo convertida para a hora corrente no fuso horário de referência UTC, ou seja, 20 de abril de 2015 às 0:19h do dia seguinte.

```
>>> cliente.email = 'robson@santos.fut.br'
>>> cliente.save()
>>> Cliente.objects.all()
[<Cliente: Cliente object>]
```

Para realizar uma alteração no banco, basta mudar o valor da propriedade no objeto correspondente e executar o método save(). Uma nova consulta mostra que existe um registro no banco.

A representação do objeto

- O Django usa a representação do objeto em vários lugares.
- Um dos lugares é no shell.
- A representação string também é utilizada na interface administrativa, admin, gerada automaticamente a partir do modelo.
- É possível adicionar métodos customizados ao modelo.

Por padrão, os objetos de classes que fazem herança do model do Django são representados pelo nome da classe seguidos da palavra object. Isso pode confundir um pouco, pois não permite diferenciar um objeto de outro de forma fácil. De qualquer modo, essa representação é utilizada no shell e a interface administrativa também utiliza essa representação em listagens.

Para contornar o problema da representação dos objetos, pode ser implementado o método __str__. Ao fazer isso, qualquer acesso à instância do objeto vai usar o retorno do método para apresentar uma representação mais significativa do objeto. Vejamos isso no exemplo a seguir:

```
import datetime

from django.db import models
from django.utils import timezone

class Cliente(models.Model):
    nome = models.CharField(max_length=200)
    endereco = models.CharField(max_length=200)
    telefone = models.CharField(max_length=20)
    email = models.CharField(max_length=200)
    registrado_em = models.DateTimeField('data do registro')

    def __str__(self):
        return self.nome

    def registro_eh_antigo(self):
        um_ano = timezone.now() - datetime.timedelta(days=365)
        return self.registrado_em < um_ano


class Reserva(models.Model):
    data_reserva = models.DateTimeField('data da reserva')
    data_evento = models.DateTimeField('data do evento')
    pessoas = models.IntegerField(default=0)

    cliente = models.ForeignKey(Cliente)

    def __str__(self):
        return '{} - {}'.format(
            self.cliente, str(self.data_evento))
```

Método customizado

- Foi importado o módulo datetime padrão para datas da linguagem Python.
- Foi importado o módulo django.utils.timezone padrão de fuso horário do Django.
- São usados no método customizado para calcular a diferença de um ano.
- Abrir o shell novamente para testar:
  - $ python3 manage.py shell

Outros métodos podem também ser adicionados à classe para customizá-la de acordo com as necessidades do sistema. É o caso do método registro_eh_antigo no exemplo anterior, que faz um cálculo com datas para ver se o registro do cliente foi anterior a um ano. Foram utilizadas as funções timezone.now() e datetime.timedelta() para verificar se há a diferença de um ano em relação à data de registro. Caso exista um ano de diferença, o método retorna True; caso contrário, retorna False. No processo de criação desses métodos recomenda-se usar o shell do Django para testar se está tudo certo.

Filtros e objetos relacionados

O ORM do Django permite o uso de filtros para delimitar os registros que serão usados. Esses métodos podem ser testados no shell do Django.

```
# /usr/bin/env python3
# modelo02.py

“””
Esse exemplo precisa ser executado manualmente dentro do
shell do Django, iniciado com:

$ python3 manage.py shell

>>> from reservas.models import Cliente, Reserva
>>> Cliente.objects.all()
[<Cliente: Robson dos Santos>]
>>> Cliente.objects.filter(id=1)
[<Cliente: Robson dos Santos>]
>>> Cliente.objects.filter(nome__startswith=’Rob’)
[<Cliente: Robson dos Santos>]
>>> from django.utils import timezone
>>> ano_atual = timezone.now().year
>>> Cliente.objects.get(registrado_em__year=ano_atual)
<Cliente: Robson dos Santos>
>>> Cliente.objects.get(id=2)
Traceback (most recent call last):
    ...
DoesNotExist: Cliente matching query does not exist.
>>> Cliente.objects.get(pk=1)
<Cliente: Robson dos Santos>
>>> cliente = Cliente.objects.get(pk=1)
>>> cliente.registro_eh_antigo()
False
>>> cliente = Cliente.objects.get(pk=1)
>>> cliente.reserva_set.all()
[]
>>> fuso = timezone.get_current_timezone()
>>> data = timezone.datetime(2015, 5, 20, 11, 30, tzinfo=fuso)
>>> data_utc = data.astimezone(timezone.utc)
>>> data_utc
datetime.datetime(2015, 5, 20, 14, 30, tzinfo=<UTC>)
```

```
>>> cliente.reserva_set.create(
...     data_reserva = timezone.now(),
...     data_evento = data_utc,
...     pessoas = 12)
<Reserva: Robson dos Santos - 2015-05-20 14:30:00+00:00>
>>> data = timezone.datetime(2015, 6, 14, 23, 30, tzinfo=fuso)
>>> data_utc = data.astimezone(timezone.utc)
>>> cliente.reserva_set.create(
...     data_reserva = timezone.now(),
...     data_evento = data_utc,
...     pessoas = 8)
<Reserva: Robson dos Santos - 2015-06-15 02:30:00+00:00>
>>> data = timezone.datetime(2015, 8, 11, 21, 30, tzinfo=fuso)
>>> data_utc = data.astimezone(timezone.utc)
>>> reserva = cliente.reserva_set.create(
...     data_reserva = timezone.now(),
...     data_evento = data_utc,
...     pessoas = 4)
>>> reserva.cliente
<Cliente: Robson dos Santos>
>>> cliente.reserva_set.all()
[<Reserva: Robson dos Santos - 2015-05-20 14:30:00+00:00>,
 <Reserva: Robson dos Santos - 2015-06-15 02:30:00+00:00>,
 <Reserva: Robson dos Santos - 2015-08-12 00:30:00+00:00>]
>>> cliente.reserva_set.count()
3
>>> Reserva.objects.filter(cliente__registrado_em__year=ano_atual)
[<Reserva: Robson dos Santos - 2015-05-20 14:30:00+00:00>,
 <Reserva: Robson dos Santos - 2015-06-15 02:30:00+00:00>,
 <Reserva: Robson dos Santos - 2015-08-12 00:30:00+00:00>]
>>> reserva = cliente.reserva_set.filter(data_evento=data_utc)
>>> reserva.delete()
“””
```

Normalmente, ao desenvolver sistemas, é recomendado evitar o uso do método all(). Ele não aplica filtro algum sobre os registros e isso pode trazer um número elevado de dados, causando lentidão.

```
>>> from reservas.models import Cliente, Reserva
>>> Cliente.objects.all()
[<Cliente: Robson dos Santos>]
>>> Cliente.objects.filter(id=1)
[<Cliente: Robson dos Santos>]
>>> Cliente.objects.filter(nome__startswith='Rob')
[<Cliente: Robson dos Santos>]
```

O método filter() aplica diversos tipos de filtros e pode ser utilizado com o nome do campo e um valor, como no exemplo Cliente.objects.filter(id=1). O ORM do Django usa uma sintaxe específica para realizar consultas.

É comum utilizar o atributo seguido de dois caracteres sublinhado e uma das várias palavras-chave indicando diferentes possibilidades de pesquisas. No exemplo temos nome__startswith='Rob', que busca registros cuja coluna nome tenha o seu conteúdo iniciado por Rob.

```
>>> from django.utils import timezone
>>> ano_atual = timezone.now().year
>>> Cliente.objects.get(registrado_em__year=ano_atual)
<Cliente: Robson dos Santos>
>>> Cliente.objects.get(id=2)
Traceback (most recent call last):
    ...
DoesNotExist: Cliente matching query does not exist.
```

Outro método fornecido pelo ORM é o método get(), que também aceita pesquisas como o filter(). No exemplo, é aplicado um filtro retornando os clientes registrados no ano atual. A diferença é que o método retorna apenas um resultado, e se não houver registro que atenda ao filtro ocorre uma exceção.

```
>>> Cliente.objects.get(pk=1)
<Cliente: Robson dos Santos>
>>> cliente = Cliente.objects.get(pk=1)
>>> cliente.registro_eh_antigo()
False
>>> cliente = Cliente.objects.get(pk=1)
>>> cliente.reserva_set.all()
[]
```

Outra forma de recuperar um registro é usar a palavra-chave pk, que filtrará os resultados pela chave primária. A instância do objeto pode utilizar o método que foi definido no modelo. Através da API do Django, podemos acessar a tabela relacionada de reservas do cliente. Utilizamos um nome padrão, que é o nome da tabela seguida por _set. No exemplo, temos cliente.reserva_set. Ainda não há nenhum registro cadastrado, e por isso o método all() retorna uma lista vazia.

```
>>> fuso = timezone.get_current_timezone()
>>> data = timezone.datetime(2015, 5, 20, 11, 30, tzinfo=fuso)
>>> data_utc = data.astimezone(timezone.utc)
>>> data_utc
datetime.datetime(2015, 5, 20, 14, 30, tzinfo=<UTC>)
>>> cliente.reserva_set.create(
...     data_reserva = timezone.now(),
...     data_evento = data_utc,
...     pessoas = 12)
<Reserva: Robson dos Santos - 2015-05-20 14:30:00+00:00>
```

A utilização de fuso horário está habilitada no arquivo de configuração; portanto, os campos de data esperam que as datas possuam informação de fuso. O código do exemplo anterior apresenta uma maneira de utilizar o fuso e transformar em um horário de referência UTC. Em seguida, é criado um registro de reserva já relacionado ao cliente e utilizando a data recém-criada com informação de fuso horário.

```
>>> reserva.cliente
<Cliente: Robson dos Santos>
>>> cliente.reserva_set.all()
[<Reserva: Robson dos Santos - 2015-05-20 14:30:00+00:00>,
 <Reserva: Robson dos Santos - 2015-06-15 02:30:00+00:00>,
 <Reserva: Robson dos Santos - 2015-08-12 00:30:00+00:00>]
>>> cliente.reserva_set.count()
3
```

São criados mais alguns registros de reserva, e então o exemplo apresenta a maneira de acessar o registro relacionado de cliente a partir da reserva, com reserva.cliente. Ao consultar os registros de reserva, novamente são retornados os registro que acabaram de ser criados. O método count() mostra a quantidade de registros de reserva cadastrados.

```
>>> Reserva.objects.filter(cliente__registrado_em__year=ano_atual)
[<Reserva: Robson dos Santos - 2015-05-20 14:30:00+00:00>,
 <Reserva: Robson dos Santos - 2015-06-15 02:30:00+00:00>,
 <Reserva: Robson dos Santos - 2015-08-12 00:30:00+00:00>]
>>> reserva = cliente.reserva_set.filter(data_evento=data_utc)
>>> reserva.delete()
```

A API oferece a possibilidade de seguir o relacionamento utilizando sublinhado duplo ("__"). No exemplo, acessamos o ano do atributo registrado_em da coluna cliente para filtrar as reservas daqueles que se registraram no ano atual. Em seguida, é recuperada a reserva, cuja data do evento é igual à última inserida, e deletamos o registro.

Atividades Práticas 

7

A aplicação de administração – Admin

objetivos

Uma vez instalado e configurado o framework Django, é preciso entender como se administra o ambiente para começar de fato a desenvolver aplicações. Nesta sessão, o aluno vai conhecer cada uma dessas etapas, continuando a desenvolver uma aplicação exemplo que começou a ser criada na sessão anterior.

conceitos

lista de alteração, fields, fieldsets, StackedInLine, TabularInLine, admin_order_field, short_description, list_filter e search_fields.

Administrando sites

Uma das tarefas mais importantes na administração de sites é a atualização de conteúdo, principalmente em sites de notícias, ambiente onde o Django foi criado. Por isso, uma das preocupações dos seus criadores foi com a interface de administração do site, justamente para facilitar o processo de criar aplicações em ritmo acelerado para atender a demanda do jornal. Foi então criada uma aplicação para automatizar a administração do site, para ser usada pela equipe interna do jornal.

- Django automatiza a administração do site.
- Foi criado em um ambiente com clara separação entre gerentes de conteúdo e público visitante.
- Permite que os gerentes de conteúdo adicionem, consultem, alterem e excluam registros do banco de dados.
- A aplicação é chamada de admin e foi projetada para administradores do site (não para o público).

No jornal existe uma separação bem definida entre quem alimenta informações, notícias e conteúdo e quem é apenas um visitante do site. A interface de administração é gerada automaticamente pela aplicação admin e permite que os gerentes de conteúdo atualizem e consultem registros das tabelas representadas pelos modelos. Essa aplicação foi projetada para os administradores do site e não deve ser exposta para o público visitante do site.

Primeiros passos

A primeira coisa a ser feita para utilizar a interface de administração é criar um usuário administrativo, o equivalente a um superusuário, que tem todas as permissões necessárias, inclusive para criar usuários e grupos e definir suas permissões.

- Criar um usuário administrativo.
  - # python manage.py createsuperuser
- Verificando o admin.
  - # python manage.py runserver
- Acessando o admin.
  - http://localhost:8000/admin
- Deve aparecer um formulário pedindo usuário e senha.

Para criar o usuário administrativo é usado o utilitário de gerenciamento do Django com o comando createsuperuser, conforme a seguir:

```
# python manage.py createsuperuser
```

São solicitados um nome de usuário, um e-mail e uma senha. O passo seguinte, para acessar a interface de administração, é iniciar o servidor de desenvolvimento, através do comando:

```
# python manage.py runserver
```

Caso o servidor de desenvolvimento tenha sido executado da forma padrão, ou seja, host=localhost e porta=8000, a url utilizada para acessar a interface administrativa é http://localhost:8000/admin.

Figura 7.1
Autenticação do administrador.

Ao acessar a interface administrativa é solicitado um usuário e uma senha, conforme podemos ver an Figura 7.1. Usar o nome e senha do superusuário criado.

Figura 7.2
Administração do site.

Quando o login é efetuado, a página inicial da interface administrativa apresenta as tabelas registradas para administração, conforme a figura 7.2.7.2 Habilitando o admin das aplicações.

As únicas tabelas disponíveis automaticamente são as tabelas de usuários e grupos que fazem parte do framework de autenticação do Django.

- Ao acessar o admin é possível editar tabelas de grupos e usuários do framework de autenticação do Django.
- Para que as tabelas da aplicação apareçam no admin, é necessário registrá-las no arquivo admin.py

As tabelas das aplicações desenvolvidas precisam ser registradas no sistema de administração do site para serem exibidas nessa lista de tabelas da interface administrativa.

Isso é feito no arquivo admin.py da aplicação. Ele fica no mesmo diretório onde está o arquivo models.py.

```
# reservas/admin.py

from django.contrib import admin
from .models import Cliente

admin.site.register(Cliente)
```

O código do exemplo anterior faz a importação da aplicação admin e o modelo de clientes "Cliente", providenciando o registro do modelo na interface de administração.7.2.1.

Explorando o admin

Figura 7.3
Registro de classe no Admin

Basta atualizar a página inicial do admin para que a tabela de clientes seja exibida na lista de tabelas passíveis de serem administradas, conforme a figura 7.3. Clicando no nome da classe, cliente, é acessada a lista de alteração que permite gerenciar os dados da tabela de clientes. Como a tabela já possuía alguns dados registrados na sessão anterior, eles já aparecem na listagem, conforme pode ser visto na figura 7.4.

Figura 7.4
Lista de alteração de clientes.

A lista de alteração possui um botão para adicionar novos registros, destacado na figura 7.5 a seguir. Ao ser clicado é apresentado um formulário com os campos vazios para cadastro de um novo cliente. Esse mesmo formulário é apresentado quando se clica em um dos registros já existentes, mas nesse caso trazendo os dados daquele registro e assim permitindo a alteração dos mesmos, conforme o exemplo da figura 7.6. Em ambos os casos o Django consulta os tipos de dados do modelo e apresenta os campos apropriados para cada coluna/atributo.

Figura 7.5
Botão para adicionar cliente.

Figura 7.6
Formulário de inclusão/alteração.

Customizando formulários

- A ordem de apresentação dos campos pode ser alterada.
- Quando o formulário se torna muito grande, é possível agrupar conjuntos de campos e utilizar esquemas alternativos de apresentação, por exemplo, usando tabelas.
- Os conjuntos de campos podem ser ocultados ou exibidos.
- Outros objetos relacionados ao modelo podem ser incluídos ou alterados em um mesmo formulário.
- A representação de campos, por padrão feita através de strings, pode ser alterada.
- Etc.

Nem sempre o formulário gerado automaticamente é o ideal para atender as necessidades dos gerentes de conteúdo. A boa notícia é que possível adaptar esses formulários sempre que necessário.

O atributo fields da classe responsável pela administração de clientes permite que sejam definidos os campos que devem aparecer no formulário de administração do modelo. Não existe uma regra ou restrição para dar nome para essa classe "administradora". O mais comum é usar o nome do modelo, nesse caso cliente, seguido da palavra Admin: ClienteAdmin. Assim, em qualquer lugar no código onde for usada a classe ClienteAdmin, fica claro qual o modelo está sendo "administrado".

No trecho de código a seguir, observe a ordem em que os campos são colocados nessa lista que define o atributo fields. Essa será a ordenação usada para gerar o formulário de inclusão ou alteração conforme, podemos ver na figura 7.7.

```
# reservas/admin.py
from django.contrib import admin
from .models import Cliente


class ClienteAdmin(admin.ModelAdmin):
    fields = [‘nome’, ‘email’, ‘telefone’,
              ‘endereco’, ‘registrado_em’]


admin.site.register(Cliente, ClienteAdmin)
```

Figura 7.7
Customização do formulário de clientes.

Para organizar melhor o formulário, também é possível separar seus campos em seções distintas. Para isso é utilizado o atributo fieldsets, que é uma lista de tuplas. Cada uma dessas tuplas têm dois elementos, sendo o primeiro o título da sessão (para deixar em branco usar o nulo do Python, None), e o segundo um dicionário. A chave fields é associada a uma lista de campos que deve aparecer na seção correspondente. A seguir, é apresentado um trecho de código onde é inicializado o atributo fieldsets, com o formulário resultante sendo apresentado na figura 7.8.

```
# reservas/admin.py
from django.contrib import admin
from .models import Cliente


class ClienteAdmin(admin.ModelAdmin):
    fieldsets = [
        (None,  {
            ‘fields’: [‘nome’, ‘email’, ‘telefone’, ‘endereco’]}),
        (‘Datas’, {
            ‘fields’: [‘registrado_em’]}),
    ]

admin.site.register(Cliente, ClienteAdmin)
```

Figura 7.8
Uso de fieldsets e a seção Datas.

