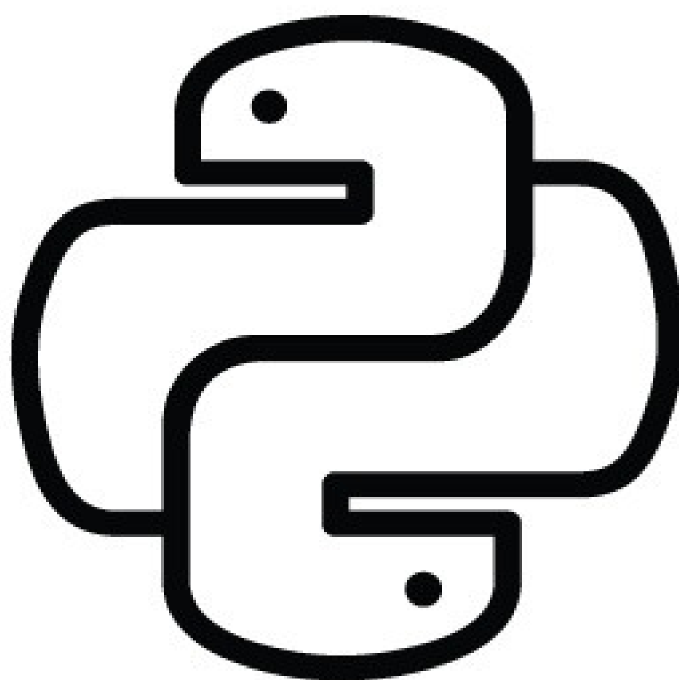
Python e Orientação a Objetos

Curso PY-14



caelum

ensino e inovação

caelum

ensino e inovação

Conheça também:

alura.com.br

casadocodigo.com.br

Blog da Caelum

blog.caelum.com.br

Newsletter

caelum.com.br/newsletter

Facebook

facebook.com.br/caelumbr

Twitter

twitter.com/caelum

Sumário

1 Como aprender Python **2**
1.1 O que é realmente importante? 2
1.2 Sobre os exercícios 3
1.3 Tirando dúvidas e indo além 3

2 O que é Python **4**
2.1 Python 4
2.2 Breve História 4
2.3 Interpretador 5
2.4 Qual versão utilizar? 6
2.5 Download 7
2.6 Cpython, Jython, IronPython? 7
2.7 PEP - O que são e pra que servem 8
2.8 Onde usar e objetivos 8
2.9 Primeiro programa 9
2.10 Modo Interativo 9
2.11 Modo Script 10
2.12 Exercício: Modificando o programa 12
2.13 O que pode dar errado? 12

3 Variáveis e tipos embutidos **15**
3.1 Tipos embutidos (built-ins) 15
3.2 Variáveis 16
3.3 Para saber mais: Nomes de variáveis 17
3.4 Instruções 18
3.5 Operadores Aritméticos 19
3.6 Strings 20
3.7 Entrada do usuário 21
3.8 Constantes 23
3.9 Comando if 26

3.10 Convertendo uma string para inteiro 27
3.11 O comando elif 28
3.12 Exercícios - Jogo da adivinhação 29
3.13 Comando while 31
3.14 Exercícios - Jogo com while 32
3.15 Comando for 34
3.16 Exercícios - Utilizando o for no jogo 36

4 Estrutura de dados 38
4.1 Exercícios: Jogo da Forca 42
4.2 Sequências 45
4.3 Conjuntos 50
4.4 Dicionários 51
4.5 Exercícios: Estrutura de dados 53

5 Funções 55
5.1 O que é uma função? 55
5.2 Parâmetros de Função 56
5.3 Função com retorno 57
5.4 Retornando múltiplos valores 58
5.5 Exercícios: Funções 59
5.6 Número arbitrário de parâmetros (*args) 60
5.7 Número arbitrário de chaves (**kwargs) 61
5.8 Exercício - *args e **kwargs 62
5.9 Exercício - Função jogar() 63
5.10 Módulos e o comando import 63

6 Arquivos 65
6.1 Escrita de um arquivo 65
6.2 Fechando um arquivo 66
6.3 Escrevendo palavras em novas linhas 66
6.4 Exercicios 67
6.5 Lendo um arquivo 68
6.6 Lendo linha por linha do arquivo 69
6.7 Gerando um número aleatório 70
6.8 Exercícios - Leitura de arquivos 71
6.9 Para saber mais - comando with 73
6.10 Melhorando nosso código 73
6.11 Exercício - Refatorando o jogo da Forca 74

7 Orientação a Objetos **81**
7.1 Funcionalidades 82
7.2 Exercício: Criando uma conta 83
7.3 Classes e Objetos 84
7.4 Construtor 86
7.5 Métodos 87
7.6 Métodos com retorno 89
7.7 Objetos são acessados por referência 90
7.8 Método transfere 92
7.9 Continuando com atributos 93
7.10 Tudo é objeto 95
7.11 Composição 96
7.12 Para saber mais: outros métodos de uma classe 98
7.13 Exercício: Primeira classe Python 99

8 Modificadores de acesso e métodos de classe **102**
8.1 Encapsulamento 105
8.2 Atributos de classe 109
8.3 Métodos de classe 112
8.4 Para saber mais - Slots 113
8.5 Exercícios: 115

9 Pycharm **117**
9.1 IDE 117
9.2 Pycharm 117
9.3 Download e Instalação do PyCharm 118
9.4 Criando um Projeto 119
9.5 Criando uma classe 122
9.6 Executando código 124
9.7 Criando métodos 126
9.8 Principais Atalhos 127
9.9 Exercício - Criando projeto banco no PyCharm 128

10 Herança e Polimorfismo **132**
10.1 Repetindo código? 132
10.2 Reescrita de métodos 136
10.3 Invocando o método reescrito 137
10.4 Para saber mais - Métodos Mágicos 140
10.5 Polimorfismo 140

10.6 Duck Typing 143
10.7 Exercício: Herança e Polimorfismo 145
10.8 Classes Abstratas 148
10.9 Exercícios - classes abstratas 150

11 Herança Múltipla e Interfaces 153
11.1 Problema do diamante 155
11.2 Mix-ins 158
11.3 Para sabe mais - Tkinter 159
11.4 (Opcional) Exercícios - Mix-Ins 160
11.5 Intefaces 162
11.6 Exercícios - Interfaces e classes Abstratas 165

12 Exceções e Erros 169
12.1 Exceções e tipos de erros 174
12.2 Tratando Exceções 176
12.3 Levantando exceções 177
12.4 Definir uma Exceção 177
12.5 Para saber mais: finally 179
12.6 Árvore de Exceções 180
12.7 Exercícios: Exceções 181
12.8 Outros Erros 182
12.9 Para saber mais - depurador do Python 183

13 Collections 184
13.1 UserList, UserDict e UserString 184
13.2 Para saber mais 186
13.3 Collections abc 188
13.4 Construindo um Container 189
13.5 Sized 190
13.6 Iterable 191
13.7 Exercício: Criando nossa Sequência 194

14 Apêndice - Python2 ou Python3? 198
14.1 Quais as diferenças? 198
14.2 A função print() 199
14.3 A função input() 199
14.4 Divisão decimal 199
14.5 Herança 200

15 Apêndice - Instalação 201

15.1 Instalando o Python no Windows 201
15.2 Instalando o Python no Linux 205
15.3 Instalando o Python no MacOS 205
15.4 Outras formas de utilizar o Python 206

Versão: 22.8.23

CAPÍTULO 1

COMO APRENDER PYTHON

1.1 O QUE É REALMENTE IMPORTANTE?

Muitos livros, ao passar dos capítulos, mencionam todos os detalhes da linguagem, juntamente com seus princípios básicos. Isso acaba criando muita confusão, em especial porque o estudante não consegue diferenciar exatamente o que é essencial aprender no início, daquilo que pode ser deixado para estudar mais tarde.

Se uma classe abstrata deve ou não ter ao menos um método abstrato, se o *if* somente aceita argumentos booleanos e todos os detalhes sobre classes internas, realmente não devem ser preocupações para aquele cujo objetivo primário é aprender Python. Esse tipo de informação será adquirida com o tempo e não é necessária no início.

Neste curso, separamos essas informações em quadros especiais, já que são informações extras. Ou então, apenas citamos em algum exercício e deixamos para o leitor procurar informações adicionais, se for de seu interesse.

Por fim, falta mencionar algo sobre a prática, que deve ser tratada seriamente: todos os exercícios são muito importantes e os desafios podem ser feitos após o término do curso. De qualquer maneira, recomendamos aos alunos estudarem em casa e praticarem bastante código e variações.

Seus livros de tecnologia parecem do século passado?

Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.
Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

Casa do Código, Livros de Tecnologia.

1.2 SOBRE OS EXERCÍCIOS

Os exercícios do curso variam, de práticos até pesquisas na internet, ou mesmo consultas sobre assuntos avançados em determinados tópicos, para incitar a curiosidade do aprendiz na tecnologia.

Existe também, em determinados capítulos, uma série de desafios. Eles focam mais no problema computacional que na linguagem, porém são uma excelente forma de treinar a sintaxe e, principalmente, familiarizar o aluno com as bibliotecas padrões do Python, além de proporcionar um ganho na velocidade de desenvolvimento.

1.3 TIRANDO DÚVIDAS E INDO ALÉM

Para tirar dúvidas de exercícios, ou de Python em geral, recomendamos o fórum do GUJ Respostas:

http://www.guj.com.br

Lá sua dúvida será respondida prontamente. O GUJ foi fundado por desenvolvedores da Caelum e hoje conta com mais de um milhão de mensagens.