Uma classe com muitos atributos pode gerar um formulário muito grande, sendo ainda possível fazer com que seções sejam ocultadas ou exibidas. Para fazer uso desse recurso é preciso incluir a chave "classes" com o valor "collapse" na tupla da seção em fieldsets que poderá ser recolhida ou expandida. Tomando como base o trecho de código anterior, veja como fica o uso desse recurso para tornar a seção Datas colapsível:

```
class ClienteAdmin(admin.ModelAdmin):
    fieldsets = [
        (None,  {
            'fields': ['nome', 'email', 'telefone', 'endereco']}),
        ('Datas', {
            'fields': ['registrado_em'],
            'classes': ['collapse']}),
    ]
```

O reflexo disso no formulário é que a seção Datas passa a ser apresentada com um link clicável entre parênteses indicando que pode ser mostrada, conforme a figura 7.9.

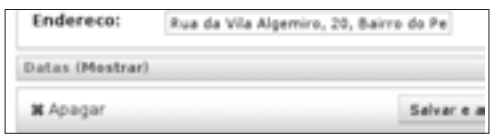

Figura 7.9
Seção datas está escondida.

Quando a seção "Datas" está sendo exibida, o link no cabeçalho passa a indicar que ela pode ser escondida, conforme vemos na figura 7.10.

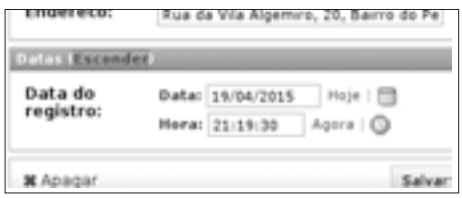

Figura 7.10
Seção datas é exibida.

Objetos relacionados

- Ao registrar o modelo relacionado, o formulário gerado permite a inclusão e edição de objetos relacionados diretamente no formulário.
- São apresentados dois ícones para edição e inclusão de objetos relacionados ao lado do combo que apresenta a lista de objetos.

Uma aplicação provavelmente trabalhará com diversas classes, todas elas devendo ser registradas na aplicação de administração do site. Se uma classe se relaciona com outra, é possível gerar um formulário que exiba ícones para inclusão e alteração de registros que possuem chave estrangeira para outras tabelas.

Ao registrar a classe Reserva na aplicação de administração, os ícones serão exibidos logo ao lado do combo de clientes, no formulário de reservas, como pode ser visto na figura 7.11.

Figura 7.11 Formulário de administração de reservas.

É muito simples tirar proveito dessa funcionalidade. Basta registrar a classe Reserva na aplicação de administração que esse comportamento será habilitado. Veja o trecho de código a seguir onde isso é feito, refletindo também na lista de administração, que passará a exibir Reservas, como pode ser visto na figura 7.12.

```
# reservas/admin.py

from django.contrib import admin
from.models import Reserva, Cliente

class ClienteAdmin(admin.ModelAdmin):
    fieldsets = [
        (None,  {
            'fields': ['nome', 'email', 'telefone', 'endereco']}),
        ('Datas', {
            'fields': ['registrado_em'],
            'classes': ['collapse']}),
    ]


admin.site.register(Cliente, ClienteAdmin)
admin.site.register(Reserva)
```

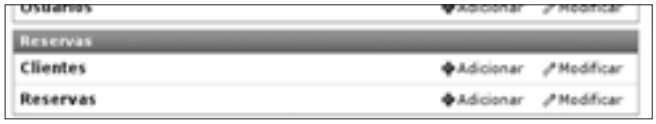

Figura 7.12 Lista de administração com clientes e reservas.

Nada acontecerá com o formulário de clientes. Apenas o formulário de reservas será afetado, como pode ser visto na figura 7.11, apresentada anteriormente.

Em alguns casos, é mais interessante permitir que seja incluído não apenas uma instância da classe relacionada, e sim um conjunto de instâncias. O exemplo de aplicação que estamos usando nesta sessão se presta a isto, pois pode ser interessante permitir que várias reservas sejam registradas ou consultadas no formulário de clientes.

- É possível adicionar várias reservas a partir do formulário de clientes.
- A classe StackedInLine de Admin é usada, informando a tabela/classe (model = Reserva) e quantas linhas extras devem ser criadas.
- Serão apresentados os registros preexistentes e um número adicional de linhas em branco para registrar novas reservas.

Deve ser criada uma classe derivada da classe admin.StackedInLine, inicializando dois atributos: model e extra. O atributo model define qual tabela/classe servirá de modelo ao ser apresentado no formulário principal. O atributo extra indica quantos itens em branco devem ser apresentados no formulário principal.

Portanto, para permitir a inclusão de várias reservas de um mesmo cliente, deve ser derivada uma classe de admin.StackedInLine e inicializados os respectivos atributos. Isso pode ser visto no trecho de código a seguir, com a criação da classe ReservasInline:

```
# reservas/admin.py
from django.contrib import admin
from .models import Reserva, Cliente

class ReservasInline(admin.StackedInline):
    model = Reserva
    extra = 3

class ClienteAdmin(admin.ModelAdmin):
    fieldsets = [
        (None,  {
            'fields': ['nome', 'email', 'telefone', 'endereco']}),
        ('Datas', {
            'fields': ['registrado_em'],
            'classes': ['collapse']}),
    ]
    inlines = [ReservasInline]

admin.site.register(Cliente, ClienteAdmin)
```

Na classe ClienteAdmin, deve ser inicializado o atributo inlines com uma lista de classes de modelos relacionados. Caso o modelo Cliente tivesse mais relacionamentos, mais itens poderiam ser adicionados nessa lista. No exemplo, inlines é inicializado apenas com a classe de ReservasInLine. O resultado pode ser visto na figura 7.13.

Figura 7.13
Reservas no formulário de clientes.

Na figura 7.13 percebe-se que à disposição dos campos de cada reserva, um a seguir do outro, não faz o melhor uso do espaço disponível. Existe a possibilidade de utilizar o modo de exibição tabular, onde os campos do modelo são apresentados em colunas. Esse layout é obtido estendendo a classe admin.TabulaInLine, conforme vemos a seguir:

```
# reservas/admin.py
...

class ReservasInline(admin.TabularInline):
    model = Reserva
    extra = 3

class ClienteAdmin(admin.ModelAdmin):
...
```

A mudança no código é pequena, bastando mudar a classe base da classe ReservasInline. Já o efeito na apresentação do formulário pode ser grande, conforme podemos constatar na figura 7.14.

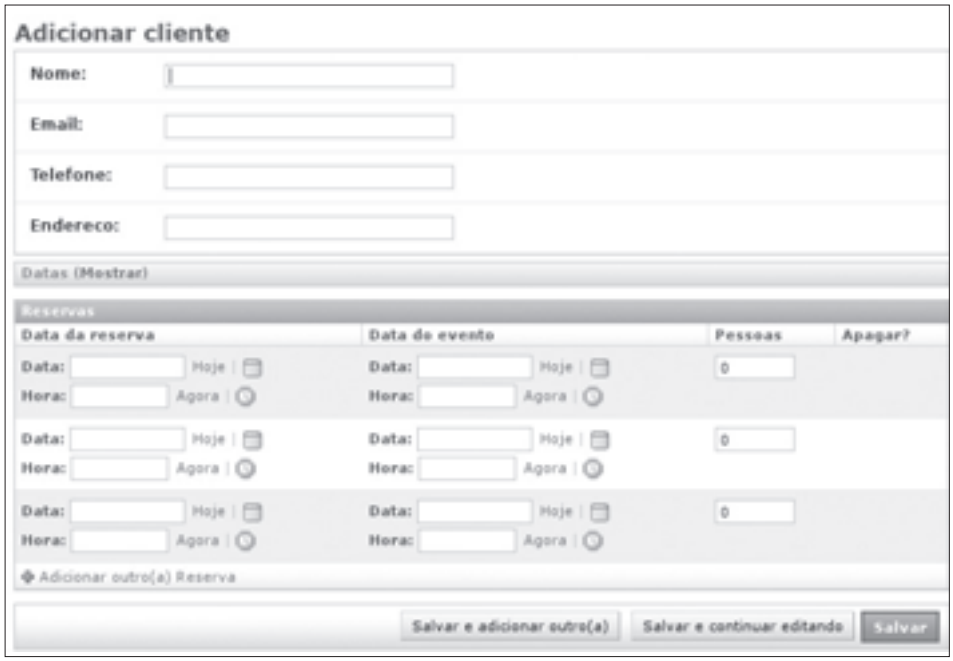

Figura 7.14
Uso de TabulaInLine para Reservas.

Customizando a lista de alteração

- A lista de alteração apresenta os objetos usando o padrão do Python ou, quando é implementada, a representação string desses objetos.
- O atributo list_display permite alterar esse padrão.
- É possível permitir que a ordenação dos objetos seja alterada conforme se clica no título da coluna, mas isso exige alterações em models.py.

Além dos formulários de administração, a lista de alteração pode ser customizada também. A lista apresenta o objeto usando o padrão do Python ou, quando é implementada, a representação string do objeto. Isso pode ser alterado através do atributo list_display, que deve conter os nomes dos campos alterados.

No trecho de código a seguir, os campos nome, e-mail e o método registro_eh_antigo serão exibidos na listagem de alteração. Além dos campos, podem ser adicionados métodos nessa tupla, como é o caso do método registro_eh_antigo.

```
# reservas/admin.py
...

class ClienteAdmin(admin.ModelAdmin):
    list_display = (‘nome’, ‘email’, ‘registro_eh_antigo’)

    fieldsets = [
...
```

O resultado pode ser observado na figura 7.15

Figura 7.15 Listagem de alteração de clientes.

Na listagem de alteração, ao clicar no cabeçalho das colunas onde são apresentados os campos, pode ser alterada a ordenação dos resultados (isso não acontece em colunas com métodos). Para habilitar esse comportamento, é necessário fazer alguns ajustes em models.py, adicionando atributos ao método. Isso pode ser visto no trecho de código a seguir.

```
# reservas/models.py
...

class Cliente(models.Model):
...
    def __str__(self):
        return self.nome

    def registro_eh_antigo(self):
        um_ano = timezone.now() - datetime.timedelta(days=365)
        return self.registrado_em < um_ano
    registro_eh_antigo.admin_order_field = 'registrado_em'
    registro_eh_antigo.boolean = True
    registro_eh_antigo.short_description = 'Cliente antigo?'
```

O atributo admin_order_field indica o nome de um campo a ser usado na ordenação. Já o atributo boolean indica se o método deve ser tratado como um booleano e em vez de aparecer o "valor" True ou False na listagem, o conteúdo correspondente será representado através de um ícone que remete a verdadeiro ou falso. Finalmente, o atributo short_description serve para alterar o título da coluna, que por padrão é o nome do próprio método com os caracteres sublinhados ("_") substituídos por espaço. O resultado de todos esses ajustes pode ser visto na figura 7.16.

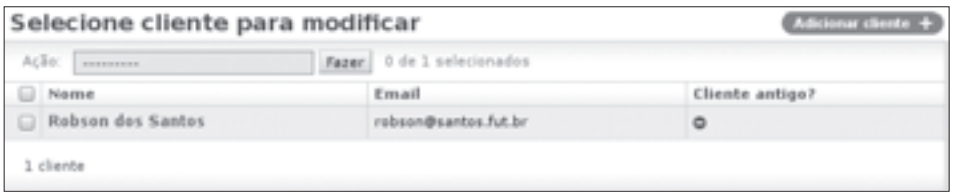

Figura 7.16 Customizando métodos na listagem de administração.

Filtros e pesquisa

- O atributo list_filter permite que um filtro seja aplicado à lista.
- O atributo search_fields adiciona uma pesquisa por campo(s).

Em princípio, uma lista de alteração vai exibir todos os registros/objetos de uma determinada tabela/classe. Existe, contudo, a possibilidade de filtrar e pesquisar determinados elementos da lista.

O atributo list_filter fará com que uma caixa com opções de filtro seja apresentada ao lado direito da listagem de alteração. Ao clicar em uma opção, os resultados serão filtrados de acordo com a opção selecionada. O atributo search_fields faz com que seja apresentado um campo de texto para que seja informado um termo de pesquisa. Essa pesquisa usa o operador LIKE do banco de dados, que pode resultar em uma pesquisa muita lenta se diversos campos e condições forem estipulados. Assim, é importante ter cuidado ao utilizar esse recurso. No trecho de código a seguir, esses dois atributos são setados.

```
# reservas/admin.py
...

class ClienteAdmin(admin.ModelAdmin):
    list_display = ('nome', 'email', 'registro_eh_antigo')

    list_filter = ['registrado_em']

    search_fields = ['nome']

    fieldsets = [
...
```

O resultado obtido é apresentado na figura 7.17, onde o campo registrado_em foi definido como filtro. Por ser um campo do tipo data, são apresentadas algumas opções pré-definidas: hoje, Últimos 7 dias etc. Para um campo do tipo booleano, as opções seriam Sim e Não ou, em inglês, Yes e No.

Figura 7.17
Filtro e pesquisa na lista de clientes.

Existem inúmeras outras possibilidades de customização no Admin, inclusive a aparência do próprio aplicativo. Para quem tiver interesse, acesse a documentação do Django ou consulte links recomendados de leituras complementares no AVA.

Atividades Práticas

8

Trabalhando com a camada de visão

objetivos

Nesta sessão, a aplicação das sessões anteriores continua a evoluir, fazendo uso dos recursos que o Django disponibiliza através do seu sistema de URLs, views e templates. São criadas e renderizadas novas views, aprende-se a tratar erros e a utilizar formulários, muitas vezes ajustando e ampliando o conteúdo de arquivos especiais utilizados pelo Django e por aplicações web.

conceitos

HTTP; Sistema de URLs; Views e templates; Tratamento de erros; Atalhos; Formulários e namespaces.

Protocolo HTTP

Em geral, qualquer rede de computadores funciona utilizando definições, padrões de comunicação que são conhecidos pelo termo protocolo. Existem inúmeros protocolos de rede, que cuidam de diferentes camadas do processo de troca de informações. Os protocolos de mais baixo nível são conhecidos apenas por especialistas em redes de computador, enquanto os protocolos das camadas superiores podem ser percebidos por um número maior de pessoas, inclusive usuários da internet.

Entre esses últimos temos os protocolos de comunicação e transferência de arquivos, tais como FTP, SMTP, POP e IMAP. Um dos mais conhecidos e utilizados é o protocolo HTTP, uma sigla que, se traduzida, significa Protocolo de Transferência de Hiper Texto. O HTTP funciona como um protocolo de requisição e resposta em um modelo cliente-servidor. Um computador cliente envia uma requisição, por exemplo, através de um navegador de internet para um servidor. O computador servidor processa a requisição recebida, prepara e envia uma resposta para o cliente. É assim que funciona o protocolo HTTP, conforme ilustrado na figura 8.1.

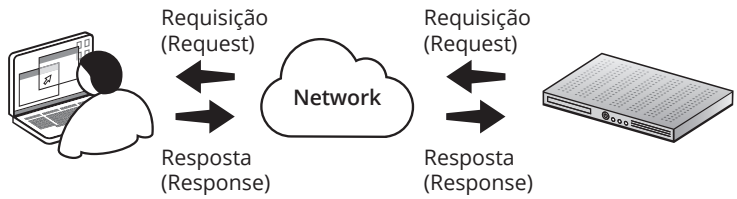

Figura 8.1
O protocolo HTTP.

É importante ter em mente o funcionamento do HTTP para ter melhor entendimento do funcionamento da camada de visão do Python que apresentamos em mais detalhes ao longo desta sessão.

Conceito de view

- As views são responsáveis por atender requisições do usuário.
- São funções específicas, normalmente associadas a algum template.
- É muito comum que acessem o banco de dados, para leitura ou escrita.
- As funções ou métodos são associados a uma URL, usando o sistema de URLs do Django.

As views no Django são associadas à funcionalidades específicas. Cada view atende um determinado propósito através das requisições de usuário. A resposta da requisição normalmente é preparada utilizando um template e, na maioria das vezes, efetua uma leitura e/ou escrita no banco de dados. Através de arquivos de configuração, as URLs são associadas a views e cada view é associada a uma URL.

O Django identifica a parte da URL que segue o nome do domínio, utilizada para selecionar a view que deve ser executada. Para selecionar a view, é utilizada uma expressão regular que tenta coincidir o texto da URL com um padrão configurado nos arquivos de configuração do sistema de URLs do Django.

Escrevendo uma view

As views são implementadas no arquivo views.py da aplicação. A primeira view a ser criada será chamada de index.

```
# reservas/views.py
from django.http import HttpResponse

def index(request):
    return HttpResponse('Bem vindo ao sistema de reservas ' \
                        'Umas e Ostras.')
```

A view index acima é a view mais simples possível de ser implementada no Django. Retorna apenas uma resposta estática sem usar bancos de dados ou templates. No entanto, é preciso ainda indicar ao Django qual URL vai acionar essa view. Isso é feito através da criação de um arquivo chamado urls.py no diretório da aplicação.

```
# reservas/urls.py
from django.conf.urls import url

from.import views

urlpatterns = [
    url(r'^$', views.index, name='index'),
]
```

Por enquanto basta saber que a URL inicial da aplicação está sendo associada à view index. A view index foi definida no módulo views; portanto, é referenciada como views.indexemplo:

```
# umas_e_ostras/urls.py
from django.conf.urls import include, url
from django.contrib import admin

urlpatterns = [
    url(r'^reservas/', include('reservas.urls')),
    url(r'^admin/', admin.site.urls),
]
```

No arquivo de URLs do projeto, onde já foi configurada a aplicação admin, deve ser configurada a aplicação reservas. Ao acessar a URL representada pelo padrão r'^reservas/', as URLs definidas no arquivo reservas/urls.py são incluídas e verificadas para identificar qual view deve ser executada. Por enquanto só existe a view index.

Argumentos da função URL

A função url() é usada para associar um padrão de expressão regular a uma view e aceita cinco parâmetros. Dois são obrigatórios: regex e view, e os demais são opcionais: kwargs, name e prefix. O parâmetro regex é um mnemônico para "regular expression", ou expressão regular, em português, usada para encontrar uma correspondência com o endereço da requisição.

- url (regex, view, kwargs, name, prefix).
- regex: é uma expressão regular, ou seja, um padrão para encontrar uma correspondência com o endereço da requisição.
- view: quando a correspondência é encontrada, a função da view especificada nesse parâmetro é chamada com os argumentos capturados da expressão regular.
- kwargs*: palavras-chave arbitrárias podem ser passadas como argumento através de um dicionário passado para a view.
- name*: é possível definir um nome para a URL a ser usado em outros lugares do Django, especialmente em templates.
- prefix*: prefixo para view, quando o parâmetro view é uma string.
- * Kwargs, name e prefix são parâmetros opcionais.

O parâmetro View será responsável por indicar a função da view que será acionada quando houver correspondência com a expressão regular, passando os argumentos que tiverem sido "capturados" da expressão regular. Já o parâmetro kwargs, se usado, contém um dicionário com palavras-chave que vão funcionar como outros argumentos para a view. O parâmetro name permite identificar a URL de forma única em todo o sistema (especialmente em templates).

Isso é particularmente útil nos casos em que há necessidade de usar URL reversa. O uso de URL reversa é necessário para as views padrão do Django, chamadas de Class Based Views e que serão apresentadas na próxima sessão. O último parâmetro, prefix, serve para indicar um prefixo que será colocado à frente do parâmetro view quando esse for uma string. Por exemplo, url(r'^reservas/', view='index', prefix='views') será redirecionado para views.index. Entretanto, esse comportamento está sendo depreciado e na versão 1.10 do Django não serão aceitas strings para o valor do parâmetro view.

Passando parâmetros para views

A passagem de parâmetros para as views ocorre através da configuração das URLs e da URL usada no navegador para realizar a requisição.

```
# reservas/urls.py
from django.conf.urls import url

from . import views

urlpatterns = [
    # ex: /reservas/
    url(r'^$', views.index, name='index'),
    # ex: /reservas/1/
    url(r'^(?P<cliente_id>[0-9]+)/$',
        views.detalhe, name='detalhe'),
    # ex: /reservas/1/lista/
    url(r'^(?P<cliente_id>[0-9]+)/lista/$',
        views.reservas, name='reservas'),
    # ex: /reservas/1/confirma/
    url(r'^(?P<cliente_id>[0-9]+)/confirma/$',
        views.confirma, name='confirma'),
]
```

A lista de URLs, urlpatterns, desse exemplo possui a definição de URLs para a aplicação reservas, uma para a view principal, outra para a view com detalhes do cliente, lista de reservas de um cliente e confirmação de reserva.

Funcionamento das URLs

- Quando uma URL é acessada no site, o Django carrega o módulo umas_e_ostras.urls.
- A lista urlpatterns é percorrida em ordem.
- As expressões regulares desse módulo não possuem o caractere de fim de texto '$', apenas uma barra ('/') no final.
- Quando o Django encontra uma função include, a primeira parte da URL que coincide com a expressão regular é extraída.

O sistema de URL funciona da seguinte forma: ao receber uma requisição, o Django carrega o módulo de configuração de URLs (no caso do exemplo é o módulo umas_e_ostras.urls). A lista de URLs, urlpatterns, é percorrida em ordem. Nesse módulo não é usado o caractere '$', pois ele só coincidiria com URLs que não possuem parâmetros. Por exemplo, se a configuração da URL reservas usasse a expressão regular r'^reservas/$' em vez de r'^reservas/', apenas a URL '/reservas/' haveria uma coincidência e isso não é o que se deseja neste momento.

Ao encontrar a função include, a primeira parte da URL que coincide com a expressão regular é extraída. Novamente, no exemplo, ao acessar qualquer coisa iniciada com reservas, será extraída a primeira parte, /reservas/, e o restante do reconhecimento será realizado pela configuração de URL incluída pela função include. No exemplo será reservas.urls.

- O restante da URL é enviado adiante para a configuração de URL incluída pela função include.
- Quando é acessada uma URL como '/reservas/42/', é encontrada a expressão que coincide '^reservas/'. Essa parte é extraída e o que sobra, '42/', é enviado para 'reservas.urls'.
- A função detalhe será chamada assim:
  - detalhe(request=<HttpRequest object>, cliente_id='42')

Um acesso à URL /reservas/42/ coincidirá com a URL de reservas. A palavra reservas será extraída e o restante da URL será enviado para o módulo reservas.urls. Assim, a configuração que aponta para a função detalhes que será chamada da seguinte forma:

```
detalhe(request=<HttpRequest object>, cliente_id='42')
```

As URLs da aplicação

As URLs da aplicação reservas, configuradas no arquivo reservas/urls.py, funcionam da seguinte maneira: ao ser realizado o acesso ao endereço /reservas/, a primeira parte será extraída e o processamento será realizado pelo módulo reservas.urls. O módulo receberá nesse momento a string vazia (a expressão regular que coincide com string vazia é r'^$'). O primeiro caractere da expressão é '^', que simboliza o início do texto. O segundo caractere é '$', que representa o fim do texto. O início do texto seguido do fim do texto coincide com uma string vazia e nada mais. Assim, a função correspondente views.index será executada.

- A primeira URL, index, com a expressão r'^$'.
- ^ representa o início de texto.
- $ representa o fim do texto.
- Portanto, é esperado que não haja nada entre o início e o fim do texto.
- Quando /reservas/ for acessada, será executada a função views.index.
- A segunda URL, detalhe, com a expressão r'^(?P<cliente_id>[0-9]+)/$'
- ^ representa o início de texto.
- Seguido de um ou mais números '[0-9]+'
- Fechando com a barra '/' e fim do texto '$'
- A grande diferença é que uma variável '?P<cliente_id>' é definida e receberá o valor do número.
- Uma URL /reservas/1/ vai executar a função views.detalhe com o parâmetro cliente_id = 1.

Quando for acessado o endereço /reservas/1/, o sistema de URLs do Django vai primeiro extrair a primeira parte, /reservas/, e o processamento será passado para o módulo reservas.urls com o texto '1/'. A expressão regular que coincide com um número é r'[0-9]'.

Para coincidir com mais de um número, a expressão regular precisa usar o caractere '+'. '+' significa dizer que a expressão anterior deve ocorrer no mínimo uma vez e pode se repetir indefinidamente. Então, a expressão r'[0-9]+' coincide com um ou mais números (que contenham pelo menos um algarismo). A expressão completa r'^(?P<cliente_id>[0-9]+)/$' coincide com início de um texto, seguido de um ou mais algarismos, seguido da barra e fim de texto. A novidade é o trecho '?P<cliente_id>', que é a maneira usada em expressões regulares para declarar uma variável. No exemplo, a variável definida é cliente_id.

Quando for acessada a URL /reservas/1/lista/, o sistema de URLs do Django vai novamente extrair a primeira parte da URL (/reservas/). O processamento será encaminhado para o módulo reservas.urls com o texto remanescente '1/lista/'. A expressão numérica é semelhante ao que foi explicado no parágrafo anterior, sendo que agora, após o número, temos ainda a palavra lista.

- A terceira URL, lista, é semelhante à anterior, mas após o número é esperada a palavra lista. Uma URL /reservas/1/lista/ executará a função views.lista com o parâmetro cliente_id = 1.
- A última URL, confirma, espera a palavra confirma após o número. Uma URL /reservas/1/confirma/ executará a função views.confirma com o parâmetro cliente_id = 1.

A última URL configurada é semelhante às anteriores. Apenas a palavra final é diferente: 'confirma' (em vez de 'lista').

Tratando parâmetros da URL

Os nomes das variáveis definidos nas expressões regulares são utilizados na execução das funções correspondentes.

```
# reservas/views.py
from django.http import HttpResponse

def index(request):
    return HttpResponse('Bem vindo ao sistema de reservas ' \
                        'Umas e Ostras.')

def detalhe(request, cliente_id):
    return HttpResponse('Foi feita uma solicitação pelo ' \
                        'cliente: {}.'.format(cliente_id))

def reservas(request, cliente_id):
    resposta = 'Foi feita uma solicitação para as reservas ' \
               ' do cliente {}.'
    return HttpResponse(resposta.format(cliente_id))

def confirma(request, cliente_id):
    resposta = 'Foi confirmada a reserva {}.'.format(cliente_id)
    return HttpResponse(resposta)
```

Para receber os parâmetros nas views, basta definir o parâmetro com o mesmo nome utilizado na expressão regular configurada no módulo reservas.urls. No exemplo são definidos os parâmetros cliente_id para as views detalhe, reservas e confirma.

Funcionamento das views

As views recebem os parâmetros como um argumento de função normal da linguagem Python. O primeiro argumento é um objeto request que contém dados da requisição. Em seguida, podem vir um ou mais argumentos dependendo da configuração feita. No caso do exemplo, temos apenas um argumento: cliente_id.

- Nas views, o parâmetro é recebido como um argumento de função.
- A resposta é enviada da maneira mais simples possível, usando a classe HttpResponse.
- Para testar, basta acessar as seguintes URLs variando o número:
  - http://localhost:8000/reservas/1/
  - http://localhost:8000/reservas/1/lista/
  - http://localhost:8000/reservas/1/confirma/

A resposta retornada pelas views do exemplo são as mais simples possíveis, utilizando a classe HttpResponse. Nos exemplos não foi realizado acesso a bancos de dados e nem foram utilizados templates. Para testar, basta que o servidor seja iniciado, com o comando python manage.py runserver, e sejam acessadas as URLs acima indicadas.

As responsabilidades das views

- Views devem retornar um objeto HttpResponse ou lançar uma exceção (por exemplo, Http404).
- Normalmente, as views lidam com bancos de dados, mas também podem gerar arquivos PDF, XML, ZIP, TXT etc.
- Pode ser usado qualquer sistema de templates escrito em Python e qualquer API de banco de dados.
- Por comodidade, é mais comum usar as APIs do Django.

As views devem seguir um comportamento padrão. Devem sempre retornar um objeto HttpResponse ou lançar uma exceção. Nos projetos com Django, que normalmente usam bancos de dados, as views é que têm a responsabilidade de fazer essa interação, criando, recuperando, alterando ou excluindo registros. Mas isso não é uma regra. As views podem gerar arquivos PDF, XML, ZIP, TXT ou CSV, acessando ou não bancos de dados.

O Django provê um sistema de templates, mas sistemas de templates fornecidos por terceiros podem também ser utilizados. Podemos dizer o mesmo sobre APIs de bancos, embora a própria API de banco de dados do Django seja a mais adequada para utilizar a interface admin do Django.

Por isso, o mais cômodo é usar as APIs do Django e, se houver alguma necessidade muito específica, utilizar outra API para resolver casos particulares.

Listando os clientes mais recentes

O acesso a bancos de dados com a API do Django é muito simples. A primeira coisa que precisa ser feita é a importação da Classe definida no model.

```
# reservas/views.py
from django.http import HttpResponse

from .models import Cliente

def index(request):
    ultimos_clientes = Cliente.objects.order_by('-registrado_em')[:5]
    retorno = ', '.join([cli.nome for cli in ultimos_clientes])
    return HttpResponse(retorno)
```

```
def detalhe(request, cliente_id):
    return HttpResponse('Foi feita uma solicitação pelo ' \
                        'cliente: {}.'.format(cliente_id))
def reservas(request, cliente_id):
    response = 'Foi feita uma solicitação para as reservas ' \
               ' do cliente {}.'
    return HttpResponse(response.format(cliente_id))

def confirma(request, cliente_id):
    return HttpResponse('Foi confirmada a ' \
                        'reserva {}.'.format(cliente_id))
```

Ao utilizar APIs de banco de dados como a API do Django, raramente é necessário escrever comandos SQL. No exemplo anterior são recuperados os cinco clientes com registro mais recente.

```
ultimos_clientes = Cliente.objects.order_by('-registrado_em')[:5]
```

A linha de código acima pode ser lida como: ordenar os objetos cliente em ordem decrescente usando o campo registrado_em e recupere os cinco primeiros. O detalhe mais sutil é o uso do operador "-", que indica que a ordem é decrescente.

```
    retorno = ', '.join([cli.nome for cli in ultimos_clientes])
    return HttpResponse(retorno)
```

Com o resultado armazenado em uma lista "ultimos_clientes", é montada uma string, utilizando compreensão de listas e o método join de strings. Em seguida, é retornado um objeto HttpResponse com o resultado.

Atividades Práticas 

Separando lógica e apresentação

- Para separar o código Python do design, deve ser usado o sistema de templates do Django.
- Primeiro precisam ser criados os diretórios dos templates. O Django vai procurar um diretório templates em cada aplicação presente em INSTALLED_APPS.
- Por questões de organização e para evitar conflitos de nomes, recomenda-se criar um subdiretório com o nome da aplicação a seguir do diretório de templates.

Até o momento, aquilo que será retornado pela view tem sido montado manualmente, utilizando objetos HttpResponse. Mas isso não é adequado, já que em sistemas web modernos o design é mais elaborado e gerar o código HTML nas views acaba misturando camadas diferentes de padrões de projeto.

O mais adequado é utilizar o sistema de templates do Django. Para isso, será necessário criar os diretórios dos templates. Por padrão, o Django procura um diretório templates em cada aplicação presente na configuração da variável INSTALLED_APPS.

Esse diretório não é criado automaticamente, sendo responsabilidade do desenvolvedor cuidar disso. Não é necessário, mas por questões de organização é importante criar um subdiretório com o nome da aplicação a seguir do diretório de templates. No caso do projeto de exemplo, o nome do subdiretório será reservas.

Template index da lista de clientes

Para implementar o template da página index do sistema de reservas e apresentar os cinco clientes com registro mais recente, deve ser criado o arquivo indexemplo:html. Esse arquivo deve ser salvo no diretório de templates recém-criado:

```
reservas/templates/reservas/index.html
```

O conteúdo do template está logo a seguir:

```
{% if ultimos_clientes %}
    <ul>
    {% for cliente in ultimos_clientes %}
        <li><a href="/reservas/{{ cliente.id }}/">{{ cliente.nome }}</a></li>
    {% endfor %}
    </ul>
{% else %}
    <p>Nenhum cliente registrado.</p>
{% endif %}
```

O template utiliza algumas estruturas simplificadas para permitir algumas iterações e testes simples.

Sobre o template

- A linguagem de template é muito simples, e não deve ser usada para implementar nada muito avançado.
- No exemplo, é testada a variável ultimos_clientes. Se ela estiver vazia, será avaliada como falsa e apenas o conteúdo da cláusula else aparecerá na página.
- Caso a variável não esteja vazia, será executado um laço for percorrendo a lista e gerando o conteúdo da página.

A linguagem de template do Django serve para realizar algumas tarefas simples e não deve ser utilizada para implementar nada muito avançado, já que a lógica principal deve ficar nas views. O exemplo mostra dois comandos comuns: um teste condicional com if que verifica se a variável ultimos_clientes possui algum item.

Mais informações sobre a linguagem de template do Django podem ser vistas na documentação oficial, em https://docs.djangoproject.com/en/1.9/ref/templates/language/.

Se a variável ultimos_clientes possuir itens, a lista será percorrida com um laço for e será criada uma lista HTML do tipo “ul” com os itens da lista “li”. Se a lista estiver vazia, será impressa a mensagem indicando que não há clientes na lista.

A linguagem de template é realmente simples e possui poucos comandos.

Usando o template na view index

Para usar o template, serão necessárias duas importações. Na primeira temos uma classe que cria um contexto a partir da requisição, chamada RequestContext. A segunda importação é de uma função para carregar o template, chamada loader.

```
# reservas/views.py
from django.http import HttpResponse
from django.template import RequestContext, loader

from .models import Cliente

def index(request):
    ultimos_clientes = Cliente.objects.order_by(
        '-registrado_em')[:5]
    template = loader.get_template('reservas/index.html')
    context = RequestContext(request, {
        'ultimos_clientes': ultimos_clientes,
    })
    return HttpResponse(template.render(context))

def detalhe(request, cliente_id):
    return HttpResponse('Foi feita uma solicitação pelo ' \
                        'cliente: {}.'.format(cliente_id))

def reservas(request, cliente_id):
    response = 'Foi feita uma solicitação para as reservas ' \
               ' do cliente {}.'
    return HttpResponse(response.format(cliente_id))

def confirma(request, cliente_id):
    return HttpResponse('Foi confirmada a ' \
                        'reserva {}.'.format(cliente_id))
```

A view acessa o banco de dados através da API do Django e utiliza o template para apresentar os dados. A variável "context" é um dicionário que mapeia nomes de variável do template para objetos Python. Para carregar a página, basta acessar o endereço da aplicação, "/reservas/". Se o servidor estiver rodando na porta 8000, que é o padrão, basta acessar: "http://localhost:8000/reservas/".

Sobre a view

- O código carrega o template.
- É criado um contexto, um dicionário mapeando nomes de variáveis para o template.
- Então um objeto HttpResponse é criado, passando o template renderizado com o contexto.

A view index recupera os cinco registros de cliente mais recentes. Em seguida, carrega o template "reservas/index.html". É criado um contexto, utilizando a classe RequestContext, com os dados da requisição ("request") e a lista de clientes ("ultimos_clientes").

A view retorna no final um objeto HttpResponse com o template já renderizado (método template.render) utilizando os dados do contexto.

Utilizando o atalho render

Como as tarefas de ler um template, preencher um contexto e retornar um objeto HttpResponse são muito comuns, o Django fornece um atalho, chamado render.

O atalho render recebe como primeiro parâmetro um objeto request. O objeto request representa a requisição recebida pelo servidor de páginas e encaminhada para a aplicação, neste caso a aplicação reservas. O segundo parâmetro aceito pela função render é o template que deve ser usado para compor a resposta.

O terceiro parâmetro é o context. O context é o contexto, ou seja, os dados que devem ser usados no template para montar a resposta. Esses dados são passados como um dicionário. As chaves são os nomes de variáveis que estarão disponíveis no template e os valores associados a essas chaves são os dados que serão apresentados. O retorno é um objeto do tipo HttpResponse, que representa uma resposta que será enviada ao navegador que fez a requisição através do servidor de páginas.

```
# reservas/views.py
from django.http import HttpResponse
from django.shortcuts import render

from .models import Cliente

def index(request):
    ultimos_clientes = Cliente.objects.order_by(
        '-registrado_em')[:5]
    context = {'ultimos_clientes': ultimos_clientes}
    return render(request, 'reservas/index.html', context)

def detalhe(request, cliente_id):
    return HttpResponse('Foi feita uma solicitação pelo ' \
                        'cliente: {}.'.format(cliente_id))

def reservas(request, cliente_id):
    response = 'Foi feita uma solicitação para as reservas ' \
               ' do cliente {}.'
    return HttpResponse(response.format(cliente_id))

def confirma(request, cliente_id):
    return HttpResponse('Foi confirmada a ' \
                        'reserva {}.'.format(cliente_id))
```

O código já fica mais simples: basta acessar o banco de dados e recuperar os registros de clientes, criar um dicionário com a lista de clientes e utilizar o atalho render para encaminhar as variáveis do contexto ao template. Por fim, o template é processado e um objeto HttpResponse é retornado.

```
    ultimos_clientes = Cliente.objects.order_by(
        '-registrado_em')[:5]
    context = {'ultimos_clientes': ultimos_clientes}
    return render(request, 'reservas/index.html', context)
```

Objeto não encontrado

- Quando uma página não é encontrada em um site, é emitido um erro 404 para o navegador.
- Quando uma view é acessada, e ela existe, isso não acontece.
- Quando o registro que devemos exibir não existe, podemos usar uma exceção do Django para emitir um erro 404.
- O código do próximo exemplo precisa conter o código a seguir no template a ser criado em reservas/templates/reservas/detalhe.html:
  - {{ cliente }}

Conforme já mencionado no início desta sessão, a camada de visão do Python está fortemente baseada no protocolo HTTP. Esse protocolo define alguns códigos numéricos para indicar o estado de um serviço. Um dos códigos mais conhecidos é o 404, que indica que um recurso não foi encontrado, seja esse recurso uma pasta, arquivo ou página.