O principal recurso oficial para encontrar documentação, tutoriais e até mesmo livros sobre Python é a Python Software Foundation (PSF):

https://www.python.org/

Destacamos também a página da comunidade no Brasil:

https://python.org.br/

Há também fóruns oficiais da comunidade:

https://python-forum.io/ (inglês)

https://python.org.br/lista-de-discussoes/ (português)

Fora isso, sinta-se à vontade para entrar em contato com seu instrutor para tirar todas as dúvidas que surgirem durante o curso.

Se o que você está buscando são livros de apoio, sugerimos conhecer a editora Casa do Código:

https://www.casadocodigo.com.br/

Há também cursos online que vão ajudá-lo a ir além, com muita interação com os instrutores:

https://www.alura.com.br/

CAPÍTULO 2

O QUE É PYTHON

2.1 PYTHON

Python é uma linguagem de programação interpretada, orientada a objetos, de alto nível e com semântica dinâmica. A simplicidade do Python reduz a manutenção de um programa. Python suporta módulos e pacotes, que encoraja a programação modularizada e reuso de códigos.

É uma das linguagens que mais tem crescido devido sua compatibilidade (roda na maioria dos sistemas operacionais) e capacidade de auxiliar outras linguagens. Programas como *Dropbox*, *Reddit* e *Instagram* são escritos em Python. Python também é a linguagem mais popular para análise de dados e conquistou a comunidade científica.

Mas antes que você se pergunte o que cada uma dessas coisas realmente significa, vamos começar a desbravar o mundo Python e entender como funciona essa linguagem de programação que tem conquistado cada vez mais adeptos.

Agora é a melhor hora de aprender algo novo

alura

Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

Conheça a Alura Cursos Online.

2.2 BREVE HISTÓRIA

Python foi criada em 1990 por Guido Van Rossum no Centro de Matemática Stichting (CWI, veja http://www.cwi.nl) na Holanda como uma sucessora da linguagem ABC. Guido é lembrado como o principal autor de Python mas outros programadores ajudaram com muitas contribuições.

A linguagem ABC foi desenhada para uso de não programadores, mas logo de início mostrou certas

limitações e restrições. A maior reclamação dos primeiros alunos não programadores dessa linguagem era a presença de regras arbitrárias que as linguagens de programação haviam estabelecido tradicionalmente - muita coisa de baixo nível ainda era feita e não agradou o público.

Guido então se lançou na tarefa de criar uma linguagem de script simples que possuísse algumas das melhores propriedades da ABC. Listas Python, dicionários, declarações básicas e uso obrigatório de indentação - conceitos que aprenderemos neste curso - diferenciam Python da linguagem ABC. Guido pretendia que Python fosse uma segunda linguagem para programadores C ou C++ e não uma linguagem principal para programadores - o que mais tarde se tornou para os usuários de Python.

Em 1995, Guido continuou seu trabalho em Python na Corporation for National Research Initiatives (CNRI, veja http://www.cnri.reston.va.us/) in Reston, Virginia onde ele lançou outras versões da linguagem.

Em maio de 2000, Guido e o time principal de Python se mudaram para a BeOpen.com para formar o time BeOpen PythonLabs. Em outubro do mesmo ano, o time da PythonLabs se moveu para a Digital Creations (hoje, Zope Corporation, veja http://www.zope.org/). Em 2001, a Python Software Foundation (PSF, veja http://www.python.org/psf/), uma organização sem fins lucrativos, foi formada especialmente para manter a linguagem e hoje possui sua propriedade intelectual. A Zope Corporation é um membro patrocinador da PSF.

Todos os lançamentos de Python são de código aberto (veja http://www.opensource.org).

2.3 INTERPRETADOR

Você provavelmente já ouviu ou leu em algum lugar que Python é uma linguagem interpretada ou uma linguagem de script. Em certo sentido, também é verdade que Python é tanto uma linguagem interpretada quanto uma linguagem compilada. Um compilador traduz linguagem Python em linguagem de máquina - código Python é traduzido em um código intermediário que deve ser executado por uma máquina virtual conhecida como PVM (Python Virtual Machine). É muito similar ao Java - há ainda um jeito de traduzir programas Python em *bytecode* Java para JVM (Java Virtual Machine) usando a implementação Jython.

O interpretador faz esta 'tradução' em tempo real para código de máquina, ou seja, em tempo de execução. Já o compilador traduz o programa inteiro em código de máquina de uma só vez e então o executa, criando um arquivo que pode ser rodado (executável). O compilador gera um relatório de erros (casos eles existam) e o interpretador interrompe a tradução quando encontra um primeiro erro.

Em geral, o tempo de execução de um código compilado é menor que um interpretado já que o compilado é inteiramente traduzido antes de sua execução. Enquanto o interpretado é traduzido instrução por instrução. Python é uma linguagem interpretada mas, assim como Java, passa por um processo de compilação. Um código fonte Java é primeiramente compilado para um *bytecode* e depois

interpretado por uma máquina virtual.

Mas devemos compilar script Python? Como compilar? Normalmente, não precisamos fazer nada disso porque o Python está fazendo isso para nós, ou seja, ele faz este passo automaticamente. Na verdade, é o interpretador Pyhton, o CPython. A diferença é que em Java é mais clara essa separação, o programador compila e depois executa o código.

CPython é uma implementação da linguagem Python. Para facilitar o entendimento, imagine que é um pacote que vem com um compilador e um interpretador Python (no caso, uma Máquina Virtual Python) além de outras ferramentas para usar e manter o Python. CPython é a implementação de referência (a que você instala do site http://python.org).

2.4 QUAL VERSÃO UTILIZAR?

Para quem está começando, a primeira dúvida na hora da instalação é qual versão do Python devemos baixar. Aqui, depende do que se deseja fazer. O Python3 ainda possui algumas desvantagens em relação a versão 2 como o suporte de bibliotecas (que é mais reduzido) e pelo fato da maioria das distribuições Linux e o MacOS ainda utilizarem a versão 2 como padrão em seus sistemas. Porém, o Python3 é mais maduro e mais recomendável para o uso.

Existem casos que exigem o Python2 ao invés do Python3 como implementar algo em um ambiente que o programador não controla ou quando precisa utilizar algum pacote/módulo específico que não possui versão compatível com Python3. Vale ressaltar para quem deseja utilizar uma implementação alternativa do Python, como o IronPython ou Jython, que o suporte ao Python3 ainda é bastante limitado.

Atualmente existe a ferramenta ***2to3*** que permite que código Python3 seja gerado a partir de código Python2. Há também a ferramenta ***3to2***, que visa converter o código Python3 de volta ao código Python2. No entanto, é improvável que o código que faz uso intenso de recursos do Python3 seja convertido com sucesso.

PARA SABER MAIS: MÓDULO FUTURE

O módulo future do Python2 contém bibliotecas que fazem uma ponte entre as versões anteriores e as mais recentes. Basta importar e biblioteca **future**:

```
>>> import __future__
```

Para que várias ferramentas disponíveis na versão 3 funcionem na versão 2, ou seja, o módulo `__future__` permite usar funcionalidades do Python3 no Python2. Mas cuidado, algumas funcionalidades são sobrescritas e é importante sempre checar a documentação: https://docs.python.org/3/library/__future__.html

Optamos pelo uso da versão mais recente para este curso, a versão 3.6, e vamos introduzir as diferenças da versão Python2 em comentários durante os capítulos e nos apêndices da apostila.

Editora Casa do Código com livros de uma forma diferente

Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.
Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

Casa do Código, ebook com preço de ebook.

2.5 DOWNLOAD

Como dito acima, o Python já vem instalado nos sistemas Linux e Mac OS mas será necessário fazer o download da última versão (Python 3.6) para acompanhar a apostila. O Python não vem instalado por padrão no Windows e o download deverá ser feito no site https://www.python.org/ além de algumas configurações extras (veja apêndice desta apostila sobre instalação).

2.6 CPYTHON, JYTHON, IRONPYTHON?

Existem outras implementações da linguagem como o Jython e o IronPython. A diferença é que estas implementações são apenas os compiladores. O *bytecode* gerado pelo Jython é interpretado por uma

JVM (Java Virtual Machine) e o *bytecode* do IronPython por uma Virtual Machine .NET.

Outra implementação que vem crescendo é o PyPy, uma implementação escrita em Python que possui uma Virtual Machine Python. É mais veloz do que o CPython e vem com a tecnologia JIT (Just In Time) que já "traduz" o código fonte em código de máquina.

O Compilador Python traduz um *programa.py* para *bytecode* - ele cria um arquivo correspondente chamado *programa.cpy*. Se quisermos ver o *bytecode* pelo terminal, basta usar o módulo *disassembler* (dis) que suporta análise do *bytecode* do CPython, desmontando-o. Você pode checar a documentação aqui: https://docs.python.org/3/library/dis.html.

2.7 PEP - O QUE SÃO E PRA QUE SERVEM

PEP, *Python Enhancement Proposals* ou Propostas para Melhoramento no Python, como o nome diz são propostas de aprimoramento ou de novas funcionalidades para a linguagem. Qualquer um pode escrever uma proposta e a comunidade Python testa, avalia e decide se deve ou não fazer parte da linguagem. Caso aprovado, o recurso é liberado para as próximas versões.

No site oficial do Python (https://www.python.org/) você pode checar todas as PEPs da linguagem. A PEP 0 é aquela que contém o índice de todas as propostas de aprimoramento do Python e pode ser acessada aqui: https://www.python.org/dev/peps.

Ao longo do curso, de acordo com a aprendizagem e uso de certas funcionalidades, citaremos algumas PEPs mais importantes.

Já conhece os cursos online Alura?

A **Alura** oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex-aluno da Caelum tem 15% de desconto neste link!

Conheça os cursos online Alura.