Quando uma view já existente é acessada, obviamente nenhum erro 404 será emitido. Mas e se o propósito da view for recuperar um objeto do banco de dados e esse objeto não for encontrado? Nesse caso, uma exceção do Django pode ser usada para emitir um erro 404 quando o registro que foi pesquisado não for encontrado.

Para o próximo exemplo, é preciso criar um novo template no diretório de templates da aplicação reservas reservas/templates/reservas/detalhe.htm, contendo o código a seguir:

```
{{ cliente }}
```

Esse conteúdo é tudo o que é necessário no momento para mostrar o funcionamento do erro 404 no Django.

Emitindo um erro 404

Para emitir um erro 404, a classe Http404 precisa ser importada. Essa classe será usada para disparar a exceção quando necessário.

```
# reservas/views.py
from django.http import HttpResponse
from django.http import Http404
from django.shortcuts import render

from .models import Cliente

def index(request):
    ultimos_clientes = Cliente.objects.order_by(
        '-registrado_em')[:5]
    context = {'ultimos_clientes': ultimos_clientes}
    return render(request, 'reservas/index.html', context)

def detalhe(request, cliente_id):
    try:
        cliente = Cliente.objects.get(pk=cliente_id)
    except Cliente.DoesNotExist:
        raise Http404("Cliente não existe")
```

```
    return render(request, 'reservas/detalhe.html', {'cliente': cliente})

def reservas(request, cliente_id):
    response = 'Foi feita uma solicitação para as reservas ' \
               ' do cliente {}.'
    return HttpResponse(response.format(cliente_id))

def confirma(request, cliente_id):
    return HttpResponse('Foi confirmada a ' \
                        'reserva {}.'.format(cliente_id))
```

Ao utilizar o método get da API de banco de dados, o registro com a chave primária "cliente_id" será recuperado. Entretanto, se a chave não existir, será disparada uma exceção "Cliente.DoesNotExist" e um erro de sistema aparecerá no navegador do usuário. Para evitar que isso aconteça, é preciso tratar essa exceção.

```
def detalhe(request, cliente_id):
    try:
        cliente = Cliente.objects.get(pk=cliente_id)
    except Cliente.DoesNotExist:
        raise Http404("Cliente não existe")
    return render(request, 'reservas/detalhe.html', {'cliente': cliente})
```

Em vez de exibir somente um código de erro, faz-se o tratamento da exceção para indicar algo que os usuários possam compreender mais facilmente. Na prática, o que terá acontecido é que o objeto com cliente_id solicitado não terá sido encontrado, disparando-se a exceção Http404. Ao passar uma string como parâmetro da exceção, teremos uma mensagem de texto indicando o que aconteceu: Http404("Cliente não existe").

get_object_or_404()

- O Django possui um atalho que executa o método get e já emite a exceção Http404 se o objeto não é encontrado.
- O atalho get_object_or_404() recebe como argumentos: um modelo e um número arbitrário de argumentos que serão passados ao método get().
- Também existe um atalho chamado get_list_or_404(), que usa o método filter para retornar uma lista, emitindo uma exceção Http404 se a lista estiver vazia.

A tarefa de disparar uma exceção Http404 quando um registro não é encontrado é também muito comum. Por isso, mais um atalho foi criado para facilitar o trabalho do desenvolvedor. Esse atalho executa o método get e quando o registro não é encontrado, já disparada a exceção Http404. Os argumentos passados ao atalho get_object_or_404() são o modelo definido no módulo models da aplicação e argumentos arbitrários que serão utilizados na chamada do método get().

Existe um atalho similar para listas, o get_list_or_404(), que usa o método filter() e dispara a exceção Http404 se a lista estiver vazia.

Usando o atalho get_object_or_404

O atalho get_object_or_404() deve ser importado do módulo de atalhos do Django, o django.shortcuts.

```
# reservas/views.py
from django.http import HttpResponse
from django.shortcuts import render, get_object_or_404

from .models import Cliente

def index(request):
    ultimos_clientes = Cliente.objects.order_by(
        '-registrado_em')[:5]
    context = {'ultimos_clientes': ultimos_clientes}
    return render(request, 'reservas/index.html', context)

def detalhe(request, cliente_id):
    cliente = get_object_or_404(Cliente, pk=cliente_id)
    return render(request, 'reservas/detalhe.html',
                {'cliente': cliente})

def reservas(request, cliente_id):
    response = 'Foi feita uma solicitação para as reservas ' \
               ' do cliente {}.'
    return HttpResponse(response.format(cliente_id))

def confirma(request, cliente_id):
    return HttpResponse('Foi confirmada a ' \
                        'reserva {}.'.format(cliente_id))
```

A utilização do atalho simplifica muito o código da view detalhe().

```
def detalhe(request, cliente_id):
    cliente = get_object_or_404(Cliente, pk=cliente_id)
    return render(request, 'reservas/detalhe.html',
                {'cliente': cliente})
```

O que era feito em quatro linhas agora é feito em uma única linha. Basta passar a classe do model, Cliente, e o parâmetro usado para recuperar o registro, "pk=cliente_id".

Um desenvolvedor deve sempre tratar as exceções para não permitir que apenas um código de erro seja exibido para o usuário. Aliás, essas mensagens só aparecem se o modo Debug estiver ligado, e uma aplicação nunca deve ser disponibilizada em modo Debug. A maneira mais adequada para customizar o tratamento de erro é criar páginas de erro 404 customizadas (https://docs.djangoproject.com/en/1.9/topics/http/views/#django.http.Http404).

O sistema de templates

- A sintaxe com pontuação (usando o ponto) é usada para descobrir atributos de variáveis.
- {{ cliente.nome }} por exemplo, obtém o valor do nome do objeto cliente passado para o template no contexto (context).
- {{ cliente.reserva_set.all }} faz uma chamada de método, interpretado como um comando Python, semelhante a cliente.reserva_set.all(). Esse método retorna um iterável que pode ser usado em um laço for.

O sistema de templates do Django usa uma sintaxe parecida com a linguagem Python. No exemplo anterior, o template era extremamente simples: "{{ cliente }}". No dia a dia o mais comum será trabalhar com objetos Python, acessando os valores dos seus atributos. Para isso é utilizada a sintaxe com pontuação.

Para exibir o nome de um cliente, deve ser usado o objeto cliente passado através do dicionário "context" na execução da renderização do template. A sintaxe utilizada é {{ cliente.nome }}.

Além de atributos, também podem ser realizadas chamadas a métodos. A sintaxe usada é: {{ cliente.reserva_set.all }}.

Template - detalhes

O novo conteúdo do template de detalhes da reserva pode ser visto a seguir e deve ser salvo no arquivo reservas/templates/reservas/detalhe.html.

O conteúdo HTML é bem simples, com o sistema de templates do Django cuidando de fazer substituições do conteúdo sempre que forem encontradas marcas especiais. Existem duas marcas que servem a propósitos específicos. Onde há chaves duplas "{{ }}" haverá a inclusão do conteúdo da variável ou atributo indicados. Já as chaves e percentil "{% %}" indicam que um comando deverá ser interpretado. O mais comum é gerar texto na saída, implementar um controle de fluxo (por exemplo, o comando for) ou ainda carregar informação externa para ser usada no template.

```
<h1>{{ cliente.nome }}</h1>
<ul>
{% for reserva in cliente.reserva_set.all %}
    <li>{{ reserva.data_evento }} - {{ reserva.pessoas }}</li>
{% endfor %}
</ul>
```

O nome do cliente será impresso entre as marcas HTML <h1></h1>.

```
<h1>{{ cliente.nome }}</h1>
```

O laço for vai percorrer todos os registros das reservas do cliente. Esse comando por si só não gera nenhum conteúdo na página após a renderização.

```
{% for reserva in cliente.reserva_set.all %}
```

Os atributos de cada reserva do cliente serão gerados como itens de lista HTML "
".

```
    <li>{{ reserva.data_evento }} - {{ reserva.pessoas }}</li>
```

Então o comando for é encerrado.

```
{% endfor %}
```

URLs flexíveis

- Para tornar as URL mais flexíveis, é utilizada a tag {% url %} no template.
- Para isso o nome da URL deve ser definido na configuração das URLs no arquivo urls.py.
- O nome da URL de detalhes já foi definido e pode ser usado no template index.
- Isso é muito prático quando a aplicação tem muitos templates, já que uma mudança no urls.py é refletida em todos os templates.

Os templates, como já foi visto, são páginas HTML com marcas especiais no padrão do sistema de templates do Django. Portanto, aceitam qualquer coisa que é aceita em HTML, incluindo links. Mas não faz muito sentido ter um sistema de templates em um framework e ter de lidar com links manualmente. O framework Django possui a marca {% url %}, que ajuda e muito no gerenciamento desses links.

Um requisito necessário para utilizar a marca {% url %} é a definição de um nome para a URL na configuração das URLs que fica no arquivo urls.py da aplicação.

Quando foi definida a configuração da URL da view detalhes no arquivo urls.py, foi também definido um nome, nome esse que pode ser usado na marcação {% url %} nos templates. Qualquer mudança no arquivo urls.py será refletida em todos os templates utilizados.

Template index com tag {% url %}

Apresentamos a seguir o conteúdo do arquivo reservas/templates/reservas/index.html. Nela a URL para o link de referência aos detalhes do cliente será gerada de acordo com o nome definido no arquivo urls.py da aplicação. O resultado final da utilização da marca {% url 'detalhe' cliente.id %} será algo parecido com "/reservas/1/" para um cliente com o campo "id" igual a 1.

Conteúdo do arquivo reservas/templates/reservas/index.html.

```
{% if ultimos_clientes %}
    <ul>
    {% for cliente in ultimos_clientes %}
        <li>
          <a href="{% url 'detalhe' cliente.id %}">
            {{ cliente.nome }}
          </a>
        </li>
    {% endfor %}
    </ul>
{% else %}
    <p>Nenhum cliente registrado.</p>
{% endif %}
```

Definindo escopo de URLs

- Quando um projeto cresce pode ter cinco, dez, vinte aplicações.
- Para evitar conflitos de nomes, o mais prático é definir o escopo (ou namespace) das URLs.
- Isso é feito no arquivo urls.py do projeto.
- Quando isso for feito, precisamos fazer uma indicação no template

O escopo de uma URL é uma maneira de evitar que as aplicações de um projeto tenham conflitos de nomes. Se cada aplicação tiver uma view chamada "index", como o Django saberá qual URL deve gerar? Para evitar isso, cada aplicação pode definir um escopo, ou pelo nome em inglês, namespace.

No arquivo de configuração de URLs do projeto, onde é realizada a inclusão dos arquivos de configuração das aplicações com a função "include", é definido o namespace. Após essa alteração, é necessário alterar também o template, pois a marca {% url %} não funcionará sem o uso do namespace definido.

Namespace reservas

A primeira alteração necessária é feita no arquivo urls do projeto, "umas_e_ostras/urls.py". Basta adicionar mais um parâmetro à configuração da URL. Logo após o parâmetro com o nome do módulo de configuração de URLs da aplicação, na função include, é passado um novo parâmetro, namespace, com o nome desejado. Em geral é usado o mesmo nome da aplicação para o namespace.

umas_e_ostras/urls.py

```
# umas_e_ostras/urls.py
from django.conf.urls import include, url
from django.contrib import admin

urlpatterns = [
    url(r'^reservas/',
        include('reservas.urls', namespace='reservas')),
    url(r'^admin/',
        admin.site.urls),
]
```

A outra alteração necessária é feita no template onde é usada a marca {% url %}, que no exemplo corresponde ao arquivo reservas/templates/reservas/index.html. Basta acrescentar o mesmo nome usado na configuração, seguido de dois pontos e do nome da visão, resultando em: "{% url 'reservas:detalhe' cliente.id %}".

reservas/templates/reservas/index.html

```
{% if ultimos_clientes %}
    <ul>
    {% for cliente in ultimos_clientes %}
        <li>
          <a href="{% url 'reservas:detalhe' cliente.id %}">
            {{ cliente.nome }}
          </a>
        </li>
    {% endfor %}
    </ul>
{% else %}
    <p>Nenhum cliente registrado.</p>
{% endif %}
```

Formulários

- A entrada de informações ocorre através de formulários.
- A aplicação de reservas vai permitir que as reservas do cliente sejam confirmadas na página de detalhes do cliente.
- O formulário vai exibir as reservas do cliente ainda não confirmadas, com uma caixa de confirmação ao lado de cada reserva.
- O template detalhes precisa de um formulário cuja ação é a view confirma.

Os formulários web são páginas HTML que possuem espaços ou campos específicos para receber informações do usuário e/ou exibir dados formatados. É muito comum as aplicações desenvolvidas em Django trabalharem com formulários, pois eles permitem a interação dos usuários com o sistema, inclusive para realizar pesquisas.

A aplicação de reservas ainda não possui uma forma de permitir a confirmação de reservas. Uma maneira de implementar isso é através da criação de um formulário na página de detalhes do cliente.

Um formulário exibindo a lista de reservas não confirmadas do cliente com uma caixa de confirmação ao lado de cada reserva possibilita a confirmação de várias reservas de uma só vez. O template de detalhes do cliente deve ter um formulário com essa lista e cuja ação será a view confirma.

Aprimorando o sistema do Exemplo

Para implementar o formulário e a view confirma algumas mudanças precisam ser feitas no sistema sendo desenvolvido. Ainda não há como filtrar as reservas que não foram confirmadas. A tabela de reservas não possui nenhum campo que permita saber se a reserva foi confirmada. A URL responsável pela view confirma precisa passar o id do cliente e não o id da reserva. Assim, as seguintes providências precisam ser tomadas:

- O model Cliente precisa de um método para filtrar apenas as reservas ainda não confirmadas.
- O model Reserva precisa de um campo para confirmação.
- A URL confirma vai receber o id do cliente e não o id da reserva, então o mais indicado é trocar o nome do parâmetro no arquivo reservas/urls.py.
- A view reservas precisa ser implementada.
- A view confirma precisa tratar as informações do formulário.

Template detalhes

O template detalhe.html terá um formulário e utilizará a marca {% url %} para formar a URL da ação do formulário. O Django possui uma aplicação, que já está habilitada, para proteger o sistema de uma técnica de ataque a sistemas web chamada Cross Site Request Forgery (csrf), por isso é usada a marca {% csrf_token %}.

```
<h1>{{ cliente.nome }}</h1>
{% if error_message %}<p><strong>{{ error_message }}</strong></p>{% endif %}
<form action="{% url 'reservas:confirma' cliente.id %}" method="post">
    {% csrf_token %}
    {% for reserva in cliente.reservas_nao_confirmadas %}
    <input type="checkbox" name="confirmacao" id="confirmacao{{ forloop.counter }}"
value="{{ reserva.id }}" {% if reserva.confirmada %}checked{% endif %}/>
    <label for="confirmacao{{ forloop.counter }}"> Confirmar? - {{ reserva.data_
evento }} - {{ reserva.pessoas }}</label><br />
    {% endfor %}
    <input type="submit" value="Atualizar" />
</form>
```

O formulário possui ainda um laço for que percorre o resultado do método que filtra as reservas não confirmadas do model cliente, reservas_nao_confirmadas, que ainda será implementado. No laço for são geradas as caixas de confirmação e o texto da reserva.

Mudança nos models

Algumas mudanças são necessárias para que o formulário funcione da maneira desejada. A linguagem de templates não foi criada para permitir o acesso a bancos de dados diretamente, justamente para separar a camada de apresentação e a camada de acesso a dados e lógica do sistema. Por isso, o filtro de reservas não confirmadas precisa ser feito em outro lugar, nesse caso no modelo de clientes. Além disso, a tabela de reservas precisa de um campo para indicar se a reserva foi confirmada ou não.

```
# reservas/models.py
import datetime

from django.db import models
from django.utils import timezone

class Cliente(models.Model):
    nome = models.CharField(max_length=200)
    endereco = models.CharField(max_length=200)
    telefone = models.CharField(max_length=20)
    email = models.CharField(max_length=200)
    registrado_em = models.DateTimeField('data do registro')

    def reservas_nao_confirmadas(self):
        return self.reserva_set.filter(confirmada=False)

    def reservas_confirmadas(self):
        return self.reserva_set.filter(confirmada=True)

    def __str__(self):
        return self.nome

    def registro_eh_antigo(self):
        um_ano = timezone.now() - datetime.timedelta(days=365)
        return self.registrado_em < um_ano
    registro_eh_antigo.admin_order_field = 'registrado_em'
    registro_eh_antigo.boolean = True
    registro_eh_antigo.short_description = 'Cliente antigo?'



class Reserva(models.Model):
    data_reserva = models.DateTimeField('data da reserva')
    data_evento = models.DateTimeField('data do evento')
    pessoas = models.IntegerField(default=0)
    confirmada = models.BooleanField(default=False)
    cliente = models.ForeignKey(Cliente)
```

```
    def __str__(self):
        return '{} - {}'.format(
            self.cliente, str(self.data_evento))
```

O método reservas_nao_confirmadas utiliza o campo confirmada do modelo Reserva para filtrar as reservas dos clientes que não foram confirmadas. A API de dados do Django implementa o método filter aplicado ao conjunto de reservas, reserva_set, que representa o relacionamento de cliente e reservas. Esse método foi usado no template detalhe.html do exemplo anterior para listar as reservas não confirmadas. Da mesma forma, será necessário um método que filtra as reservas confirmadas do cliente, para exibição na lista de reservas do cliente.

```
    def reservas_nao_confirmadas(self):
        return self.reserva_set.filter(confirmada=False)

    def reservas_confirmadas(self):
        return self.reserva_set.filter(confirmada=True)
```

O campo confirmada, do tipo booleano, foi adicionado ao modelo Reserva para indicar se a reserva foi confirmada.

```
    confirmada = models.BooleanField(default=False)
```

Quando um modelo é alterado, é necessário atualizar o banco de dados. Isso é feito usando o utilitário manage.py com as opções makemigrations e migrate.

```
Depois dessa mudança é preciso rodar:
$ python manage.py makemigrations
$ python manage.py migrate
```

Mudança na URL confirma

A URL confirma recebia o parâmetro reserva_id. Nessa evolução do sistema, o parâmetro deve passar a ser cliente_id, para indicar de qual cliente as reservas devem ser confirmadas.

```
# reservas/urls.py
from django.conf.urls import url

from . import views

urlpatterns = [
    # ex: /reservas/
    url(r'^$', views.index, name='index'),
    # ex: /reservas/1/
    url(r'^(?P<cliente_id>[0-9]+)/$',
        views.detalhe, name='detalhe'),
    # ex: /reservas/1/lista/
    url(r'^(?P<cliente_id>[0-9]+)/lista/$',
        views.reservas, name='reservas'),
    # ex: /reservas/1/confirma/
    url(r'^(?P<cliente_id>[0-9]+)/confirma/$',
        views.confirma, name='confirma'),
]
```

Assim sendo, a última URL cujo nome é confirma tem o nome do parâmetro alterado de reserva_id para cliente_id.

```
    url(r'^(?P<cliente_id>[0-9]+)/confirma/$',
        views.confirma, name='confirma'),
```

Mudança nas views

A view confirma precisa agora receber os parâmetros enviados pelo formulário, cliente_id e as reservas que devem ser confirmadas.

```
# reservas/views.py
from django.http import HttpResponse
from django.http import HttpResponseRedirect
from django.shortcuts import render
from django.shortcuts import get_object_or_404
from django.core.urlresolvers import reverse

from .models import Cliente

def index(request):
    ultimos_clientes = Cliente.objects.order_by(
        '-registrado_em')[:5]
    context = {'ultimos_clientes': ultimos_clientes}
    return render(request, 'reservas/index.html', context)

def detalhe(request, cliente_id):
    cliente = get_object_or_404(Cliente, pk=cliente_id)
    return render(request, 'reservas/detalhe.html',
                  {'cliente': cliente})

def reservas(request, cliente_id):
    cliente = get_object_or_404(Cliente, pk=cliente_id)
    return render(request, 'reservas/lista.html',
                  {'cliente': cliente})

def confirma(request, cliente_id):
    cliente = get_object_or_404(Cliente, pk=cliente_id)
    confirmados = request.POST.getlist('confirmacao')
    for reserva_id in confirmados:
        try:
                reserva = cliente.reserva_set.get(pk=reserva_id)
        except (KeyError, Reserva.DoesNotExist):
            # Reapresenta o formulário do cliente.
            return render(request, 'reservas/detail.html', {
                'cliente': cliente,
                'error_message': "Código da reserva não encontrado.",
            })
        else:
            reserva.confirmada = True
            reserva.save()
```

```
    # Sempre retornar uma HttpResponseRedirect depois de tratar com
    # sucesso os dados do POST. Isso evita que dados sejam postados
    # novamente caso o usuário pressione o botão voltar.
    return HttpResponseRedirect(reverse('reservas:reservas',
                                        args=(cliente.id,)))
```

É altamente recomendado realizar um redirecionamento após realizar operações que alteram o estado do sistema. No Django sempre deve ser retornado um objeto HttpResponseRedirect depois de tratar com sucesso os dados do POST. Isso evita que dados sejam postados novamente caso o usuário pressione o botão voltar ou atualizar página.

Caso o id do cliente esteja incorreto ou não exista no banco de dados, será emitido um erro 404.

```
    cliente = get_object_or_404(Cliente, pk=cliente_id)
```

O formulário encaminhará todos os componentes checkbox marcados em uma lista. Para recuperar essa lista, o método getlist do objeto POST é chamado passando o nome do componente como parâmetro.

```
    confirmados = request.POST.getlist('confirmacao')
```

Como os componentes checkbox gerados no template detalhe.html foram definidos com o valor utilizando o id das reservas, é utilizado o valor para recuperar a reserva do banco de dados.

```
                reserva = cliente.reserva_set.get(pk=reserva_id)
```

Se a reserva não for encontrada, o formulário é reapresentado com uma mensagem de erro.

```
            return render(request, 'reservas/detail.html', {
                'cliente': cliente,
                'error_message': "Código da reserva não encontrado.",
            })
```

Se a reserva for encontrada, o campo confirmada é alterado para True e a reserva é salva.

```
            reserva.confirmada = True
            reserva.save()
```

Para evitar problemas com o navegador, caso o usuário atualize a página ou clique em voltar, sempre deve ser realizado um redirecionamento. Isso evita que alguma informação seja gravada em duplicidade, especialmente importante quando um novo registro é criado.
A maneira de fazer isso no Django é retornar um objeto HttpResponseRedirect.

```
    return HttpResponseRedirect(reverse('reservas:reservas',
                                        args=(cliente.id,)))
```

Template lista

Após a confirmação das reservas, a view confirma redireciona a resposta para a view reservas, que por sua vez exibe a lista de reservas do cliente. O template utilizado é lista.html, que exibe a lista de reservas confirmadas utilizando o método criado no models.py.

```
<h1>{{ cliente.nome }}</h1>
<ul>
{% for reserva in cliente.reservas_confirmadas %}
```

```
    <li>Data da reserva: {{ reserva.data_reserva }} --
        Data do evento: {{ reserva.data_evento }} --
        Quantidade de pessoas: {{ reserva.pessoas }}
    </li>
{% endfor %}
</ul>
<a href="{% url 'reservas:detalhe' cliente.id %}">
  Confirmar mais reservas?
</a>
```

Capturando parâmetros via método GET

Existem duas formas de capturar informações enviadas pelo navegador por método GET. Uma delas já foi bastante utilizada nesse curso e faz uso do sistema de URLs do Django. A outra forma utiliza um formato conhecido como query string. Esse formato é um padrão do protocolo HTTP e consiste em utilizar algumas marcas para definir parâmetros. Uma das marcas é a interrogação, que indica o início de uma query string. Logo após a interrogação, são passados nomes de variáveis e valores, separados pelo caractere &. Uma query string de exemplo pode ser:

```
'/reservas/?ordem=nome&direcao=desc'.
```

Nesse exemplo, foram definidos dois parâmetros: ordem com valor nome e direcao com valor desc.

- Via sistema de URLs do Django, já apresentado.
- Via padrão HTTP método GET, usando query strings.
- Padrão iniciado com interrogação (?), seguido de pares param=valor, separados por &.
- Exemplo: /reservas/?ordem=nome&direcao=desc

Para recuperar esses valores no Django, usamos um dicionário da requisição: request.GET. get('ordem'). O código da view index mostra um exemplo de uso:

```
# reservas/views.py
from django.http import HttpResponse
from django.shortcuts import render, get_object_or_404

from .models import Cliente

def index(request):
    ordem = request.GET.get('ordem', 'registrado_em')
    direcao = request.GET.get('direcao', 'desc')
    campos = (field.name for field in Cliente._meta.fields)
    if ordem not in campos:
        ordem = 'registrado_em'

    if direcao == 'desc':
        direcao = '-'
    else:
        direcao = ''
```

```
    ordenacao = '{}{}'.format(direcao, ordem)
    ultimos_clientes = Cliente.objects.order_by(
        ordenacao)[:5]
    context = {'ultimos_clientes': ultimos_clientes}

    return render(request, 'reservas/index.html', context)

def detalhe(request, cliente_id):
    cliente = get_object_or_404(Cliente, pk=cliente_id)
    return render(request, 'reservas/detalhe.html',
                  {'cliente': cliente})

def reservas(request, cliente_id):
    resposta = 'Foi feita uma solicitação para as reservas ' \
               ' do cliente {}.'
    return HttpResponse(resposta.format(cliente_id))
def confirma(request, cliente_id):
    resposta = 'Foi confirmada a reserva {}.'.format(cliente_id)
    return HttpResponse(resposta)
```

Atividades Práticas 

9

Arquivos estáticos, views genéricas e testes

objetivos

Prosseguir explorando o Django, usando agora arquivos de estilo (css), códigos escritos em javascript e imagens, elementos indispensáveis em modernas interfaces web. São exploradas as views genéricas, que facilitam muito o desenvolvimento de aplicações que recuperam dados e carregam um template onde a resposta esperada é renderizada. Finalmente, são explorados os testes automatizados e as ferramentas disponíveis no Django para suportar esses testes.

conceitos

Arquivos estáticos, formulários, views genéricas e testes automatizados.

Servindo arquivos estáticos

- Sistemas web completos utilizam, além do HTML, arquivos de estilo (css), códigos escritos em javascript e imagens.
- Em pequenos projetos, esses arquivos podem ficar reunidos em um único diretório.
- É um desafio gerenciar arquivos de projetos maiores, com múltiplas aplicações.
- O django.contrib.static files é um coletor que facilita esse trabalho.

Até o momento, tudo o que foi apresentado como resposta a solicitações do navegador foi conteúdo HTML gerado pelo Django via templates, views, e models. Sistemas web completos utilizam, além do HTML, arquivos de estilo (css), códigos escritos em javascript e imagens, tornando assim os resultados mais agradáveis e interativos.

Para pequenos projetos, armazenar esses arquivos em pastas do servidor web já é suficiente. Quando os projetos começam a crescer, isso pode não ser adequado. Misturar arquivos de diversas aplicações pode complicar o gerenciamento e manutenção do projeto.

O Django possui para isso uma aplicação chamada django.contrib.static, que coleta arquivos estáticos das aplicações, permitindo ainda configurar locais adicionais para seu armazenamento. Para usar essa aplicação, é necessário fazer alguns ajustes no sistema sendo desenvolvido, começando pela criação de um subdiretório static no diretório da aplicação reservas.

- Criar um subdiretório static no diretório da aplicação: reservas.
- O Django busca arquivos estáticos em diretórios static das aplicações instaladas.
- Criar um subdiretório reservas dentro do diretório static recém-criado e nele criar um arquivo style.css. Esse arquivo poderá ser referenciado como reservas/style.css nos templates.

A aplicação django.contrib.static busca por arquivos estáticos em diretórios static das aplicações instaladas. Assim como é feito com os templates, por questões práticas para evitar conflitos de nomes, um subdiretório com o mesmo nome da aplicação deve ser criado a seguir do diretório static. Em nosso exemplo deve ser criado o diretório reservas, que conterá um arquivo style.css que poderá ser referenciado como "reservas/style.css" nos templates.

Arquivos estáticos

Para observar os efeitos da customização, o ideal é começar com algo simples. Assim, o conteúdo do arquivo de estilos proposto vai apenas mudar a cor dos links usados nos itens da lista.

Conteúdo do arquivo: reservas/static/reservas/style.css:

```
li a {
    color: green;
}
```

Algumas mudanças também são necessárias no arquivo de templates. Por enquanto, para experimentar os efeitos da customização, apenas o template index.html será alterado.

Conteúdo do arquivo: reservas/templates/reservas/index.html:

```
{% load staticfiles %}
<link rel="stylesheet" type="text/css" href="{% static 'reservas/style.css' %}" />
{% if ultimos_clientes %}
    <ul>
    {% for cliente in ultimos_clientes %}
        <li>
          <a href="{% url 'reservas:detalhe' cliente.id %}">
            {{ cliente.nome }}
          </a>
        </li>
    {% endfor %}
    </ul>
{% else %}
    <p>Nenhum cliente registrado.</p>
{% endif %}
```

Duas linhas foram alteradas. A primeira linha adicionada carrega a marca {% static %} que gera a URL absoluta para arquivos estáticos.

```
{% load staticfiles %}
```

A outra alteração foi a adição da linha que carrega a folha de estilos style.css

```
<link rel="stylesheet" type="text/css" href="{% static 'reservas/style.css' %}" />
```

Essa linha utiliza a marca {% static %} para gerar a URL absoluta do arquivo "reservas/style.css".

Para conferir o resultado, basta iniciar o servidor de desenvolvimento do Django e visitar a página da aplicação reservas, http://localhost:8000/reservas/. Se todos os ajustes forem feitos corretamente, os links agora devem estar verdes.

Adicionando imagem background

- Criar subdiretório de imagens: reservas/static/reservas/imagens/
- Adicionar imagem background.gif no diretório.
- Editar arquivo de estilos style.css.

Para continuar experimentando com customizações simples, vamos agora adicionar uma imagem de fundo na página de reservas. O diretório onde a imagem será armazenada precisa ser criado, dentro da pasta de arquivos estáticos. Um bom nome para o novo diretório é "imagens".

O próximo passo é salvar uma imagem de fundo, por exemplo background.gif, no recém-criado diretório. Em seguida, é preciso alterar o arquivo de estilos para adicionar a imagem de fundo à página de reservas. A seguir, podemos observar que o código css adicional define uma imagem de fundo para o corpo do HTML.

Conteúdo do arquivo: reservas/static/reservas/style.css:

```
li a {
    color: green;
}
body {
    background: white url("imagens/background.gif") no-repeat;
}
```

Template padrão do projeto

- Com o Django é possível criar um template compartilhado entre as aplicações.
- Uma pasta templates deve ser criada na raiz do projeto, onde fica o arquivo manage.py.
- O template precisará de arquivos estáticos de css, javascript e imagens.
- Uma pasta para os arquivos estáticos deve ser criada e a variável STATICFILES_DIRS deve ser configurada no arquivo settings.py, para armazenar o nome dessa nova pasta.

Assim, é possível criar um template básico padrão para seus projetos e dar a eles melhor aparência.

Apesar da simplicidade das configurações feitas até o momento, é possível perceber a quantidade de trabalho necessário para customizar as views e templates individualmente. Uma das características mais legais do Django é que o seu sistema de templates permite fazer a extensão de um template qualquer. Mais do que isso, permite compartilhar o template entre várias aplicações do mesmo projeto.

A pasta templates no diretório do projeto precisa ser criada. O diretório do projeto é o diretório onde está o arquivo manage.py. Nessa pasta será salvo o arquivo base.html.

templates/base.html

O código a seguir é do template básico que servirá como padrão para todos os outros templates das aplicações do projeto. São carregados arquivos de estilo e definidos diversos blocos que podem ser sobrescritos nos templates das aplicações.

```
<!DOCTYPE html>
<html>
    <head>
             {% load staticfiles %}
             <meta http-equiv="Content-type"
            content="test/html; charset=UTF-8">
             <link rel="stylesheet"
            type="text/css"
            href="{% static 'css/style.css' %}" />
    </head>
    <body>
             {% block header %}
             <header></header>
             {% endblock %}
                     {% block navigation %}
             <nav>
                      <a href="{% url 'index' %}">
                               <div id="inicio" class="navButtons"></div>
                      </a>
                      <a href="{% url 'reservas:index' %}">
                      <div id="reservas" class="navButtons"></div>
             </a>
                      <div id="cardapio" class="navButtons"></div>
                      <div id="contato" class="navButtons"></div>
             </nav>
             {% endblock %}
       <section id="body">
                      <div class="conteudo">
                      {% block conteudo %}
                      {% endblock %}
                      </div>
             </section>
</html>
```

O bloco que deve ser sobrescrito frequentemente é o bloco conteudo.

templates/index.html

O template index será usado para fornecer a página inicial do site. Agora é possível perceber a importância de criar subpastas e escopo (namespace) para os templates. Isso evita confusão entre esse o template index.html do site – e o template reservas/index.html.

```
{% extends "base.html" %}
{% block conteudo %}
    Bem vindo ao seu restaurante preferido, <b> Umas e Ostras. </b>
{% endblock %}
```

O template index.html estende o template básico e apenas sobrescreve o bloco conteudo com uma mensagem de boas-vindas.

- Arquivos estáticos usados pelo template base.html:
- umas_e_ostras/static/css/style.css
- umas_e_ostras/static/images/background.png
- umas_e_ostras/static/images/cardapio.png
- umas_e_ostras/static/images/contato.png
- umas_e_ostras/static/images/inicio.png
- umas_e_ostras/static/images/logo.png
- umas_e_ostras/static/images/reservas.png

A pasta static precisa ser copiada para o diretório umas_e_ostras com os respectivos arquivos.

Configuração de arquivos estáticos

A pasta de arquivos estáticos do projeto pode ser criada na pasta onde ficam as configurações do projeto, onde também está o arquivo settings.py. O mais importante é definir a variável STATICFILES_DIRS.

```
STATICFILES_DIRS = (
        os.path.join(BASE_DIR, 'umas_e_ostras/static'),
)
```

Foi usado o método os.path.join para unir a variável BASE_DIR e o nome da pasta "umas_e_ostras/static". Essa técnica é importante, pois quando esse projeto for implantado em um servidor, não será necessário alterar essa configuração. BASE_DIR terá o caminho do projeto e será concatenado com o nome da subpasta onde os arquivos estáticos serão armazenados.

Os templates da aplicação reservas precisam ser alterados para utilizar o template do projeto. A primeira linha deve ter o comando para estender o template base. O conteúdo precisa ser gerado dentro de um bloco, chamado de conteúdo.

```
{% extends 'base.html' %}
... Qualquer html preexistente que não faz parte do conteúdo.
{% block conteudo %}
... código preexistente nos templates ...
{% endblock %}
```

Isso precisa ser aplicado a todos os templates do projeto. Em nosso exemplo, são os templates da aplicação reservas, já que o projeto só tem uma aplicação.

reservas/templates/reservas/index.html

O template index.html da aplicação reservas serve como exemplo para explicar o funcionamento da capacidade de estender templates.

```
{% extends 'base.html' %}
{% load staticfiles %}
<link rel="stylesheet" type="text/css"
 href="{% static 'reservas/style.css' %}" />
{% block conteudo %}
    {% if ultimos_clientes %}
            <ul>
            {% for cliente in ultimos_clientes %}
```

```
                    <li>
                      <a href="{% url 'reservas:detalhe' cliente.id %}">
                             {{ cliente.nome }}
                      </a>
                    </li>
            {% endfor %}
            </ul>
    {% else %}
            <p>Nenhum cliente registrado.</p>
    {% endif %}
{% endblock %}
```

O trecho antigo mantido fora do bloco conteudo não causa mais efeito algum, pois não está contido em nenhum bloco e é ignorado pelo sistema de templates.

```
{% load staticfiles %}
<link rel="stylesheet" type="text/css"
 href="{% static 'reservas/style.css' %}" />
```

Só aquilo que está no bloco conteudo será apresentado ao acessar a view que utiliza esse template.

reservas/templates/reservas/lista.html

O template lista.html teve a adição do cabeçalho com o comando para estender o template básico padrão (base.html) e a redefinição do bloco conteudo. O código anterior do template ficou dentro do bloco conteudo.

```
{% extends "base.html" %}
{% block conteudo %}
    <h1>{{ cliente.nome }}</h1>
    <ul>
    {% for reserva in cliente.reservas_confirmadas %}
            <li>Data da reserva: {{ reserva.data_reserva }} --
                    Data do evento: {{ reserva.data_evento }} --
                    Quantidade de pessoas: {{ reserva.pessoas }}
            </li>
    {% endfor %}
    </ul>
    <a href="{% url 'reservas:detalhe' cliente.id %}">
      Confirmar mais reservas?
    </a>
{% endblock %}
```

reservas/templates/reservas/detalhe.html

Da mesma forma o template detalhe.html teve apenas a adição do comando para estender o template base.html e a redefinição do bloco conteudo.

```
{% extends "base.html" %}
{% block conteudo %}
    <h1>{{ cliente.nome }}</h1>
```

```
    {% if error_message %}
     <p><strong>{{ error_message }}</strong></p>
   {% endif %}
    <form action="{% url 'reservas:confirma' cliente.id %}"
         method="post">
            {% csrf_token %}
            {% for reserva in cliente.reservas_nao_confirmadas %}
            <input type="checkbox" name="confirmacao"...
            <label for="confirmacao{{ forloop.counter }}">...
            {% endfor %}
            <input type="submit" value="Atualizar" />
    </form>
{% endblock %}
```

Importante perceber que no exemplo anterior parte do código foi omitido por ser repetido.