2.8 ONDE USAR E OBJETIVOS

Python é uma linguagem de propósito geral. Muitas vezes precisamos lidar com tarefas laterais:

buscar dados em um banco de dados, ler uma página na internet, exibir graficamente os resultados, criar planilhas etc. E Python possui vários módulos prontos para realizar essas tarefas.

Por esse e outros motivos que Python ganhou grande popularidade na comunidade científica. Além disso, Python é extremamente legível e uma linguagem expressiva, ou seja, de fácil compreensão. As ciências, por outro lado, possuem raciocínio essencialmente complicado e seria um problema adicional para cientistas conhecerem, além de seu assunto de pesquisa, assuntos complexos de um programa de computador como alocação de memória, gerenciamento de recursos etc. Python faz isso automaticamente de maneira eficiente e possibilitando o cientista se concentrar no problema estudado.

2.9 PRIMEIRO PROGRAMA

Vamos para nosso primeiro código! Um programa que imprime uma mensagem simples.

Para mostrar uma mensagem específica, fazemos:

```
print('Minha primeira aplicação Python!')
```

Certo, mas onde digitar esse comando? Como rodar uma instrução Python?

2.10 MODO INTERATIVO

Iremos, primeiro, aprender o **modo interativo** utilizando o terminal (Linux e MacOS) ou o prompt de comando (Windows) para rodar o programa acima. Abra o terminal e digite:

```
dev@caelum:~$ python3.6
```

Isso vai abrir o modo interativo do Python na versão 3.6 da linguagem, também chamado de console do Python. Após digitar este comando, as seguintes linhas irão aparecer no seu console:

```
Python 3.6.4 (default, Jan 28 2018, 00:00:00)
[GCC 4.8.4] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

A primeira linha indica que a versão utilizada do Python é a versão 3.6.4. A segunda indica o sistema operacional (no caso, o Linux). A terceira mostra algumas palavras chaves do interpretador para acessar algumas informações - digite alguma delas e aperte `ENTER` para testar.

O '>>>' indica que entramos no modo interativo do Python e basta começar a escrever os comandos. Vamos então escrever nosso primeiro programa Python:

```
>>> print('Minha primeira aplicação Python!')
```

Ao apertar `ENTER` , temos:

```
>>> print('Minha primeira aplicação Python!')
Minha primeira aplicação Python!
```

O **print()** é uma **função** do Python utilizada para imprimir alguma mensagem na tela. Mais detalhes sobre funções são tratados em um capítulo específico desta apostila. Neste momento, entenda uma função como uma funcionalidade pronta que a linguagem fornece.

Uma mensagem deve estar delimitada entre aspas simples ('') ou duplas (""), como feito no exemplo acima com a mensagem: 'Minha primeira aplicação Python!'. O interpretador, no modo interativo, já vai mostrar a saída deste comando no console, logo abaixo dele.

Mas e se um programa possuir 1.000 linhas de código? Teremos que digitar essas mil linhas todas as vezes para rodar o programa? Isso, obviamente, seria um problema. Existe outro modo de desenvolvimento no Python, mais utilizado, que evita digitar um programa longo no console toda vez que precisar executá-lo.

Você pode também fazer o curso PY-14 dessa apostila na Caelum

Querendo aprender ainda mais sobre? Esclarecer dúvidas dos exercícios? Ouvir explicações detalhadas com um instrutor?
A Caelum oferece o **curso PY-14** presencial nas cidades de São Paulo, Rio de Janeiro e Brasília, além de turmas incompany.

Consulte as vantagens do curso *Python e Orientação a Objetos*

2.11 MODO SCRIPT

O modo interativo é mais utilizado para testes enquanto que o **modo script** é mais comumente utilizado na hora de desenvolver. No modo script isolamos o código Python em um arquivo com extensão **.py**. Dessa maneira, o código é escrito uma única vez e executado pelo interpretador através do comando **python3** (ou o comando **python** se estiver utilizando o Python2).

Abra um editor de texto de sua preferência e escreva o programa anterior nele:

```
print('Minha primeira aplicação Python!')
```

Salve o arquivo como **programa.py**. Para executá-lo, abra o terminal, navegue até o diretório onde se encontra o arquivo **programa.py** e digite:

```
dev@caelum:~$ python3 programa.py
```

Ao apertar `ENTER` , vai aparecer no console:

```
dev@caelum:~$ python3 programa.py
```

```
Minha primeira aplicação Python!
```

Veja que agora isolamos o código em uma arquivo e o executamos através do comando **python3**. Mas não devemos compilar script Python? Como compilar?

Normalmente não precisamos fazer nada disso porque o Python está fazendo isso nos bastidores, ou seja, ele faz este passo automaticamente. Se por algum motivo você queira compilar um programa Python manualmente, você deve usar o módulo **py_compile** no console do Python:

```
>>> import py_compile
>>> py_compile.compile('programa.py')
'__pycache__/programa.cpython-34.pyc'
```

O comando **import** importa o módulo **py_compile** que disponibiliza a função **compile()**. Ou podemos obter o mesmo resultado utilizando o seguinte comando no terminal:

```
dev@caelum:~$ python3 -m py_compile programa.py
```

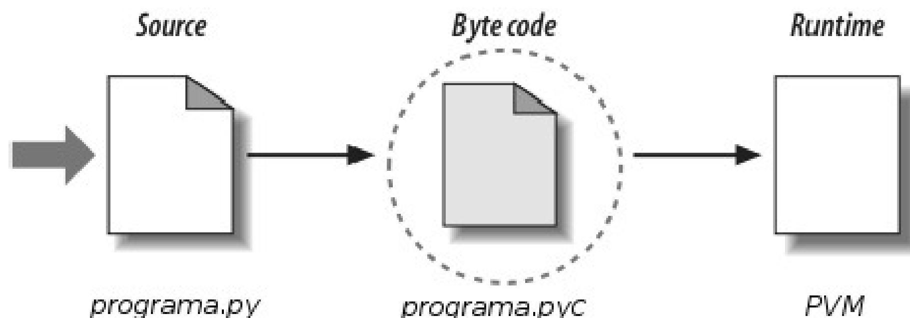
Também é possível compilar todos os arquivos Python de uma única vez usando o módulo **compileall**:

```
dev@caelum:~$ python3 -m compileall
```

Mas nada disso é necessário. O processo de compilação é feito automaticamente e não é preciso repetir todo este processo para rodar um programa Python.

Apenas rodando o programa, sem precisar compilá-lo, note que uma nova pasta chamada "__pycache__" é criada (caso ela não exista) no mesmo diretório que o programa foi executado. Dentro desta pasta é criado um arquivo **programa.cpython-34.pyc** - esta é a versão compilada do programa, o código *bytecode* gerado pelo CPython.

Sempre que um programa Python é chamado, o Python checará se existe uma versão compilada com a extensão **.pyc** - este arquivo deve ser mais novo do que o de extensão **.py** (se o arquivo existir). O Python vai carregar o *bytecode*, o que vai acelerar o script. Senão existir a versão *bytecode*, o Python criará o arquivo *bytecode* antes de iniciar a execução do programa. Execução de um programa Python significa a execução de um código *bytecode* na Python Virtual Machine.



Toda vez que um script Python é executado, um código *bytecode* é criado. Se um script Python é importado como um módulo, o *bytecode* vai armazenar seu arquivo **.pyc** correspondente.

Portanto, o passo seguinte não criará o arquivo *bytecode* já que o interpretador vai verificar que não existe nenhuma alteração:

```
dev@caelum:~$ python programa.py
Minha primeira aplicação Python!
dev@caelum:~$
```

2.12 EXERCÍCIO: MODIFICANDO O PROGRAMA

1. Altere o programa para imprimir uma mensagem diferente.

2. Altere seu programa para imprimir duas linhas de código utilizando a função `print()` .

3. Sabendo que os caracteres `\n` representam uma quebra de linha, imprima duas linhas de texto usando uma única linha de código.

2.13 O QUE PODE DAR ERRADO?

Nem sempre as coisas acontecem como esperado. O Python tem uma sintaxe própria, um vocabulário próprio. Digitar algo que o interpretador não entende causará um erro no programa. Vejamos alguns exemplos:

Esquecer os parênteses

```
>>> print Minha primeira aplicação Python!
Traceback (most recent call last):
File "<stdin>", line 1
    print 'Minha primeira aplicação Python!'
                        ^
SyntaxError: Missing parentheses in call to 'print'. Did you mean print('Minha primeira aplicação
Python!')?
```

Não se assuste com a mensagem. Vamos entender o que ela quer dizer. Na primeira linha aparece a palavra `Traceback` que significa algo como: "O que o programa estava fazendo quando parou porque algo de errado aconteceu?". É por este motivo que a mensagem `most recent call last` (chamada mais recente) é mostrada.

A `Traceback` faz referência a um arquivo - que é o nome do arquivo Python chamado acima pelo nome de `stdin` que possui métodos para leitura, onde o programa lê a entrada do teclado. O programa acusa que este erro está na primeira linha do programa: `File "<stdin>", line 1` .

Logo em seguida é mostrado exatamente a parte do código que gerou o erro: `print 'Minha primeira aplicação Python!'` . A próxima linha é a mensagem de erro: `SyntaxError` . Se você não faz a menor ideia do que esta mensagem significa é um bom começo e uma boa prática durante a

aprendizagem pesquisar a respeito dela na internet, assim como demais erros que possam surgir.

Neste caso, é um `SyntaxError` , ou seja, `Erro de Sintaxe` - o Python não entendeu o que foi digitado. A mensagem diz que faltam os parênteses! Então, é fácil achar um erro quando ele acontece.

Algumas vezes você verá a palavra `Exception` em uma mensagem de erro. Uma `Exception` é um problema que ocorre enquanto o código está sendo executado. Já o `SyntaxError` é um problema detectado quando o Python verifica o código antes de executá-lo, ou seja, em tempo de compilação.

Esquecer de fechar os parênteses

O interpretador vai aguardar (continuar imprimindo reticências cada vez que a tecla `ENTER` for apertada) até que o parêntese seja fechado:

```
>>> print('Minha primeira aplicação Python!'
...
...
...
```

Neste caso não é uma exceção ou erro, a não ser que você digite qualquer outra coisa que não um fechamento de parêntese e aperte a tecla `ENTER` .

Esquecer de colocar a mensagem entre aspas (simples ou duplas)

```
>>>    print(Minha primeira aplicação Python!)
  File "<stdin>", line 1
print(Minha primeira aplicação Python!)
                 ^
SyntaxError: invalid syntax
```

Mais uma vez acusa erro de sintaxe.

Estes foram alguns erros que o programa pode gerar por desatenção do programador. São mais comuns de acontecer do que se imagina. Outros erros serão abordados em um capítulo específico desta apostila. Neste momento iremos aprender outros recursos que a linguagem Python oferece e se familiarizar com sua sintaxe.

Seus livros de tecnologia parecem do século passado?

Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.
Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

Casa do Código, Livros de Tecnologia.

CAPÍTULO 3

VARIÁVEIS E TIPOS EMBUTIDOS

Neste capítulo vamos conhecer os tipos da biblioteca padrão do Python. Os principais tipos internos são números, sequências, mapas, classes, objetos e exceções, mas iremos focar primeiramente nos números e sequências de texto (*strings*). São objetos nativos da linguagem, recursos que já vêm prontos para uso e chamados de *built-ins*.

Neste início da aprendizagem trabalharemos com o modo interativo e ao final produziremos uma pequena aplicação em um script.

3.1 TIPOS EMBUTIDOS (BUILT-INS)

Um valor, como um número ou texto, é algo comum em um programa. Por exemplo, '*Hello, World!*', 1, 2, todos são valores. Estes valores são de diferentes tipos: 1 e 2 são números inteiros e '*Hello World!*' é um texto, também chamado de ***String***. Podemos identificar **strings** porque são delimitadas por aspas (simples ou duplas) - e é exatamente dessa maneira que o interpretador Python também identifica uma **string**.

A função `print()` utilizada no capítulo anterior também trabalha com inteiros:

```
>>> print(2)
2
```

Veja que aqui não é necessário utilizar aspas por se tratar de um **número**. Caso você não tenha certeza qual é o tipo de um valor, pode usar a função `type()` para checar:

```
>>> type('Hello World')
<class 'str'>

>>> type(2)
<class 'int'>
```

Strings são do tipo `str` (abreviação para *string*) e inteiros do tipo `int` (abreviação para *integer*). Ignore a palavra `class` por enquanto, teremos um capítulo especial para tratar dela. Veremos que funções como `type()` e `print()` também são tipos embutidos no Python.

Outro tipo que existe no Python são os números decimais que são do tipo `float` (ponto flutuante):

```
>>> type(3.2)
<class 'float'>
```

E qual será o tipo de valores como '2' e '3.2'? Eles se parecem com números mas são delimitados por aspas como *strings*. Utilize a função `type()` para fazer a verificação:

```
>>> type('2')
<class 'str'>

>>> type('3.2')
<class 'str'>
```

Como estão delimitados por aspas, o interpretador vai entender esses valores como *strings*, ou seja, como texto.

O Python também possui um tipo específico para números complexos. Números complexos são definidos por dois valores: a parte real e a parte imaginária. No Python é escrito na forma **real + imag j**. No caso, o número imaginário (definido pela raiz de -1 e chamado de 'i' na matemática) é designado pela letra `j` no Python. Por exemplo:

```
>>> 2 + 3j
>>> type(2 + 3j)
<class 'complex'>
```

2 é a parte real e 3 a parte imaginária do número complexo. Utilizando a função `type()` podemos nos certificar que seu tipo é `complex` .

Editora Casa do Código com livros de uma forma diferente

Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.
Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.]

3.2 VARIÁVEIS

Podemos pedir para o Python lembrar de um valor que queiramos utilizar em outro momento do programa. O Python vai guardar este valor em uma **variável**. Variável é um nome que faz referência a um valor. É como uma etiqueta que colocamos naquele valor e quando precisarmos usar, chamamos pelo nome que foi dado na etiqueta.

Um comando de atribuição (o sinal de igualdade `=`) cria uma nova variável e atribui um valor a ela:

```
>>> mensagem = 'oi, python'
'oi, python'

>>> numero = 5
5

>>> pi = 3.14
3.14
```

Três atribuições foram feitas neste código. Atribuímos a variável `mensagem` uma *string*; a variável `numero` um inteiro e a variável `pi` um valor aproximado do número pi. No modo interativo, o interpretador mostra o resultado após cada atribuição.

Para recuperar esses valores, basta chamar pelos nomes das variáveis definidas anteriormente:

```
>>> mensagem
oi, python

>>> numero
5

>>> pi
3.14
```

Utilize a função `type()` para verificar seus tipos:

```
>>> type(mensagem)
<class 'str'>

>>> type(numero)
<class 'int'>

>>> type(pi)
<class 'float'>
```

3.3 PARA SABER MAIS: NOMES DE VARIÁVEIS

Programadores escolhem nomes para variáveis que sejam semânticos e que ao mesmo tempo documentem o código. Esses nomes podem ser bem longos, podem conter letras e números. É uma convenção entre os programadores Python começar a variável com letras minúsculas e utilizar o underscore (_) para separar palavras como: **meu_nome**, **numero_de_cadastro**, **telefone_residencial**. Esse padrão é chamado de ***snake case***. Variáveis também podem começar com underscore (_) mas deve ser evitado e utilizado em casos mais específicos.

Se nomearmos nossas variáveis com um nome ilegal, o interpretador vai acusar um erro de sintaxe:

```
>>> 1nome = 'python'
  File "<stdin>", line 1
  1nome = 'python'
    ^
SyntaxError: invalid syntax

>>> numero@ = 10
  File "<stdin>", line 1
  numero@ = 10
```

```
      ^
SyntaxError: invalid syntax

>>> class = 'oi'
  File "<stdin>", line 1
  class = oi
    ^
SyntaxError: invalid syntax
```

`1nome` é ilegal porque começa com um número, `numero@` é ilegal porque contém um caractere especial (o @) considerado ilegal para variáveis. E `class` é ilegal porque `class` é uma **palavra chave** em Python. O interpretador utiliza palavras chaves como **palavras reservadas** da linguagem, como um vocabulário próprio.

Python possui 33 palavras reservadas:

```
and        del        from        None        True
as        elif        global        nonlocal        try
assert        else        if        not        while
break        except        import        or        with
class        False        in        pass        yield
continue        finaly        is        raise        def
for        lambda        return
```

Portanto, não podemos utilizar essas palavras para nomear nossas variáveis.

3.4 INSTRUÇÕES

Uma instrução (ou comando) é uma unidade de código que o Python pode executar. Por exemplo, a função `print()` para imprimir uma mensagem na tela é um comando:

```
>>> print("Hello, World!")
Hello, World!
```

Quando executamos um comando no modo interativo, o interpretador Python apresenta o resultado, caso exista, deste comando. Um script contém uma sequência de instruções. Se existir mais de um comando, os resultados vão aparecendo durante a execução do programa:

```
print(1)
x = 2
print(x)
```

E produz a saída:

```
1
2
```

Para rodar um script em Python é preciso concentrar esses comandos em um mesmo lugar e pedir para o interpretador executá-los. Criamos um arquivo de extensão **.py** com estes comandos, como aprendemos no capítulo anterior.

Arquivo **programa.py**:

Note que devemos utilizar a função `print()` para exibir os resultados na tela já que o modo script, diferente do modo interativo, não exibe os resultados após a declaração de variáveis.

Navegue até o diretório onde se encontra o arquivo **programa.py** e digite o comando no terminal:

```
dev@caelum:~$ python3 programa.py
```

Que vai gerar a saída:

```
dev@caelum:~$ python3 programa.py
oi, python
5
3.14
```

Já conhece os cursos online Alura?

A **Alura** oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex-aluno da Caelum tem 15% de desconto neste link!

Conheça os cursos online Alura.

3.5 OPERADORES ARITMÉTICOS

Operadores são símbolos especiais que representam cálculos como adições e multiplicações. Para fazer cálculos com números utilizamos os operadores +, -, *, / e ** que representam, respectivamente, adição, subtração, multiplicação, divisão e potenciação.

Uma expressão é uma combinação de valores, variáveis e operadores como `x + 17`, `1 + 1` etc. Quando digitamos uma expressão no modo interativo, o interpretador vai calcular e imprimir o resultado:

```
>>> 1 + 1
```

```
2

>>> 2 * 3
6
```

Também podemos usar variáveis:

```
>>> x = 1
>>> y = 3
>>> x + y
4

>>> x - y
-2

>>> x * y
3

>>> x / y
0.3333333333333333

>>> x ** y
1
```

Além dos operadores comentados, temos também o operador // que representa a divisão inteira:

```
>>> 7 // 2
3
```

E o operador módulo % que resulta no resto da divisão entre dois números inteiros:

```
>>> 7 % 3
1
```

7 divido por 3 é 2 e gera resto igual a 1. Esse operador é bem útil quando queremos checar se um número é divisível por outro.

Os principais operadores são:

Operação	Nome	Descrição
a + b	adição	Soma entre *a* e *b*
a - b	subtração	Diferença entre *a* e *b*
a * b	multiplicação	Produto entre *a* e *b*
a / b	divisão	Divisão entre *a* e *b*
a // b	divisão inteira	Divisão inteira entre *a* e *b*
a % b	módulo	Resto da divisão entre *a* e *b*
a ** b	exponenciação	*a* elevado a potência de *b*

3.6 STRINGS

O operador `+` também funciona com *strings* de uma maneira diferente dos números. Ele funciona

concatenando *strings*, ou seja, juntando duas *strings*:

```
>>> texto1 = 'oi'
>>> texto2 = 'Python'
texto1 + texto2
oiPython
```

O operador `*` também funciona com *strings*, multiplicando seu conteúdo por um inteiro. Vamos checar esse resultado:

```
>>> texto1 = 'python'
>>> texto1 * 3
python python python
```

Ao multiplicar por 3 o Python replica a *string* três vezes. *Strings* possuem muitas funcionalidades prontas chamadas de métodos. O método `upper()` , por exemplo, retorna o texto em letras maiúsculas. Já o método `capitalize()` retorna o texto capitalizado (com a primeira letra em maiúscula):

```
>>> texto1.upper()
'PYTHON'

>>> texto1.capitalize()
'Python'
```

Outras funcionalidades de *strings* estão presentes na documentação que pode ser acessada neste link: https://docs.python.org/3/library/stdtypes.html#string-methods

3.7 ENTRADA DO USUÁRIO

Agora vamos criar mais interatividade e pedir para o usuário entrar com um valor digitado do teclado.

O Python possui uma função que captura a entrada de valores: a função `input()` . Quando essa função é chamada, o programa para e espera o usuário digitar alguma coisa. Quando o usuário aperta a tecla `ENTER` , o programa processa e imprime o valor digitado em forma de *string*:

```
>>> entrada = input()
'oi pyhton'

>>> print(entrada)
'oi python'
```

Mas o ideal é pedir algo específico ao usuário e dizer qual dado queremos receber. Podemos passar uma *string* para a função `input()` :

```
>>> nome = input("digite seu nome:\n")
digite seu nome:
caelum

>>> print(nome)
caelum
```

O `\n` no final representa uma nova linha e o interpretador vai quebrar uma linha após imprimir a

string. Por este motivo, o valor digitado pelo usuário aparece na próxima linha.

Com o conteúdo aprendido até aqui já é possível começar a escrever o primeiro script. Crie um arquivo **programa2.py** e acrescente um código que vai pedir que o usuário entre com algum valor e, em seguida, o programa deve imprimir este valor.

Arquivo **programa2.py**:

```
numero = input('Digite um número:\n')
print(numero)
```

Podemos melhorar e imprimir uma mensagem como *O número digitado foi* :

```
numero = input('Digite um número:\n')
print('O número digitado foi ' + numero)
```

Concatenamos a *string* com a variável `numero` utilizando o operador `+` . Agora, se o usuário digitar o número 2, a saída será *O número digitado foi 2*. Outra maneira mais elegante é usar a função `format()` :

```
print('O número digitado foi {}'.format(numero))
```

A função `format()` vai substituir o `{}` pela variável `numero` . A princípio, pode parecer uma alternativa pior já que escrevemos mais código para conseguir o mesmo resultado. Mas a função `format()` fornece mais facilidades. Suponha que o programa receba dois valores digitados pelo usuário e os imprima em uma única mensagem:

```
nome = input('Digite seu nome ' + nome)
idade = input('Digite sua idade ' + idade)
print('Seu nome é {} e sua idade é {}'.format(nome, idade))
```

Veja que essa forma facilita a impressão e formatação dos dados uma vez que não quebra a *string* em várias partes como a concatenação faz. Além do que, com o operador `+` , sempre temos que lembrar dos espaço em branco entre as palavras:

```
print('Seu nome é ' + nome + ' e sua idade é ' + idade)
```

Neste caso a função `format()` é mais recomendada e facilita na impressão de mensagens na tela. Agora o script está melhor e podemos executá-lo pelo terminal:

```
dev@caelum:~$ python3 programa2.py
```

A saída:

```
digite seu nome:
caelum
digite sua idade:
20
Seu nome é caelum e sua idade é 20
```

PARA SABER MAIS: A FUNÇÃO FORMAT()

A função `format()` faz parte de um conjunto de funções de formatação de *strings* chamada **Formatter**. Para mais detalhes acesse a documentação: https://docs.python.org/3/library/string.html#string.Formatter.

Há outras funções de formatação e a `format()` é a principal delas e a mais utilizada. Podemos passar qualquer tipo de parâmetro e ela é especialmente útil para formatar números passando seu *format code*. Por exemplo, podemos arredondar o número flutuante 245.2346 para duas casas decimais através do código de formatação `:.2f`:

```
>>> x = 245.2346
>>> print('{:.2f}'.format(x))
245.23
```

O `:.2f` diz que queremos apenas duas casas decimais para a variável `x`. Na documentação oficial do Python você acessa os códigos de formatação ou através da PEP 3101: https://www.python.org/dev/peps/pep-3101/.

Você pode também fazer o curso PY-14 dessa apostila na Caelum

Querendo aprender ainda mais sobre? Esclarecer dúvidas dos exercícios? Ouvir explicações detalhadas com um instrutor?
A Caelum oferece o **curso PY-14** presencial nas cidades de São Paulo, Rio de Janeiro e Brasília, além de turmas incompany.

Consulte as vantagens do curso *Python e Orientação a Objetos*

3.8 CONSTANTES

O Python possui poucas constantes embutidas. As mais utilizadas são **True**, **False** e **None**. Essas também são palavras chaves do Python, portanto palavras reservadas que não podemos utilizar como nomes de variáveis.

`True` e `False` são valores **booleanos** que representam, respectivamente, **verdadeiro** e **falso**. O Python também possui a função `bool()` que retorna `True` quando o argumento passado é verdadeiro e retorna `False`, caso contrário.

Podemos representar `True` e `False` através de expressões. Por exemplo "O número 1 é igual a *string* '1'?". Vamos perguntar ao Python:

```
>>> 1 == '1'
False
```

```
print(1 == '1')
```

O operador `==` é usado para verificar se algo é igual a outro. Não confundir com o `=` que atribui um valor a uma variável. Também podemos verificar se um número é maior, utilizando o operador `>`, ou menor (`<`) do que outro:

```
>>> 2 > 1
True
```

```
>>> 2 < 1
False
```

Podemos também utilizar a função `bool()` para fazer a verificação:

```
>>> bool(3 > 5)
False
```

```
>>> bool(1 == 1)
True
```

O comando `bool()` não recebe apenas expressões, ele pode receber qualquer coisa e vai responder se tal valor é considerado `True` ou `False` :

```
>>> bool(0)
False
```

```
>>> bool('')
False
```

```
>>> bool(None)
False
```

```
>>> bool(1)
True
```

```
>>> bool(-100)
True
```

```
>>> bool(13.5)
True
```

```
>>> bool('teste')
True
```

```
>>> bool(True)
True
```

Repare que a função resulta `False` em *strings* vazias, quando um número é zero ou quando é `None` . Ainda não falamos o que o `None` representa. É um valor do tipo `NoneType` e é usado para representar a abstenção de um valor - como quando um argumento padrão não é passado para uma função (que veremos em outro capítulo).

```
type(None)
<class 'NoneType'>
```

Em outras linguagens de programação é comum utilizar a palavra **Null** para representar a abstenção de valor. Para programadores mais experientes e com algum conhecimento em linguagens como Java e C#, é importante observar que diferente do **Null**, o **None** ocupa espaço na memória, é um objeto com referência.

No exemplo acima foram utilizados três operadores diferentes daqueles já vistos anteriormente: o `==` (igual), o `>` (maior do que) e o `<` (menor do que). Estes operadores não são aritméticos, são conhecidos por **operadores de comparação**. O Python possui mais operadores deste tipo:

Operação	Descrição
a == b	*a* igual a *b*
a != b	*a* diferente de *b*
a < b	*a* menor do que *b*
a > b	*a* maior do que *b*
a <= b	*a* menor ou igual a *b*
a >= b	*a* maior ou igual a *b*

Outros operadores que retornam valores **booleanos** são:

Operação	Descrição
a is b	True se *a* e *b* são idênticos
a is not b	True se *a* e *b* não são idênticos
a in b	True se *a* é membro de *b*
a not in b	True se *a* não é membro de *b*

É importante saber que os operadores `==` e `is` funcionam de maneira diferente. Vamos usar o exemplo de duas listas e checar se elas são iguais:

```
>>> x = [1, 2, 3]
>>> y = [1, 2, 3]
>>> x == y
True

>>> x is y
False
```

O operador `==` checa se o conteúdo das variáveis são iguais e seu comportamento pode variar de interpretador para interpretador - o exemplo acima é o comportamento padrão do CPython. Já o operador `is` checa se `a` e `b` são o mesmo **objeto**. Falaremos de objetos em um outro capítulo, mas é importante ter em mente que tudo em Python é um objeto e cada objeto possui uma referência na memória. O operador `is` vai checar exatamente se `x` e `y` são o mesmo objeto, ou seja, se possuem a

mesma referência.

3.9 COMANDO IF

E se quisermos apresentar uma mensagem diferente para o usuário dependendo do valor de entrada? Vamos atribuir um valor para uma variável `numero` e pedir para o usuário entrar com um valor. Devemos verificar se os valores são iguais como um jogo de adivinhação em que o usuário deve adivinhar o número definido.