Se esse código for colado na aplicação, não funcionará corretamente.

umas_e_ostras/urls.py

- O projeto precisa de uma view para o novo template “templates/index.html”.
- A view genérica TemplateView pode ser usada.

A última alteração necessária para customizar a aparência do projeto é a criação da view index do site, que renderizará o template index.html.

Existe uma view genérica básica que pode ser usada para renderizar o template index. É chamada de TemplateView e serve para renderizar templates como esse, que não precisam de um model.

```
# umas_e_ostras/urls.py
from django.conf.urls import include, url
from django.contrib import admin
from django.views.generic import TemplateView

urlpatterns = [
    url('^$', TemplateView.as_view(template_name='index.html'),
        name='index'),
    url(r'^reservas/',
        include('reservas.urls', namespace='reservas')),
    url(r'^admin/',
        admin.site.urls),
]
```

Basta adicionar a definição da URL para que o template seja renderizado. Não é necessário nem mesmo escrever uma classe e fazer herança para que o funcionamento seja obtido.

```
    url('^$', TemplateView.as_view(template_name='index.html'),
        name='index'),
```

Para ver o site funcionando, basta iniciar o servidor de desenvolvimento e acessar a URL indicada no console: http://localhost:8000/.

Formulários com django.forms.Form

- Módulo django.forms possui ferramentas para manipular formulários.
- Contempla a definição dos campos, geração do HTML e captura dos dados recebidos via POST, entre outras coisas.
- Classe django.forms.Form fornece os mecanismos necessários para o funcionamento.

O framework Django também oferece facilidades para criar nossos próprios formulários. No módulo django.forms, estão disponíveis diversas classes e funções para auxiliar a criação de formulários. Vale lembrar que em aplicações do Django o conceito de formulário é um pouco mais amplo. Além do formulário HTML, existe também o objeto form (que gera o formulário HTML ou os dados estruturados retornados quando o formulário é submetido, ou ainda, o conjunto dessas partes funcionando do começo ao fim). Todo esse mecanismo é fornecido pela classe Form do módulo django.forms.

Para explicar seu uso, será criado um formulário de contato. Esse formulário precisará de uma URL para acesso e outra URL para a página de sucesso. Precisará também de uma view para tratar o formulário e outra view para renderizar a página de sucesso. O template templates/base.html precisa ser alterado para adicionar o link de acesso ao formulário de contato. Um template para o formulário de contato e um template para a página de sucesso precisam ser criados. E a classe com o formulário de contato que herda de django.forms.Form precisa ser criada. Vejamos a seguir essas alterações com as respectivas explicações.

Novo arquivo forms.py

Para definir um novo formulário, é necessário implementar uma classe que herde da classe django.forms.Form. Nessa classe são definidos os campos e tipos de cada campo do formulário.

```
# reservas/forms.py
from django import forms
class FormContato(forms.Form):
    email = forms.EmailField()
    assunto = forms.CharField(label='Assunto',
        max_length=150)
    comentarios = forms.CharField(label='Comentários',
        widget=forms.Textarea)
```

O exemplo define uma classe para um formulário de contato com os campos: e-mail, assunto e comentários. Os campos aceitam parâmetros que customizam o comportamento do formulário. O parâmetro label define a etiqueta que aparece ao lado do campo no formulário. O parâmetro max_length aplica um limite ao tamanho do campo. O parâmetro widget modifica o componente que será usado para coletar a informação no formulário. Nesse exemplo foi usado um componente de texto com várias linhas, o forms.Textarea.

Template base.html

No template base.html foi adicionado um link no menu de navegação para a URL do formulário de contato.

```
<!DOCTYPE html>
<html>
    <head>
            {% load staticfiles %}
            <meta http-equiv="Content-type" content="test/html; charset=UTF-8">
            <script src="https://ajax.googleapis.com/ajax/libs/jquery/2.1.4/
jquery.min.js"></script>
            <script src="https://ajax.googleapis.com/ajax/libs/jqueryui/1.11.4/
jquery-ui.min.js"></script>
            <link rel="stylesheet" type="text/css" href="{% static 'css/style.css'
%}" />
    </head>
    <body>
            {% block header %}
            <header></header>
            {% endblock %}
            {% block navigation %}
            <nav>
                    <a href="{% url 'index' %}">
                            <div id="inicio" class="navButtons"></div>
                    </a>
                    <a href="{% url 'reservas:index' %}">
                            <div id="reservas" class="navButtons"></div>
                    </a>
                    <div id="cardapio" class="navButtons"></div>
                    <a href="{% url 'reservas:contato' %}">
                            <div id="contato" class="navButtons"></div>
                    </a>
            </nav>
            {% endblock %}
            <section id="body">
                    <div class="conteudo">
                    {% block conteudo %}
                    {% endblock %}
                    </div>
            </section>
    </body>
</html>
```

Como os templates usados no projeto estendem esse template base.html, nenhuma outra modificação é necessária.

Template contato.html

Para apresentar o formulário, utilizamos o template contato.html. Esse template precisa definir uma tag HTML <form> para indicar qual a ação será executada quando o botão Enviar for pressionado.

```
{% extends "base.html" %}
{% block conteudo %}
    <form action="{% url 'reservas:contato' %}" method="post" class="elegant-aero">
        {% csrf_token %}
        {{ form.as_ul }}
        <input type="submit" value="Enviar" />
    </form>
{% endblock %}
```

Utilizamos a macro da linguagem de template do Django {% url %} para definir a ação. Além disso é definida uma classe CSS, elegant-aero, para customizar a aparência do formulário. O estilo elegant-aero foi definido no arquivo estático umas_e_ostras/static/css/style.css.

Template obrigado.html

Após o envio do e-mail com sucesso é apresentada uma página de agradecimento. O template obrigado.html é usado para exibir essa mensagem.

```
{% extends "base.html" %}
{% block conteudo %}
    <p>Recebemos sua mensagem. Obrigado pelo contato.</p>
    <p>Entraremos em contato em breve.</p>
{% endblock %}
```

Alteração no arquivo urls.py

As URLs de contato e agradecimento precisam ser mapeadas para as views que exibirão o formulário, processar os dados por ele enviados e apresentar a mensagem de agradecimento.

```
# reservas/urls.py
from django.conf.urls import url
from.import views
urlpatterns = [
    # ex: /reservas/
    url(r'^$', views.index, name='index'),
    # ex: /reservas/1/
    url(r'^(?P<cliente_id>[0-9]+)/$',
        views.detalhe, name='detalhe'),
    # ex: /reservas/1/lista/
    url(r'^(?P<cliente_id>[0-9]+)/lista/$',
        views.reservas, name='reservas'),
    # ex: /reservas/1/confirma/
    url(r'^(?P<cliente_id>[0-9]+)/confirma/$',
        views.confirma, name='confirma'),
```

```
    # ex: /reservas/contato/
    url(r'^contato/$', views.contato,
        name='contato'),
    # ex: /reservas/obrigado/
    url(r'^obrigado/$', views.obrigado,
        name='obrigado'),
]
```

A URL /reservas/contato/ é associada à view views.contato que exibirá o formulário e também receberá os dados para enviar o e-mail. A URL /reservas/obrigado/ é associada à view views.obrigado que exibirá a mensagem de agradecimento.

Alterações no arquivo views.py

Falta então criar as views contato e obrigado, que vão exibir o formulário em branco, processar os dados, enviar o e-mail e exibir a mensagem de agradecimento.

```
# reservas/views.py
from django.http import HttpResponse
from django.http import HttpResponseRedirect
from django.shortcuts import render
from django.shortcuts import get_object_or_404
from django.core.urlresolvers import reverse

from .models import Cliente, Reserva
from .forms import FormContato

def index(request):
    ordem = request.GET.get('ordem', 'registrado_em')
    direcao = request.GET.get('direcao', 'desc')
    campos = (field.name for field in Cliente._meta.fields)
    if ordem not in campos:
        ordem = 'registrado_em'

    if direcao == 'desc':
        direcao = '-'
    else:
        direcao = ''

    ordenacao = '{}{}'.format(direcao, ordem)
    ultimos_clientes = Cliente.objects.order_by(
        ordenacao)[:5]
    context = {'ultimos_clientes': ultimos_clientes}

    return render(request, 'reservas/index.html', context)

def detalhe(request, cliente_id):
    cliente = get_object_or_404(Cliente, pk=cliente_id)
    return render(request, 'reservas/detalhe.html',
                  {'cliente': cliente})
```

```
def reservas(request, cliente_id):
    cliente = get_object_or_404(Cliente, pk=cliente_id)
    return render(request, 'reservas/lista.html',
                 {'cliente': cliente})

def confirma(request, cliente_id):
    cliente = get_object_or_404(Cliente, pk=cliente_id)
    confirmados = request.POST.getlist('confirmacao')
    for reserva_id in confirmados:
        try:
                reserva = cliente.reserva_set.get(pk=reserva_id)
        except (KeyError, Reserva.DoesNotExist):
            # Reapresenta o formulário do cliente.
            return render(request, 'reservas/detail.html', {
                'cliente': cliente,
                'error_message': "Código da reserva não encontrado.",
            })
        else:
            reserva.confirmada = True
            reserva.save()
    # Sempre retornar uma HttpResponseRedirect depois de tratar com
    # sucesso os dados do POST. Isso evita que dados sejam postados
    # novamente caso o usuário pressione o botão voltar.
    return HttpResponseRedirect(reverse('reservas:reservas',
                                        args=(cliente.id,)))

def contato(request):
    if request.method == 'POST':
        form = FormContato(request.POST)
        if form.is_valid():
            assunto = form.cleaned_data['assunto']
            comentarios = form.cleaned_data['comentarios']
            remetente = form.cleaned_data['email']
            destinatarios = ['info@exemplo.com.br']
            try:
                send_mail(assunto, comentarios, remetente, destinatarios)
            except BadHeaderError:
                return HttpResponse('Cabeçalho inválido.')
            return HttpResponseRedirect('/reservas/obrigado/')
    else:
        form = FormContato()

    return render(request, 'reservas/contato.html', {'form': form})

def obrigado(request):
    return render(request, 'reservas/obrigado.html')
```

A view contato começa testando o método usado. Quando a URL /reservas/contato é acessada, o método é GET e o resultado do teste condicional if é falso. Portanto, o bloco executado é do else e um formulário em branco é exibido.

```
    else:
        form = FormContato()
```

Quando o formulário é preenchido e o botão Enviar for pressionado a view contato é acionada novamente. Dessa vez o método é POST, fazendo com que seja criado um objeto Form com os dados do POST. O método is_valid() valida os campos do formulário e os valores que estiverem OK são disponibilizados no atributo cleaned_data. Se todos os campos forem válidos o método retorna True, fazendo com que o e-mail seja enviado através da função send_mail. É importante tratar a exceção BadHeaderError pois existe uma técnica conhecida que utiliza os formulários de e-mail para enviar cabeçalhos de e-mail maliciosos.

```
if request.method == 'POST':
    form = FormContato(request.POST)
    if form.is_valid():
        assunto = form.cleaned_data['assunto']
        comentarios = form.cleaned_data['comentarios']
        remetente = form.cleaned_data['email']
        destinatarios = ['info@exemplo.com.br']
        try:
            send_mail(assunto, comentarios, remetente, destinatarios)
        except BadHeaderError:
            return HttpResponse('Cabeçalho inválido.')
        return HttpResponseRedirect('/reservas/obrigado/')
```

Se tudo der certo, o usuário é redirecionado para a página de agradecimento.

Usando views genéricas

- As views genéricas do Django foram criadas para resolver problemas comuns de aplicações web.
- Ao usar as views genéricas, o código repetitivo pode ser evitado ou eliminado.
- Passos necessários:
  - Ajustar as urls.
  - Substituir views antigas por views genéricas do Django.

Nas sessões anteriores, vimos como recuperar uma lista de registros do banco de dados, carregar um template e retornar uma resposta com o template renderizado. Essas são todas tarefas muito comuns em sistemas de informação. Para facilitar a realização dessas tarefas, foram criadas as views genéricas, que facilitam muito o desenvolvimento de aplicações web com funcionalidades como estas.

Até aqui usamos um método para escrever as views que permite mostrar como o Django funciona, facilitando o entendimento do que acontece de forma mais detalhada. As views genéricas permitem alcançar o mesmo objetivo, mas tornam o código mais compacto, legível e mais prático para manter. Por isso mesmo é recomendado utilizar as views genéricas sempre que possível.

Nesta sessão a aplicação que vem sendo desenvolvida será ajustada para passar a utilizar views genéricas. Para isso será preciso ajustar as URLs para fazer a associação com as views genéricas que serão utilizadas. Como resultado, algumas views antigas serão removidas e novas views, baseadas nas views genéricas do Django, serão adicionadas no arquivo views.py.

URLs das views genéricas

Os nomes das views podem até ser mantidos. O que muda é a função associada e o nome do parâmetro passado para a view.

```
# reservas/urls.py
from django.conf.urls import url

from . import views

urlpatterns = [
    # ex: /reservas/
    url(r'^$', views.IndexView.as_view(), name='index'),
    # ex: /reservas/1/
    url(r'^(?P<pk>[0-9]+)/$',
        views.DetalheView.as_view(), name='detalhe'),
    # ex: /reservas/1/lista/
    url(r'^(?P<pk>[0-9]+)/lista/$',
        views.ReservasView.as_view(), name='reservas'),
    # ex: /reservas/1/confirma/
    url(r'^(?P<cliente_id>[0-9]+)/confirma/$',
        views.confirma, name='confirma'),
    # ex: /reservas/contato/
    url(r'^contato/$', views.contato,
        name='contato'),
    # ex: /reservas/obrigado/
    url(r'^obrigado/$', views.obrigado,
        name='obrigado'),
]
```

A view index já não recebia parâmetro. A mudança nesse caso é pequena, substituindo views.index por views.IndexView.as_view(), criando a view IndexView baseada na view genérica ListView, utilizada para listar registros. Mais à frente veremos como essas views são implementadas.

Antes tínhamos:

```
    url(r'^$', views.index, name='index'),
```

Com views genéricas, temos:

```
    url(r'^$', views.IndexView.as_view(), name='index'),
```

Já as outras duas views, detalhe e reservas, recebem parâmetros que identificam o registro a ser exibido. Antes a codificação era da seguinte forma:

```
    url(r'^(?P<cliente_id>[0-9]+)/$',
        views.detalhe, name='detalhe'),
```

A view genérica que é utilizada como base é DetailView, que recebe como parâmetro a chave do registro desejado no banco de dados. Esse parâmetro, por padrão, tem como nome "pk".

```
    url(r'^(?P<pk>[0-9]+)/$',
        views.DetalheView.as_view(), name='detalhe'),
```

A view confirma é um tanto quanto particular e por isso não será reescrita usando views genéricas.

Implementação das views

As views genéricas do Django são implementadas no módulo generic do pacote django.views. Portanto, esse pacote precisa ser importado no módulo reservas.views, através do comando:

```
from django.views import generic
```

A view IndexView é baseada na view genérica generic.ListView. As views genéricas utilizam diversas padronizações e por isso apenas o modelo precisa ser informado. A view ListView, por padrão, tentará renderizar o template que segue o padrão APLICAÇÃO/MODEL_list.html. Nesse caso, o nome do template procurado seria reservas/cliente_list.html. Da mesma forma a variável de contexto é por padrão MODELO_list, o que seria cliente_list.

Mas esse comportamento pode ser customizado. O nome padrão do template pode ser sobrescrito se for definida a propriedade template_name. Definimos template_name com 'reservas/index.html'. A variável de contexto pode ser sobrescrita com a propriedade context_object_name. Como veremos a seguir, também utilizamos essa propriedade, atribuindo a mesma 'ultimos_clientes'. Finalmente, em vez de utilizar um modelo, é definido um conjunto de dados através do método get_queryset. Assim podemos limitar a quantidade de registros e definir a ordenação desejada. Vejamos então o código para implementar view IndexView.

```
class IndexView(generic.ListView):
    template_name = 'reservas/index.html'
    context_object_name = 'ultimos_clientes'

    def get_queryset(self):
        """Retorna os últimos cinco clientes registrados."""
        return Cliente.objects.order_by('-registrado_em')[:5]
```

As views DetalheView e ReservasView são similares. O template padrão dessas views é baseado na view generic.DetailView e segue o padrão APLICAÇÃO/MODELO_detail.html. A variável de contexto é inicializada com os respectivos nomes dos modelos. Nessa aplicação seriam reservas/cliente_detail.html e cliente, mas como os templates já foram criados anteriormente, inicializamos template_name com os respectivos nomes, conforme pode ser visto no trecho de código a seguir:

```
class DetalheView(generic.DetailView):
    model = Cliente
    template_name = 'reservas/detalhe.html'

class ReservasView(generic.DetailView):
    model = Cliente
    template_name = 'reservas/lista.html'
```

Atividades Práticas

Testes automatizados

- Boas práticas no desenvolvimento de aplicações recomendam que sejam feitos testes para certificar seu bom funcionamento.
- Testes podem ser realizados manualmente ou podem ser automatizados.
- Testes automatizados economizam tempo, previnem problemas, tornam o código mais confiável e ajudam no trabalho em equipe.

Boas práticas no desenvolvimento de aplicações recomendam que sejam feitos testes para certificar seu bom funcionamento. Esses testes podem ser realizados manualmente, utilizando roteiros predefinidos e técnicas de controle de qualidade, mas podem também ser automatizados.

Até o momento, as funcionalidades implementadas nas aplicações sendo desenvolvidas foram verificadas (testadas) manualmente, através da sua execução via navegador. Conforme o sistema vai crescendo, aumenta consideravelmente a quantidade de "coisas" que precisam ser verificadas. Até porque novas funcionalidades sendo implementadas podem afetar outras partes da aplicação e aquilo que já estava funcionando passa a apresentar problemas.