```
numero = 42
chute = input('Digite um número: ')
```

Até aqui, nenhuma novidade. Agora devemos mostrar a mensagem "Você acertou" caso o `numero` seja igual ao `chute`, e "Você errou" caso o `numero` seja diferente do `chute`. Em português, seria assim:

```
Se chute igual a número: "Você acertou"
Se chute diferente de número: "Você errou"
```

Ou melhor:

```
Se chute igual a número: "Você acertou"
Senão: "Você errou"
```

Podemos traduzir isso para código Python. O Python possui o operador condicional para representar a palavra **se** que é o **if** e a palavra **senão** que é o **else**. A sintaxe ficaria:

```
if chute == numero:
    print('Você acertou')
else:
    print('Você errou')
```

Este código ainda não funciona porque o Python entende as instruções `if` e `else` como blocos e os blocos devem seguir uma **indentação**. Como `print('Você acertou')` é a instrução que deve ser executada caso a verificação do `if` seja verdadeira, devemos ter um recuo para a direita em quatros espaços:

```
if chute == numero:
   print('Você acertou')
else:
   print('Você errou')
```

Caso contrário, o interpretador vai acusar erro de sintaxe. Dessa maneira o código fica mais legível e o que em outras linguagens é uma escolha do programador, o Python te obriga a fazer - forçando, desta maneira, a organizar o código. Tudo que estiver no bloco da primeira condição (do `if`) deve estar indentado, ou seja, recuado para direita. Assim como as instruções que estiverem no bloco do `else`.

No fim, nosso programa, que salvaremos em um arquivo chamado **adivinhacao.py** fica:

```
numero = 42
chute = input('Digite um número: ')
```

```
if chute == numero:
   print('Você acertou')
else:
   print('Você errou')
```

E executamos no terminal:

```
dev@caelum:~$ python3 adivinhacao.py
Digite um número:
25
Você errou
```

Note que a condição de um **if** deve ser um booleano, ou seja, **True** ou **False**. Passamos a expressão `chute == numero` que vai checar se ela é verdadeira ou não. Caso seja verdadeira, vai executar o código dentro do bloco do **if**, senão, vai executar o código dentro do **else**. Agora vamos chutar o número 42 e verificar se tudo está funcionando:

```
dev@caelum:~$ python3 adivinhacao.py
Digite um número:
42
Você errou
```

Algo de errado aconteceu! Digitamos o número correto e mesmo assim o programa não funcionou como esperado. Vamos entender o que aconteceu.

3.10 CONVERTENDO UMA STRING PARA INTEIRO

A função `input()` lê o valor digitado pelo usuário como uma *string*.

```
chute = input('Digite um número: ')
```

Se o usuário digitar o número 42, a variável `chute` vai guardar o valor `"42"` , ou seja, um texto. Podemos checar isso através da função `type()` que retorna o tipo da variável. Vamos testar isso no terminal:

```
>>> chute = input('Digite um número: ')
Digite um número: 42
>>> type(chute)
<class 'str'>
```

Agora fica mais claro porque o programa não está funcionando como o esperado. Quando o interpretador verificar `chute == numero` vai retornar **False** já que `"42"` (texto) é diferente de 42 (número).

Para funcionar, precisamos **converter** a *string* "42" para um número inteiro. O **int** também funciona como uma função (mais para frente entenderemos que não é realmente uma função) que pode receber uma *string* e retornar o inteiro correspondente:

```
>>> numero_em_texto = '12'
'12'
>>> type(numero_em_texto)
<class 'str'>
```

```
>>> numero = int(numero_em_texto)
12
>>> type(numero)
<class 'int'>
```

Mas devemos tomar cuidado, nem toda *string* pode ser convertida para um número inteiro:

```
>>> texto = 'caelum'
>>> numero = int(texto)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'caelum'
```

O interpretador acusa um **ValueError** dizendo que o valor passado para `int()` é inválido, ou seja, é um texto que não representa um número inteiro.

Seus livros de tecnologia parecem do século passado?

Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.
Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

Casa do Código, Livros de Tecnologia.

3.11 O COMANDO ELIF

Podemos melhorar ainda mais o jogo: caso o chute não seja igual ao número secreto, podemos dar uma pista para o usuário se ele foi maior ou menor do que o chute inicial. Ou seja, devemos acrescentar esse tratamento caso o usuário erre o chute:

```
Se chute = número:
    "Você acertou!"
Senão:
    Se chute maior do que número_secreto:
        "Você errou! O seu chute foi maior que o número secreto"
    Senão
        "Você errou! O seu chute foi menor que o número secreto"
```

Já sabemos traduzir isso para Python:

```
if(chute == numero_secreto):
print('Você acertou!')
else:
    if(chute > numero_secreto):
        print('Você errou! O seu chute foi maior que o número secreto')
```

```
    else:
        print('Você errou! O seu chute foi menor que o número secreto')
```

Mas neste caso podemos fazer um `else` com uma condição de entrada, o `elif`. Vamos utilizá-lo para deixar o código mais semântico, já que na prática não há diferença:

```
if (numero_secreto == chute):
    print('Você acertou!')
elif (chute > numero_secreto):
    print('Você errou! O seu chute foi maior que o número secreto')
elif (chute < numero_secreto):
    print('Você errou! O seu chute foi menor que o número secreto')
```

Podemos melhorar ainda mais a legibilidade do código para que os outros programadores, que podem ajudar a desenvolvê-lo no futuro, entendam melhor. Vamos deixar nossas condições mais claras. `chute == numero_secreto` quer dizer que o usuário acertou. Então, extraímos essa condição para uma variável:

```
acertou = chute == numero_secreto

if(acertou):
    print('Você acertou!')

#restante do código
```

A variável `acertou` guarda uma expressão e, portanto é do tipo **booleano** e podemos usar como condição no comando `if`. Agora a condição `if` fica um pouco mais clara. Vamos fazer a mesma coisa para as outras duas condições:

```
  acertou = chute == numero_secreto
    maior = chute > numero_secreto
    menor = chute < numero_secreto

    if(acertou):
        print('Você acertou!')
    elif(maior):
        print('Você errou! O seu chute foi maior que o número secreto')
    elif(menor):
        print('Você errou! O seu chute foi menor que o número secreto')
```

3.12 EXERCÍCIOS - JOGO DA ADIVINHAÇÃO

1. Crie um arquivo chamado *adivinhacao.py* em uma pasta chamada *jogos* dentro do diretório home:

```
|_ home
    |_ jogos
        |_ adivinhacao.py
```

2. Abra o arquivo no editor de texto de sua preferência e comece a escrever um cabeçalho para o usuário saber do que se trata o programa:

```
print('*********************************')
print('*    Jogo da adivinhação     *')
print('*********************************')
```

3. Vamos definir a variável `numero_secreto` que vai guardar o valor a ser adivinhado pelo usuário:

```
print('*********************************')
print('*     Jogo da adivinhação     *')
print('*********************************')

numero_secreto = 42
```

4. Capture a entrada do usuário usando a função `input()` :

```
print('*********************************')
print('*     Jogo da adivinhação     *')
print('*********************************')

numero_secreto = 42

chute = input('Digite o seu número: ')
print('Você digitou: ', chute)
```

5. Compare o valor digitado pelo usuário com o `numero_secreto` . Se os valores forem iguais mostre uma mensagem de acerto, caso contrário, mostre uma mensagem de erro:

```
print('*********************************')
print('*     Jogo da adivinhação     *')
print('*********************************')

numero_secreto = 42

chute = input('Digite o seu número: ')
print('Você digitou: ', chute)

if(numero_secreto == chute):
    print('Você acertou!')
else:
    print('Você errou!')
```

6. Rode o código acima pelo terminal e teste o jogo chutando o número 42:

```
dev@caelum:~$ python3 jogos/adivinhacao.py
```

7. O chute 42 não funciona como esperado. Esquecemos de converter o chute digitado pelo usuário para um número inteiro. Modifique o código e utilize a função `int()` para receber a entrada do usuário:

```
chute = int(input('Digite o seu número: '))
```

8. Rode o código novamente com a entrada igual a 42 e veja que agora funciona como esperado.

9. Vamos apresentar uma pista para o usuário e imprimir uma mensagem dizendo se o chute foi maior ou menor do que o número secreto. Para isso usaremos o `elif` :

```
if (numero_secreto == chute):
    print('Você acertou!')
elif (chute > numero_secreto):
    print('Você errou! O seu chute foi maior que o número secreto')
elif (chute < numero_secreto):
    print('Você errou! O seu chute foi menor que o número secreto')
```

10. Agora vamos melhorar a legibilidade do código extraindo as condições para variáveis:

```
acertou = chute == numero_secreto
maior = chute > numero_secreto
menor = chute < numero_secreto

if(acertou):
    print('Você acertou!')
elif(maior):
    print('Você errou! O seu chute foi maior que o número secreto')
elif(menor):
    print('Você errou! O seu chute foi menor que o número secreto')
```

11. Rode o programa e teste com todas as situações possíveis.

3.13 COMANDO WHILE

Queremos dar mais de uma oportunidade para o usuário tentar acertar o número secreto, já que é um jogo de adivinhação. A primeira ideia é repetir o código, desde a função `input()` até o bloco do `elif` . Ou seja, para cada nova tentativa que quisermos dar ao usuário, copiaríamos esse código novamente.

Só que copiar código sempre é uma má prática, queremos escrever o código apenas uma vez. Se queremos repetir o código, fazemos um laço, ou um loop, que deve repetir a instrução dentro de bloco **enquanto** ela for verdadeira. O laço que devemos fazer é:

```
Enquanto ainda há tentativas, faça:
    chute_str = input('Digite o seu número: ')
    print('Você digitou: ', chute_str)
    chute = int(chute_str)

    acertou = numero_secreto == chute
    maior = chute > numero_secreto
    menor = chute < numero_secreto

    if (acertou):
        print('Você acertou!')
    elif (maior):
        print('Você errou! O seu chute foi maior que o número secreto')
    elif (menor):
        print('Você errou! O seu chute foi menor que o número secreto')

    print('Fim do Jogo!')
```

Como dito anteriormente, o Python não entende português e assim como o `if` ele tem um comando que substituirá a palavra `enquanto` do nosso exemplo. O **while** é esse comando que, assim como o `if` , recebe uma condição. A diferença é que o `if` , caso a condição seja verdadeira, executa apenas uma vez o código de seu bloco, já o `while` executa **enquanto** a condição for verdadeira, por exemplo:

```
x = 5
enquanto x for maior do que 1, faça:
    imprime(x)
    x = x - 1
```

Que em Python, é equivalente a:

```
>>> x = 5
>>> while(x > 1):
...     print(x)
...     x = x - 1
5
4
3
2
```

Mas tome cuidado, o que acontece se esquecermos essa linha do código `x = x - 1` ?