Os testes automatizados permitem verificar as funcionalidades do sistema, economizando tempo e prevenindo problemas. Além disso, quando várias pessoas precisam mexer no mesmo código, é necessário se preocupar com os efeitos das mudanças produzidas individualmente. Ao preparar rotinas para testar o sistema de forma automatizada, o trabalho de integração se torna mais produtivo, já que os testes podem ser repetidos e a qualidade geral aumenta. O trabalho em equipe é facilitado e o código se torna mais confiável (já que é testado repetitivamente).

Um primeiro teste manual

- Ao implementar um novo método, um teste automatizado pode ser escrito para verificar o funcionamento correto do método.
- O modelo Cliente precisa de um método para converter um número de telefone representado por letras para números (conforme os teclados de telefones).
- O teste deve verificar se um telefone com letras foi convertido corretamente.

Um novo método precisa ser implementado. Para facilitar a memorização de números de telefone, eles podem ser representados por letras: os teclados de telefone mais antigos possuem as letras que correspondem aos números. Na hora de teclar, basta encontrar a letra respectiva. A relação é:

1	2 abc	3 def
4 ghi	5 jkl	6 mno
7 pqrs	8 tuv	9 wxyz
	0	

A classe de modelo do cliente precisa de um método que converta esse número de telefone com letras para a representação em números. Por exemplo, o número (021) vaidjango deve ser convertido em (021) 824352646. Para verificar que o método funciona, um teste automatizado será escrito.

O método deve converter o número representado por letras em um número de telefone comum. Existe um método de strings (str.translate) que permite "traduzir" certos caracteres com base em uma tabela de conversão. A tabela é um dicionário com o código do caractere como chave e o caractere como valor. Mas outra função facilita a criação dessa tabela, método (str.maketrans).

```
    def telefone_numerico(self):
        teclado = str.maketrans('abcdefghijklmnopqrstuvwxyz',
                                '22233344455566677778889999')
        return self.telefone.translate(teclado)
```

O método associa as letras do teclado do telefone com os respectivos números e retorna o número convertido. Aqueles caracteres que não estão na tabela (p.e., "0", "1", espaço, parênteses etc.) são retornados sem conversão.

Para testar o funcionamento do método telefone_numerico, basta executar o código a seguir no shell do Django.

- Verificar criando um cliente com data no futuro pelo Admin.
- Ou usar o código a seguir no shell (python manage.py shell):

```
>>> from django.utils import timezone
>>> from reservas.models import Cliente
>>> cliente = Cliente(
...         nome='Dr. Emmett Brown', endereco='Hill Valley',
...         telefone='de-lore-an12', email='doc@future.com',
...         registrado_em=(timezone.now()))
>>> cliente.telefone_numerico()
'33-5673-2612'
```

Um cliente é criado com um número de telefone representado por letras. Quando o método é executado, o valor do número convertido é retornado, sem alterar o número original do cliente.

Automatizando o teste

- O teste realizado no shell pode ser convertido em um teste automatizado.
- O local mais comum para escrita de um teste é o arquivo tests.py da aplicação.
- Deve ser criada uma subclasse de django.test.TestCase com um método que tenha o nome iniciado por test, que implementa o teste automatizado.
- O sistema de testes procura por arquivos cujos nomes comecem por test e executa todos eles.

Repetir esse teste cada vez que for necessário confirmar se alguma alteração no sistema afetou o método pode se tornar dispendioso. Por isso, o mais prático seria converter esse código em um teste automatizado. O local onde os testes são escritos é o arquivo tests.py da aplicação.

O sistema de testes do Django procura por arquivos de teste cujo nome comece por test. O arquivo deve ter classes que herdam de django.test.TestCase. Todos os métodos que tenham os nomes começando por test serão executados.

Para implementar o teste de conversão do número do telefone do cliente, é necessário criar um cliente, exatamente como foi feito através do código que foi executado no shell. É esperado que a chamada do método retorne o número do telefone convertido conforme a especificação original para esse método.

```
import datetime

from django.utils import timezone
from django.test import TestCase

from .models import Cliente

class ClienteMethodTests(TestCase):

    def test_cliente_telefone_numerico(self):
        """
        telefone_numerico() deve converter
        letras em números.
        """
        horario = timezone.now()
        cliente = Cliente(
            nome='Dr. Emmett Brown',
            endereco='Hill Valley',
            telefone='de-lore-an12',
            email='doc@future.com',
            registrado_em=horario)

        self.assertEqual(cliente.telefone_numerico(),
                         '33-5673-2612')
```

O trecho de código que cria o cliente já é bem conhecido. A novidade é o método que verifica o resultado do método. É utilizado nesse caso o método assertEqual, que compara os dois valores que devem ser iguais.

Para rodar o teste, é utilizado o utilitário manage.py com o comando test e o nome da aplicação.

```
$ python manage.py test reservas
```

Todos os testes encontrados na aplicação reservas são executados (em nosso exemplo só foi implementado um teste até agora). Um relatório é exibido, conforme pode ser visto a seguir. Logo no início aparece um ponto, indicando que um teste foi executado com sucesso. Caso o teste tivesse falhado, detalhes da falha seriam apresentados, como o nome do teste que falhou (test_cliente_telefone_numerico) e a classe onde o método de teste foi implementado (reservas.tests.ClienteMethodTests). Também seria apresentado o nome do arquivo e a linha onde ocorreu a falha, bem como o valor obtido na execução do método e o valor que era esperado. Ao final, aparecem a quantidade de testes executados e o número de falhas.

```
Creating test database for alias 'default'...
.
----------------------------------------------------------------------
Ran 1 tests in 0.002s
OK
Destroying test database for alias 'default'...
```

Ampliando os testes

- Uma correção pode introduzir novos erros.
- Para verificar se novos erros foram introduzidos, outros testes podem ser criados.
- Um teste pode verificar um número de telefone sem letras.

Um sistema precisa de um conjunto considerável de testes para tentar garantir que não ocorrerão erros (e mesmo assim eles acontecem). Até porque uma correção em um determinado ponto pode gerar outros erros em lugares diferentes. Assim, vamos desenvolver um novo teste para nossa aplicação, simulando a criação de um número de telefone sem letras.

Vejamos a seguir o código que implementa o teste para garantir que o método retorna o valor adequado para clientes com número de telefone sem letras.

```
class ClienteMethodTests(TestCase):

    def test_cliente_telefone_numerico(self):
        """
        telefone_numerico() deve converter
        letras em números.
        """
        horario = timezone.now()
        cliente = Cliente(
            nome='Dr. Emmett Brown',
            endereco='Hill Valley',
            telefone='de-lore-an12',
            email='doc@future.com',
            registrado_em=horario)

        self.assertEqual(cliente.telefone_numerico(),
                         '33-5673-2612')

    def test_cliente_telefone_numerico_sem_letras(self):
        """
        telefone_numerico() sem letras deve permanecer
        correto.
        """
        horario = timezone.now()
        cliente = Cliente(
            nome='Rui Barbosa',
            endereco='Petrópolis',
            telefone='21-2001-0001',
            email='rui@barbosa.com',
```

```
            registrado_em=horario)
        self.assertEqual(cliente.telefone_numerico(),
                         '21-2001-0001')
```

Quando são criados testes, é importante tentar cobrir o maior número possível de variações. Caso aconteça algo no futuro, um erro não previsto, um novo teste deve ser criado para cobrir a nova situação.

Nesse teste, o método telefone_numerico() deve retornar o número sem alteração.

```
        self.assertEqual(cliente.telefone_numerico(),
                         '21-2001-0001')
```

Ao rodar os testes, observa-se que a correção realizada manteve o código funcionando para todos os casos pensados.

```
# python manage.py test reservas
Creating test database for alias 'default'...
..
----------------------------------------------------------------------
Ran 2 tests in 0.001s
OK
Destroying test database for alias 'default'...
```

Os testes garantem que o comportamento esperado do método telefone_numerico() para os dois casos implementados serão verificados sempre que os testes forem executados.

Cliente de testes do Django

- O Django possui um cliente de testes.
- Implementa os testes para as views e também pode ser usado no shell.
- Simula o comportamento de um usuário utilizando um navegador no nível de views.
- Exemplo: acessar a view IndexView, criar cliente e acessar novamente.

O Django possui um cliente de testes que pode ser usado para implementar os testes para as views e também pode ser usado no shell. Ele simula o comportamento de um usuário utilizando um navegador no nível de views.

Para ver essa ferramenta funcionando, será realizada uma simulação de acesso à view IndexView sem cliente. Será criado um cliente; em seguida a view IndexView será acessada novamente.

Os testes nas views servem para verificar que o comportamento esteja correto como se um usuário estivesse utilizando o site. O Django já fornece as ferramentas para criar os testes automatizados para as views.

```
>>> from django.test.utils import setup_test_environment
>>> setup_test_environment()

>>> from django.test import Client
>>> client = Client()

>>> response = client.get('/nao_existe')
Not Found: /nao_existe
```

```
>>> response.status_code
404
>>> from django.core.urlresolvers import reverse
>>> response = client.get(reverse('reservas:index'))
>>> response.status_code
200
>>> response.content
b'<!DOCTYPE html>\n<html>...</html>\n'
>>> from reservas.models import Cliente
>>> from django.utils import timezone
>>> cli = Cliente(
...     nome='Dr. Emmett Brown',
...     endereco='Hill Valley',
...     telefone='de-lore-an12',
...     email='doc@future.com',
...     registrado_em=timezone.now())
>>> cli.save()
>>> response = client.get('/reservas/')
>>> response.content
b'<!DOCTYPE html>\n<html>...Dr. Emmett Brown...</html>\n'
>>> response.context['ultimos_clientes']
[<Cliente: Dr. Emmett Brown>, <Cliente: Robson dos Santos>]
```

Após a execução do setup_test_environment(), um renderizador de templates é instalado. Isso permite que atributos adicionais sejam examinados. Vale ressaltar que esse método não inicializa um banco de dados de teste (o que for executado afetará o banco de dados da aplicação).

No shell é necessário importar a classe cliente de testes. Quando formos escrever os testes no arquivo tests.py, isso não será necessário, pois será utilizado o cliente da classe django.test.TestCase.

```
>>> from django.test import Client
>>> client = Client()
```

Com o cliente de testes inicializado, as requisições realizadas no navegador podem ser feitas com o cliente. No primeiro exemplo, é solicitada uma página que não existe, pois não foi implementada, gerando um erro 404.

```
>>> response = client.get('/nao_existe')
Not Found: /nao_existe
>>> response.status_code
404
```

A função reverse pode ser usada para obter uma URL válida da view IndexView, que deve retornar uma resposta válida com algum conteúdo, mesmo que seja uma mensagem indicando que não existem registros no banco de dados.

```
>>> from django.core.urlresolvers import reverse
>>> response = client.get(reverse('reservas:index'))
>>> response.status_code
200
>>> response.content
b'<!DOCTYPE html>\n<html>...</html>\n'
```

Ao criar um cliente, este deve constar no resultado da visão. As URLs podem ser usadas diretamente sem a necessidade de utilizar a função reverse. Entretanto, nos testes, isso pode tirar a flexibilidade do sistema de URLs. Caso ocorra uma mudança, o teste terá de ser alterado também. No shell isso não chega a ser um problema.

```
>>> response = client.get('/reservas/')
>>> response.content
b'<!DOCTYPE html>\n<html>...Dr. Emmett Brown...</html>\n'
>>> response.context['ultimos_clientes']
[<Cliente: Dr. Emmett Brown>]
```

Como pode ser observado, o cliente consta na resposta e na lista de últimos clientes, presente no contexto enviado ao template pela view.

Testes para a view detalhes

- A view detalhes poderia ser verificada criando registros e acessando as URLs para verificar se o comportamento esperado ocorre.
- O mais adequado é escrever testes automatizados.
- Um teste para verificar se a visão detalhe de clientes com um ID de cliente inexistente retorna um erro 404.
- E um teste para verificar se a visão detalhe de clientes com um cliente registrado mostra o nome do cliente.

A verificação do funcionamento da view DetalheView poderia ser feita manualmente através da interface de administração. Bastaria simular o acesso à view com um ID de cliente inexistente e conferir se um erro 404 é retornado. Outro teste poderia verificar que o nome de um cliente registrado aparece na view.

O mais indicado, entretanto, é escrever testes automatizados. Um teste deve ser criado para verificar se a visão de detalhes do cliente retorna um erro 404 (objeto não encontrado) para um cliente com ID inexistente. O outro teste vai verificar se um cliente registrado é apresentado pela view de detalhes do cliente.

Os testes da view detalhes serão implementados em uma nova classe de testes, que pode ter qualquer nome, desde que também seja derivada da classe django.test.TestCase.
A classe implementada no exemplo a seguir chama-se IndexClienteDetalhes e pode ser colocada bem no final do arquivo tests.py.

```
import datetime
from django.utils import timezone
from django.test import TestCase
from django.core.urlresolvers import reverse
from .models import Cliente

def criar_cliente(nome, dias):
    """
    Cria um cliente com o nome e data de registro com a diferença
    de dias da data de hoje, dias negativos são no passado e
    positivos são no futuro.
    """
    horario = timezone.now() + datetime.timedelta(days=dias)
```

```
    return Cliente.objects.create(
        nome=nome,
        endereco='Teste',
        telefone='teste',
        email='cliente@teste.com.br',
        registrado_em=horario
    )

class ClienteMethodTests(TestCase):

    def test_cliente_telefone_numerico(self):
        """
        telefone_numerico() deve converter
        letras em números.
        """
        horario = timezone.now()
        cliente = Cliente(
            nome='Dr. Emmett Brown',
            endereco='Hill Valley',
            telefone='de-lore-an12',
            email='doc@future.com',
            registrado_em=horario)

        self.assertEqual(cliente.telefone_numerico(),
                         '33-5673-2612')

    def test_cliente_telefone_numerico_sem_letras(self):
        """
        telefone_numerico() sem letras deve permanecer
        correto.
        """
        horario = timezone.now()
        cliente = Cliente(
            nome='Rui Barbosa',
            endereco='Petrópolis',
            telefone='21-2001-0001',
            email='rui@barbosa.com',
            registrado_em=horario)
        self.assertEqual(cliente.telefone_numerico(), '21-2001-0001')

class IndexClienteDetalhes(TestCase):
    def test_cliente_id_nao_existe(self):
        """
        A visão detalhe de clientes com id de cliente que
        não existe devem retornar um erro 404.
        """
        response = self.client.get(
            reverse('reservas:detalhe', args=(151,)))
```

```
        self.assertEqual(response.status_code, 404)

    def test_cliente_registrado(self):
        """
        Visão detalhe de clientes válido
        deve mostrar o nome do cliente.
        """
        cliente = criar_cliente(
            nome='Benta Encerrabodes de Oliveira',
            dias=-5)
        response = self.client.get(
            reverse('reservas:detalhe',
                    args=(cliente.id,)))
        self.assertContains(
            response, cliente.nome,
            status_code=200)
```

O primeiro teste verifica se o código de retorno é igual a 404 quando um ID inexistente é usado. O segundo teste é similar, mas cria um cliente e verifica se o código de retorno é 200, que indica sucesso na operação.

Testando outras views

- O processo de testes se torna repetitivo, testes similares podem ser implementados para as outras views.
- A view Reservas poderia limitar a exibição de clientes sem reserva. Dois testes poderiam ser criados, um para verificar clientes sem reserva, outro para verificar clientes com reserva.
- Em dado momento os testes podem precisar ser atualizados, quando a lógica do sistema mudar. Os testes falharão e ficará clara essa necessidade.

As views são similares e os seus testes também acabam ficando parecidos. O ideal é repetir o processo de expandir a cobertura de teste. A view Reservas, por exemplo, poderia limitar a exibição de clientes que não possuem reserva. Assim, um teste poderia verificar se clientes com reserva são exibidos. Outro teste poderia verificar se clientes sem reserva geram um erro 404 (objeto não encontrado).

Sempre que a lógica do sistema sofrer alterações, os seus testes precisarão ser atualizados.

Regras de ouro para testes

- Criar classes TestClass separadas para cada modelo ou visão.
- Um método separado para cada conjunto de condições a ser testada.
- Os nomes dos métodos de teste deve descrever claramente a função do teste.

É importante organizar e gerenciar os testes de um sistema, criando uma classe separada de testes para cada modelo ou visão. Cada conjunto de condições ou funcionalidade deve ser testada em um método de teste separado. Isso facilita no caso de falhas, pois o relatório de teste vai indicar claramente qual teste falhou.

O relatório de teste também apresenta o nome do teste que falhou. Se o nome utilizado indicar claramente a intenção do teste, isso vai facilitar a compreensão do problema a ser resolvido. Além disso, aqueles que estiverem lendo o teste terão informações claras sobre a intenção por trás de cada teste.

Outros tipos de teste

- Django inclui a classe LiveServerTestCase para facilitar integração com ferramentas de testes com navegador web, como o framework Selenium.
- Para rodar automaticamente os testes, é muito comum que sejam usadas ferramentas de integração contínua.
- Para verificar a cobertura de testes sobre o código, são utilizadas ferramentas como o coverage.py, inclusive ajudando a identificar partes mortas do código.

Existem muitos outros conceitos e tipos de testes. O Django inclui uma classe especial para integração com ferramentas de testes que utilizam navegadores web para simular o funcionamento do site, tais como o framework Selenium.

Um conceito muito utilizado por equipes que adotam métodos ágeis é a integração contínua. As ferramentas utilizadas para integração contínua rodam os testes automaticamente. Entre elas temos o Jenkins e Travis, duas das mais conhecidas.

Para obter informações sobre as partes do código que possuem ou não testes, são usadas ferramentas de verificação de cobertura de testes. Para código escrito em Python, existe uma ferramenta chamada coverage.py.

Atividades Práticas 

10

Entrega/Manutenção da aplicação

objetivos

Colocar uma aplicação Django em funcionamento usando um servidor web que, para isso, precisa ser corretamente configurado. Como todo sistema que entra em produção, este precisará ser eventualmente corrigido e provavelmente evoluirá. Será explorado também o uso do Depurador (Debugger) de código disponível em Python.

conceitos

WSGI, ambiente virtual e debugger

Implantação em servidor web

- Existem várias maneiras de implantar um projeto Django, mas a mais indicada atualmente é WSGI.
- O comando startproject prepara uma configuração WSGI simples compatível com servidores de aplicação com suporte a WSGI.
- Será usado um ambiente virtual para ser utilizado pelo servidor.
- Para implantar o projeto com o Apache, o módulo mod-wsgi precisa ser instalado.

É possível implantar um projeto Django utilizando diversas técnicas e tecnologias. Atualmente, a forma mais indicada é utilizando um servidor que suporte o "Web Server Gateway Interface", também conhecido como WSGI. Trata-se de uma especificação que descreve como um servidor se comunica com aplicações web.

Ao executar o comando "django-admin startproject...", um arquivo wsgi.py é gerado com uma configuração básica compatível com servidores de aplicação com suporte a WSGI.

Não é incomum desenvolver projetos com Python, que individualmente utilizam versões diferentes de uma mesma biblioteca. Em um outro cenário, um projeto Python antigo pode demandar uma versão de biblioteca diferente daquela disponibilizada pelo Sistema Operacional da máquina onde está instalada. Para resolver esse tipo de problema, os desenvolvedores Python lançam mão de ambientes virtuais que são criados para a implantação do projeto no servidor web.

Neste curso, o servidor web que será usado é o Apache, que suporta o padrão WSGI. Mas para que tudo funcione corretamente, é necessário instalar o módulo mod-wsgi do Apache.

Criando um ambiente virtual

A criação de um ambiente virtual demanda a instalação das seguintes bibliotecas:

- pip, que é um gerenciador de pacotes Python.
- python3-dev, que são os cabeçalhos usados para compilação de algumas bibliotecas.
- build-essential, que são ferramentas para compilação de código-fonte em sistemas debian gnu/linux.

O comando usado para instalar essas bibliotecas é:

```
$ sudo apt-get install python3-pip python3-dev build-essential
```

Para facilitar o gerenciamento de ambientes virtuais, será usado o gerenciador virtualenvwrapper. Ao instalá-lo, as dependências necessárias, tais como o pacote virtualenv, também serão instaladas.

O comando usado para instalar o gerenciador de ambientes virtuais é:

```
$ sudo pip3 install virtualenvwrapper
```

- Para criar um Ambiente Virtual:
- Instalar o gerenciador de pacotes Python, pip:
- $ sudo apt-get install python3-pip python3-dev build-essential
- Instalar o gerenciador de ambientes virtuais, virtualenvwrapper.
- $ sudo pip3 install virtualenvwrapper
- Criar um ambiente virtual para o projeto:
- $ mkvirtualenv umas_e_ostras -p /usr/bin/python3.4

Em seguida, será criado um ambiente virtual para o projeto utilizando a versão do Python recomendada, com o comando:

```
$ mkvirtualenv umas_e_ostras -p /usr/bin/python3.4
```

O ambiente virtual é ativado tão logo ele é criado. Para identificar sua localização, podemos usar o comando pip e a opção –V, que informa a versão e o diretório onde estão as bibliotecas do Python.

```
$ pip -V
pip 1.5.6 from /home/aluno/.virtualenvs/umas_e_ostras/lib/python3.4/site-packages
(python 3.4)
```

- Verificar o local de instalação:
  - $ pip -V
  - pip 1.5.6 from /home/aluno/.virtualenvs/umas_e_ostras/lib/python3.4/site-packages (python 3.4)
- Instalar o Django no ambiente virtual:
  - $ pip install django
- Instalar o servidor Web apache com o módulo wsgi:
  - $ sudo aptitude install apache2 libapache2-mod-wsgi-py3

Esse ambiente virtual ainda não tem as bibliotecas do Django instaladas. Para instalar, utiliza-se o comando:

```
$ pip install django
```

Falta então instalar o servidor web e o módulo para adicionar o suporte ao padrão WSGI. Isso é feito através do seguinte comando:

```
$ sudo aptitude install apache2 libapache2-mod-wsgi-py3
```

Servidor web

O próximo passo é copiar o projeto para uma pasta do servidor web. Se o servidor estivesse hospedado em outro local, seria necessário enviar o projeto para este outro local. A cópia é feita com o comando:

```
$ sudo cp -R umas_e_ostras /var/www/
```

A aplicação admin precisa dos arquivos estáticos padrão que vêm com a instalação do Django. Por isso, também é preciso copiar esses arquivos para o servidor web. O servidor Apache roda com as permissões do usuário www-data; portanto, é necessário alterar o proprietário da pasta umas_e_ostras com o comando chwon. Esse conjunto de passos é apresentado a seguir:

```
$ cd /var/www/umas_e_ostras
$ sudo cp -r ~/.virtualenvs/umas_e_ostras/lib/python3.4/site-packages/django/
contrib/admin/static/admin/ umas_e_ostras/static/
$ sudo chown -R www-data.www-data /var/www/umas_e_ostras
```

Copiar o projeto para a pasta do servidor web:

- $ sudo cp -R umas_e_ostras /var/www/

Copiar os arquivos estáticos do admin para o servidor web:

- $ cd /var/www/umas_e_ostras
- $ sudo cp -r ~/.virtualenvs/umas_e_ostras/lib/python3.4/site-packages/django/contrib/admin/static/admin/ umas_e_ostras/static/
- $ sudo chown -R www-data.www-data /var/www/umas_e_ostras

Configurar o servidor Apache:

- /etc/apache2/sites-enabled/001-umas_e_ostras.conf

Finalmente, é preciso configurar o servidor Apache. Recomenda-se criar um arquivo separado com a configuração do projeto. O nome do arquivo não afeta o funcionamento da configuração, mas para facilitar o gerenciamento do Apache, pode ser usado um número sequencial e o nome do projeto. O local correto para o arquivo é a pasta sites-enabled do Apache, conforme o exemplo:

```
 /etc/apache2/sites-enabled/001-umas_e_ostras.conf
```

No trecho de código a seguir, apresentamos o conteúdo do arquivo de configuração.

```
Alias /static/ /var/www/umas_e_ostras/umas_e_ostras/static/
<Directory /var/www/umas_e_ostras/umas_e_ostras/static>
    Require all granted
```

```
</Directory>
WSGIScriptAlias / /var/www/umas_e_ostras/umas_e_ostras/wsgi.py
WSGIPythonPath /var/www/umas_e_ostras/:/home/aluno/.virtualenvs/umas_e_ostras/lib/
python3.4/site-packages
<Directory /var/www/umas_e_ostras/umas_e_ostras>
    <Files wsgi.py>
        Require all granted
    </Files>
</Directory>
```

A primeira linha cria um apelido (alias) para os arquivos estáticos. Sempre que a URL /static/ for acessada, o servidor saberá que os arquivos estão na pasta:

```
 /var/www/umas_e_ostras/umas_e_ostras/static/.
```

A linha WSGIScriptAlias define uma espécie de apelido especial. Essa linha indica que o Apache deve servir requisições a seguir da URL usando o script indicado. Nesse caso, é a URL "/" e o script "wsgi.py".

A linha do WSGIPythonPath configura o caminho de importação de bibliotecas do Python. Daí a necessidade de descobrir o local das bibliotecas do ambiente virtual.

Entendido o conteúdo do arquivo de configuração, falta apenas reiniciar o servidor Apache. Isso é feito com o comando:

```
$ sudo service apache2 restart
```

Foram configuradas duas pastas, a pasta do projeto, "/var/www/umas_e_ostras", e a pasta do ambiente virtual, "/home/aluno/.virtualenvs/umas_e_ostras/lib/python3.4/site-packages". O nome da pasta pode variar, pois depende do nome do usuário que rodou o comando ao criar o ambiente virtual. Em caso de dúvida, rodar o comando pip -V novamente.

Atividades Práticas 

Depurando aplicações Django

Uma das atividades mais comuns no desenvolvimento de software é buscar identificar a origem de problemas no código-fonte dos programas. Em aplicações Python é muito comum utilizar o Python Debugger - pdb. Existem muitas formas de utilizá-lo, mas uma das mais usadas é colocar o seguinte código no ponto onde se deseja investigar:

```
import pdb; pdb.set_trace
```

- Python debugger: pdb.

Adicionar ponto de parada no código:

```
import pdb; pdb.set_trace()
```

Pacote django-pdb. Instalar pacote:

```
$ pip install django-pdb
```

Deve ser adicionada django_pdb às aplicações instaladas no arquivo settings.py em INSTALLED_APPS.

Deve ser adicionada django_pdb.middleware.PdbMiddleware em MIDDLEWARE_CLASSES

Outra técnica possível em projetos desenvolvidos com Django é utilizar o pacote django-pdb. Primeiro, é necessário instalar o pacote, com o comando:

```
$ pip install django-pdb
```

Em seguida, a aplicação precisa ser adicionada ao arquivo settings.py, mais precisamente nos seguintes dois lugares: INSTALLED_APPS e MIDDLEWARE_CLASSES. Vejamos esses trechos do arquivo settings.py a seguir:

```
"""
Django settings for umas_e_ostras project.
# Application definition
INSTALLED_APPS = (
    'django_pdb',
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'reservas',
)
MIDDLEWARE_CLASSES = (
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.auth.middleware.SessionAuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
    'django.middleware.security.SecurityMiddleware',
    'django_pdb.middleware.PdbMiddleware',
)
```

Quando o modo de desenvolvimento estiver habilitado, DEBUG = True, o django-pdb já pode ser usado. Uma alternativa é acessar a URL de uma view passando um parâmetro GET chamado pdb, por exemplo, /reservas/contato/?pdb. O navegador ficará aguardando uma resposta. No terminal, o debug será aberto e ficará aguardando comandos, como pode ser visto a seguir na figura 10.1.

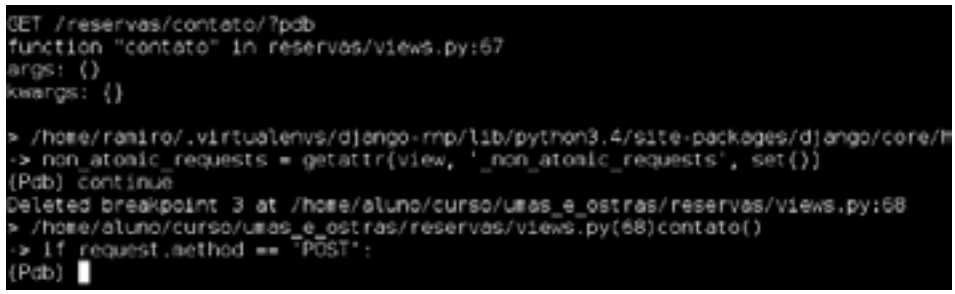

Figura 10.1
Debugger com parâmetro GET.

O primeiro ponto de parada é um gatilho do próprio pacote. Se for digitado o comando continue, o debug vai parar na view da aplicação associada à URL digitada.

Para obter ajuda, basta digitar o comando help. Será exibida uma lista com os comandos disponíveis, conforme pode ser visto na figura 10.2.

```
(Pdb) help

Documented commands (type help <topic>):
========================================
EOF     c          d        h         list      q        rv       undisplay
a       cl         debug    help      ll        quit     s        unt
alias   clear      disable  ignore    longlist  r        source   until
args    commands   display  interact  n         restart  step     up
b       condition  down     j         next      return   tbreak   w
break   cont       enable   jump      p         retval   u        whatis
bt      continue   exit     l         pp        run      unalias  where

Miscellaneous help topics:
==========================
pdb  exec
```

Figura 10.2
Lista de comandos disponíveis.

Para obter ajuda de um comando específico, basta digitar help e o nome do comando. Na figura 10.3 são mostrados exemplos com os comandos next, step e list. O comando next serve para executar a instrução atual. O comando step "entra" na função ou método. O comando list mostra algumas linhas de código, antes e depois da instrução atual.

```
(Pdb) help next
n(ext)
        Continue execution until the next line in the current function
        is reached or it returns.
(Pdb) help step
s(tep)
        Execute the current line, stop at the first possible occasion
        (either in a function that is called or in the current
        function).
(Pdb) help list
l(ist) [first [,last] | .]

        List source code for the current file.  Without arguments,
        list 11 lines around the current line or continue the previous
        listing.  With . as argument, list 11 lines around the current
        line.  With one argument, list 11 lines starting at that line.
        With two arguments, list the given range; if the second
        argument is less than the first, it is a count.

        The current line in the current frame is indicated by "->".
        If an exception is being debugged, the line where the
        exception was originally raised or propagated is indicated by
        ">>", if it differs from the current line.
(Pdb)
```

Figura 10.3
Ajuda dos comandos next, step e list.

Na figura 10.4 é apresentado um exemplo de execução do comando list. Por padrão, são exibidas 10 linhas, sendo 5 anteriores e 5 posteriores em relação à instrução/comando atual, que é marcada com uma seta.

```
> /home/aluno/curso/umas_e_ostras/reservas/views.py(68)contato()
-> if request.method == 'POST':
(Pdb) list
 63         return HttpResponseRedirect(reverse('reservas:reservas',
 64                                             args=(cliente.id,)))
 65
 66
 67     def contato(request):
 68 B->     if request.method == 'POST':
 69             form = FormContato(request.POST)
 70             if form.is_valid():
 71                 assunto = form.cleaned_data['assunto']
 72                 comentarios = form.cleaned_data['comentarios']
```

Figura 10.4
Exemplo do comando list.

O comando continue retoma a execução normal da aplicação, conforme pode ser observado na figura 10.5. Não é exibido o registro das operações posteriores ao comando continue. O formulário foi exibido, preenchido e enviado ao servidor. Após o processamento, houve o redirecionamento para a página de agradecimento.

```
(Pdb) continue
[26/Dec/2015 20:13:40]"GET /reservas/contato/?pdb HTTP/1.1" 200 1400
[26/Dec/2015 20:15:12]"POST /reservas/contato/ HTTP/1.1" 302 0
[26/Dec/2015 20:15:12]"GET /reservas/obrigado/ HTTP/1.1" 200 912
```

Figura 10.5
Fluxo normal da aplicação.

Outra forma de utilizar o debugger é passando um parâmetro para o servidor de desenvolvimento no momento da inicialização:

```
$ python manage.py runserver --pdb
```

Dessa forma, o debugger entrará em ação quando forem realizadas requisições para o servidor. Sempre que uma requisição for realizada, o debugger adicionará um ponto de parada na view associada à URL. Essa forma de utilização pode ser vista na figura 10.6:

```
GET /reservas/contato/
function "contato" in reservas/views.py:67
args: ()
kwargs: {}

> /home/ramiro/.virtualenvs/django-rnp/lib/python3.4/site-packages/django/cor
-> non_atomic_requests = getattr(view, '_non_atomic_requests', set())
(Pdb) continue
Deleted breakpoint 4 at /home/aluno/curso/umas_e_ostras/reservas/views.py:68
> /home/aluno/curso/umas_e_ostras/reservas/views.py(68)contato()
-> if request.method == 'POST':
(Pdb) request
<WSGIRequest: GET '/reservas/contato/'>
(Pdb) continue
[26/Dec/2015 20:21:02]"GET /reservas/contato/ HTTP/1.1" 200 1400

POST /reservas/contato/
function "contato" in reservas/views.py:67
args: ()
kwargs: {}

> /home/ramiro/.virtualenvs/django-rnp/lib/python3.4/site-packages/django/cor
-> non_atomic_requests = getattr(view, '_non_atomic_requests', set())
(Pdb) continue
Deleted breakpoint 5 at /home/aluno/curso/umas_e_ostras/reservas/views.py:68
> /home/aluno/curso/umas_e_ostras/reservas/views.py(68)contato()
-> if request.method == 'POST':
(Pdb) request
<WSGIRequest: POST '/reservas/contato/'>
(Pdb)
```

Figura 10.6
Rodando o debugger com o servidor de desenvolvimento.

Na figura 10.6, podemos perceber que após a execução do comando continue o debugger para novamente na view contato. Essa parada permite capturar o envio (POST) do formulário.

O debuger permite também acessar as variáveis que estão disponíveis no escopo da view. Na figura 10.7, foi primeiramente acessada a requisição request. Após alguns comandos next, antes de executar o método form.is_valid(), verificamos que ainda não existiam dados por meio da expressão 'cleaned_data' (dados limpos). Isso porque eles são criados pelo método form.is_valid(), e logo após sua execução, percebemos que os dados estão disponíveis com os valores enviados pelo navegador.

Figura 10.7
Verificando valores de variáveis no debugger.

Outra ferramenta muito útil para depurar código chama-se django debug toolbar. Para instalar essa ferramenta, basta executar o comando:

```
$ pip install django-debug-toolbar
```

Ferramenta de debug integrada à página.

- $ pip install django-debug-toolbar

Deve ser adicionada debug_toolbar às aplicações instaladas no arquivo settings.py em INSTALLED_APPS.

Após a configuração, as tabelas da aplicação precisam ser criadas com o comando migrate.

- $ python manage.py migrate

Para utilizar essa ferramenta, esta precisa ser adicionada ao arquivo settings.py na opção INSTALLED_APPS.

```
# Application definition
INSTALLED_APPS = (
    'django_pdb',
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'debug_toolbar',
    'reservas',
)
```

Para que ela funcione corretamente, é necessário ainda criar as tabelas da aplicação, com o seguinte comando:

```
$ python manage.py migrate
```

Se todos os passos foram executados sem problemas, a barra de debug do django estará habilitada. É preciso certificar-se de que o parâmetro DEBUG = True está ativado nas configurações de desenvolvimento. A barra de Debug será exibida no canto direito da página da aplicação. Se estiver no modo discreto, ela aparece na forma de um botão com as iniciais DjDT, como pode ser visto na figura 10.8.

Figura 10.8
Botão discreto da barra de debug.

No modo expandido, a barra é exibida com uma série de opções clicáveis, conforme podemos constatar na figura 10.9.

Figura 10.9
Debug toolbar expandida.

Se clicarmos em qualquer uma das opções da barra, são exibidas as informações adicionais correspondentes e que estão relacionadas ao ambiente do Django, além da view que foi acessada. Na figura 10.10, é exibido um exemplo de acesso SQL executado ao acessar a view index.

Figura 10.10
Exemplo de debug de consultas SQL.

Atividades Práticas AVA

Ramiro B. da Luz é Bacharel em informática. Iniciou a carreira de desenvolvimento de sistemas em 1991. Realizou mestrado em 2013 pela Universidade Tecnológica Federal do Paraná, onde foi pesquisador da área de Engenharia de Software com ênfase em métodos ágeis do Programa de Pós-Graduação em Computação Aplicada. Programador concursado da Câmara Municipal de Curitiba. Fundador do grupo de usuários de Python do Paraná e um dos fundadores do grupo dojo Paraná. Anfitrião da PythonBrasil[6] em 2010, membro da Associação Python Brasil.

esr.rnp.br

LIVRO DE APOIO AO CURSO

O curso apresenta um breve histórico da criação e evolução da linguagem de programação Python. Conceitos relacionados com orientação a objetos são explorados, oferecendo um conjunto sólido de conhecimentos que são a base indispensável para começar a desenvolver sistemas em Python. Em seguida é apresentado o framework Django, que permite acelerar o ciclo de desenvolvimento de aplicações e sistemas em Python, passando pela criação de um projeto, configuração inicial, implementação de suas funcionalidades e a utilização da API de banco de dados. Ao final do curso são apresentados os conceitos de configuração de servidor e depuração de aplicações Django, cobrindo a etapa de colocação em produção e manutenção de aplicações escritas em Python/Django.