```
>>> x = 5
>>> while(x > 1):
...     print(x)
5
5
5
5
5
...
```

O programa vai imprimir o número 5 infinitamente, já que a condição passada é sempre verdadeira e não muda dentro do bloco.

Agora é a melhor hora de aprender algo novo

alura

Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

Conheça a Alura Cursos Online.

3.14 EXERCÍCIOS - JOGO COM WHILE

1. Daremos ao usuário do jogo um número máximo de tentativas. Abra o arquivo *adivinhacao.py* na pasta **jogos** e inicie a variável `total_de_tentativas` com 3 e acrescente o bloco do comando `while` :

```
numero_secreto = 42
total_de_tentativas = 3

while(ainda há total_de_tentativas):
    #executa o código
```

2. Resta agora a expressão **ainda há**. A ideia é que o usuário tenha 3 tentativas, representada no código pela variável `total_de_tentativas` . A cada rodada subtraímos 1 do valor dessa variável, até o valor chegar a 0, que é quando devemos sair do `while` . Logo, vamos executá-lo enquanto a variável `total_de_tentativas` for maior do que 0:

```
numero_secreto = 42
total_de_tentativas = 3

while (total_de_tentativas > 0):
    chute = int(input('Digite o seu número: '))
    print('Você digitou: ', chute)

    acertou = chute == numero_secreto
    maior = chute > numero_secreto
    menor = chute < numero_secreto

    if(acertou):
        print('Você acertou')
    elif(maior):
        print('Você errou! O seu chute foi maior que o número secreto')
    elif(menor):
        print('Você errou! O seu chute foi menor que o número secreto')

    total_de_tentativas = total_de_tentativas - 1
```

OBS: Não esqueça de indentar o código dentro do bloco `while` para não receber erro de sintaxe.

3. Além das tentativas, podemos apresentar qual o número da **rodada** que o usuário está jogando para deixar claro quantas tentativas ele têm. Para isso vamos criar a variável `rodada` , que começa com o valor 1:

```
total_de_tentativas = 3
rodada = 1
```

4. E vamos imprimi-la antes do usuário digitar o seu chute:

```
total_de_tentativas = 3
rodada = 1

while (total_de_tentativas > 0):
    print('Tentativa {} de {}'.format(rodada, total_de_tentativas))
    chute = int(input('Digite o seu número: '))
    print('Você digitou: ', chute)

    # restante do código aqui
```

5. E para a variável `total_de_tentativas` continuar com o valor 3, não vamos mais subtrair 1 do seu valor, e sim adicionar 1 ao valor da variável `rodada` :

```
total_de_tentativas = 3
rodada = 1

while (total_de_tentativas > 0):
    print('Tentativa {} de {}'.format(rodada, total_de_tentativas))

    chute = int(input('Digite o seu número: '))
      print('Você digitou: ', chute)
```

```
        acertou = numero_secreto == chute
        maior = chute > numero_secreto
        menor = chute < numero_secreto

        if (acertou):
            print('Você acertou!')
        elif (maior):
          print('Você errou! O seu chute foi maior que o número secreto')
      elif (menor):
          print('Você errou! O seu chute foi menor que o número secreto')

        rodada = rodada + 1

    print('Fim do jogo')
```

6. Por fim, precisamos modificar a condição. Já que o `total_de_tentativas` permanecerá com o valor 3, o código precisa ficar executando enquanto o valor da rodada for menor ou igual ao total de tentativas:

```
total_de_tentativas = 3
rodada = 1

while (rodada <= total_de_tentativas):
    print('Tentativa {} de {}'.format(rodada, total_de_tentativas))
    chute_str = input('Digite o seu número: ')

    #restante do código
```

   Agora conseguimos imprimir para o usuário quantas tentativas restantes ele possui! Teste chamando seu arquivo **adivinhacao.py** com o comando 'python3'.

7. Falta arrumar uma coisa: quando o usuário acerta, o jogo continua pedindo um novo chute. Queremos terminar a execução do programa quando o usuário acerta. Para isso usamos o comando **break**. Quando o interpretador encontrar o comando **break** ele para a execução do programa. vamos acrescentar isso quando o usuário acertar, ou seja, no primeiro comando `if` após a exibição da mensagem de acerto:

```
if(acertou):
    print('Você acertou!')
    break
elif(maior):
    # restante do código
```

8. Teste o programa e veja se tudo está funcionando como o esperado.

3.15 COMANDO FOR

Ainda no código do jogo da adivinhação, implementamos o *loop* `while`, no qual temos uma variável `rodada` que começa com o valor 1, e é incrementada dentro do *loop*, que por sua vez tem uma condição de entrada, que é a `rodada` ser menor ou igual ao total de tentativas, que é 3.

Ou seja, a `rodada` tem um valor inicial, que é 1, e vai até 3. Fazemos um laço começando com um

valor inicial, até um valor final, sempre incrementando esse valor a cada iteração. Mas se esquecermos de incrementar a rodada, entramos em um *loop* infinito.

Em casos como esse, existe um outro *loop* que simplifica essa ideia de começar com um valor e incrementá-lo até chegar em um valor final: o *loop* **for**.

Para entender o *loop*, ou laço **for**, podemos ir até o console do Python para ver o seu funcionamento. A ideia é definirmos o valor inicial e o valor final, que o *loop* o incrementa automaticamente. Para definir o valor inicial e final, utilizamos a função embutida `range()` , passando-os por parâmetro, definindo assim a série de valores. A sintaxe é a seguinte:

```
Para variável em uma série de valores:
    Faça algo
```

Isso, em Python, pode ficar assim:

```
for rodada in range(1, 10):
```

O `range(1, 10)` vai gerar o intervalo de números inteiros de 1 a 9. Na primeira iteração, o valor da variável `rodada` será 1, depois 2 e até chegar ao valor final da função `range()` menos 1, isto é, o segundo parâmetro da função não é inclusivo. No exemplo acima, a série de valores é de 1 a 9. Podemos confirmar isso imprimindo o valor da variável `rodada` no console do Python:

```
>>> for rodada in range(1,10):
...     print(rodada)
...
1
2
3
4
5
6
7
8
9
```

Com a função `range()` podemos definir um *step* (um passo), que é o intervalo entre os elementos. Por padrão o *step* tem valor igual a 1 mas podemos alterar este valor passando um terceiro parâmetro para a função:

```
>>> for rodada in range(1,10,2):
...     print(rodada)
...
1
3
5
7
9
```

Veja que o intervalo entre cada elemento da séria agora é 2, a cada iteração o laço pula dois passos (incrementa 2). Mas não necessariamente precisamos usar a função `range()` no `for` , podemos passar os valores da sequência manualmente conseguindo o mesmo resultado:

```
>>> for rodada in [1,2,3,4,5]:
...     print(rodada)
...
1
2
3
4
5
```

Tanto o `while` quanto o `for` podem ser usados no jogo. Conseguiremos o mesmo resultado mas o código fica mais verboso com o `while`, além de corrermos o risco de esquecer de incrementar a rodada (`rodada = rodada + 1`) e nosso código entrar em um *loop* infinito. Neste casos, é preferível utilizar o comando `for`.

3.16 EXERCÍCIOS - UTILIZANDO O FOR NO JOGO

1. Substitua o comando `while` pelo `for` começando no 1 e indo até o `total_de_tentativas`. Não esqueça de remover a declaração da variável `rodada` e o seu incremento dentro do loop:

```
numero_secreto = 42
total_de_tentativas = 3

for rodada in range(1, total_de_tentativas):
    print('Tentativa {} de {}'.format(rodada, total_de_tentativas))

    chute = int(input('Digite o seu número: '))
    print('Você digitou: ', chute)

    acertou = numero_secreto == chute
    maior = chute > numero_secreto
    menor = chute < numero_secreto

    if (acertou):
        print('Você acertou!')
    elif (maior):
        print('Você errou! O seu chute foi maior que o número secreto')
    elif (menor):
        print('Você errou! O seu chute foi menor que o número secreto')

print('Fim do jogo!')
```

2. É importante saber que o `for` não é obrigado a ter parênteses. Podemos testar e ver que o programa dá apenas 2 tentativas. Isso porque, como foi falado anteriormente, o segundo parâmetro da função `range` não é inclusivo, no caso do nosso jogo, `range(1,3)` irá gerar a série 1 e 2 somente. Portanto, vamos somar 1 ao `total_de_tentativas` dentro da função range:

```
for rodada in range(1, total_de_tentativas + 1):
```

3. Teste novamente o jogo e veja que tudo está funcionando perfeitamente!

4. (opcional) Crie um nível de dificuldade para o jogo. Crie uma variável chamada `nível` e peça para o usuário escolher em qual nível ele deseja jogar. O nível é mensurável de acordo com o total de tentativas: nível 1(tentativas = 20), nível 2(tentativas = 10) e nível 3 (tentativas = 5).

5. (opcional) Acrescente um total de pontos ao jogador que deve iniciar com 1000 e a cada chute errado deve ser subtraído do total de pontos um valor que corresponde a diferença entre o chute e o número secreto. Para este exercício você vai precisar da função `abs()` . Veja na documentação do Python como ela funciona.

Editora Casa do Código com livros de uma forma diferente

Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.
Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

Casa do Código, ebook com preço de ebook.

Capítulo 4

ESTRUTURA DE DADOS

No capítulo passado criamos o jogo da adivinhação, agora vamos criar o jogo da Forca. Vamos começar, com os conhecimentos que temos até aqui, para estruturar nosso jogo. Vamos criar um arquivo chamado *forca.py* na pasta *jogos*:

```
|_ home
|_ jogos
    |_ adivinhacao.py
    |_ forca.py
```

Como no jogo da adivinhação, devemos adivinhar uma palavra secreta, nada mais justo do que defini-la em uma variável. Por enquanto, a palavra será fixa, com o valor **banana**:

```
print('*********************************')
print('***Bem vindo ao jogo da Forca!***')
print('*********************************')

palavra_secreta = 'banana'

print('Fim do jogo')
```

Mais à frente deixaremos essa palavra secreta mais dinâmica.

Como estamos tratando de um jogo da forca, o usuário deve acertar uma palavra e chutar letras. Além disso, precisa saber se o usuário acertou ou errou e saber se foi enforcado ou não. Então precisaremos de 2 variáveis booleanas `enforcou` e `acertou` para guardar esta informação e o jogo continuará até uma delas for `True` ; para tal acrescentamos um laço `while` :

```
acertou = False
enforcou= False

while(not acertou and not enforcou):
    print('Jogando...')
```

O usuário do jogo também vai chutar letras. Além disso, se o chute for igual a uma letra contida na `palavra_secreta` , quer dizer que o usuário encontrou uma letra. Podemos utilizar um laço `for` para tratar isso, assim como fizemos no jogo da adivinhação para apresentar as tentativas e rodadas:

```
while(not acertou and not errou):
    chute = input('Qual letra?')

    posicao = 0
    for letra in palavra_secreta:
        if (chute == letra):
            print('Encontrei a letra {} na posição {}'.format(letra, index))
```

```
        posicao = posicao + 1

        print('Jogando...')
```

Como uma *string* é uma sequência de letras, o *loop* `for` vai iterar por cada letra.

Nossa palavra_secreta é 'banana'. E se o usuário chutar a letra 'A' ao invés de 'a'? Vamos testar:

```
>>> $python3 jogos/forca.py
*********************************
***Bem vindo ao jogo da Forca!***
*********************************
Qual letra? A
Jogando...
Qual letra?
```

O Python é *case-sensitive*, ou seja 'a' e 'A' são distintos para o interpretador. Será preciso acrescentar este tratamento e fazer o jogo aceitar como acerto tanto 'a' como 'A'. *String* (`str`) é um tipo embutido no Python que possui algumas funções prontas e uma delas vai nos ajudar neste problema.

Existe uma função chamada `upper()` que devolve uma *string* com todas as letras maiúsculas. Também possui a `lower()` que devolve uma *string* com todas as letras minúsculas:

```
>>> texto = 'python'
>>> texto.upper()
'PYTHON'
>>>
>>> texto = 'PYTHON'
>>> texto.lower()
'python'
```

Agora precisamos modificar a condição `chute == letra` do `if` para `chute.upper() == letra.upper()` :

```
if (chute.upper() == letra.upper()):
    print('Encontrei a letra {} na posição {}'.format(letra, index))
```

Dessa maneira, o jogador pode chutar tanto 'a' como 'A' que o tratamento do `if` vai considerar todas como maiúsculas.

Agora podemos testar novamente com chute igual a 'A' (ou 'a') e vemos o programa funcionar como o esperado:

```
*********************************
***Bem vindo ao jogo da Forca!***
*********************************
Qual letra? A
Encontrei a letra a na posição 1
Encontrei a letra a na posição 3
Encontrei a letra a na posição 5
Jogando...
Qual letra?
```

Atualmente, já dizemos ao jogador em que posição a letra que ele chutou está na palavra secreta, caso a letra exista na palavra. Mas em um jogo real de forca, o jogador vê quantas letras há na palavra secreta.

Algo como:

```
Qual letra? _ _ _ _ _ _
```

E se ele encontrar alguma letra, a mesma tem a sua lacuna preenchida. Ao digitar a letra "a", ficaria:

```
_ a _ a _ a
```

Muito mais intuitivo, não? Vamos implementar essa funcionalidade. Para exibir as letras dessa forma, precisamos guardar os chutes certos do usuário, mas como fazer isso?

Para tal, o Python nos oferece um tipo de estrutura de dados que nos permite guardar mais de um valor. Essa estrutura é a `list` (lista). Para criar uma lista, utilizamos colchetes ([]):

```
>>> valores = []
>>> type(valores)
<class 'list'>
```

Assim como a *string*, `list` também é uma sequência de dados. Podemos ver sua documentação através da função `help()` :

```
>>> help(list)
    Help on class list in module builtins:
    class list(object)
     |  list() -> new empty list
     |  list(iterable) -> new list initialized from iterable's items
     |
     |  Methods defined here:
     |
     |  __add__(self, value, /)
     |      Return self+value.
     |
     |  __contains__(self, key, /)
     |      Return key in self.
     |
     |  __delitem__(self, key, /)
     |      Delete self[key].
     |
     |  __eq__(self, value, /)
     |      Return self==value.
     |
     |  __ge__(self, value, /)
     |      Return self>=value.

     #código omitido
```

Vemos o que podemos fazer com uma lista. Podemos, por exemplo, verificar o seu valor mínimo com `min` e o seu máximo com `max` . Nossa lista ainda está vazia mas já podemos iniciá-la com alguns valores e utilizar essas funções para verificar o seus valores máximo e mínimo:

```
>>> valores = [0, 1, 2, 3]
>>> min(valores)
0
>>> max(valores)
3
```

Para acessar um valor específico, podemos acessá-lo através do seu índice (posição). O primeiro

elemento da lista possui índice 0, o segundo possui índice 1 e assim por diante:

```
>>> valores = [0, 1, 2, 3]
>>> valores[2]
2
>>> valores[0]
0
```

Para modificar um valor, basta usar o operador de atribuição em uma determinada posição:

```
>>> valores[0] = 4
>>> valores
[4, 1, 2, 3]
```

É possível saber o tamanho da lista com a função `len` e verificar se determinado valor está guardado nela com o comando `in` :

```
>>> valores = [0, 1, 2, 3]
>>> len(valores)
4
>>> 0 in valores
True
>>> 6 in valores
False
```

Além disso, existem funções específicas da lista, que podem ser acessadas na documentação: https://docs.python.org/3.6/library/stdtypes.html#mutable-sequence-types.

Podemos adicionar elementos ao final da lista com a função `append()` , exibir e remover um elemento de determinada posição com a função `pop()` , entre diversas outras funcionalidades.

Agora que sabemos como guardar valores em uma lista, podemos voltar ao nosso jogo e guardar os acertos do usuário. Como queremos exibir os espaços vazios primeiro, criaremos uma lista com eles, na mesma quantidade de letras da palavra secreta:

```
palavra_secreta = 'banana'
letras_acertadas = ['_', '_', '_', '_', '_', '_']
```

Já temos a posição da letra (também chamado de índice). Logo, caso o chute seja correto, basta guardar a letra dentro da lista, na sua posição correta e imprimir a lista após o laço `for` :

```
posicao = 0

for letra in palavra_secreta:
    if (chute.upper() == letra.upper()):
        letras_acertadas[posicao] = letra
    posicao = posicao + 1

print(letras_acertadas)
```

Ou seja, para cada letra na palavra secreta, o programa vai verificar se `chute` é igual a `letra` . Em caso afirmativo, adiciona a letra na posição correta e incrementa a `posicao` após o bloco do `if` .

Ao executar o jogo e chutar algumas letras, temos:

```
$ python3 jogos/forca.py
*********************************
***Bem vindo ao jogo da Forca!***
*********************************
Qual letra? b
['b', '_', '_', '_', '_', '_']
Jogando...
Qual letra? a
['b', 'a', '_', 'a', '_', 'a']
Jogando...
Qual letra?
```

A saída ainda não está visualmente agradável. Para ficar ainda melhor, vamos exibir a lista no início do jogo também e excluir o print('Jogando...') para o código ficar mais limpo.

```
print(letras_acertadas)

while(not acertou and not errou):
    chute = input('Qual letra?')

    #código omitido
```

4.1 EXERCÍCIOS: JOGO DA FORCA

Neste exercício, vamos aproveitar nosso novo conhecimento em listas para fazer com que o jogo da forca se lembre das letras acertadas pelo jogador.

1. Crie um arquivo chamado *forca.py* na pasta *jogos*:

```
|_ home
|_ jogos
    |_ adivinhacao.py
    |_ forca.py
```

2. Primeiro, precisamos mostrar para o jogador a mensagem de abertura, similar ao que foi feito no jogo da adivinhação:

```
print('*********************************')
print('***Bem vindo ao jogo da Forca!***')
print('*********************************')
```

3. Crie a variável *palavra_secreta* que será iniciada com o valor 'banana' e uma lista para representar as letras acertadas.

```
palavra_secreta = 'banana'
letras_acertadas = ['_', '_', '_', '_', '_', '_']
```

4. Crie as variáveis booleanas `acertou` e `errou` que vamos utilizar no laço `while`. E a variável `erros` que guardaremos o número de erros do usuário:

```
acertou = False
enforcou= False
erros = 0

while(not acertou and not enforcou):
    # código
```

5. Crie a variável `chute` que vai guardar entrada do usuário.

```
while(not acertou and not enforcou):
    chute = input('Qual letra? ')
```

6. Dentro do `while` crie um *loop* `for` para que vai checar se a letra existe na palavra secreta. Se o chute for igual a letra digitada pelo jogador, vamos adicionar a letra na nossa lista `letras_acertadas` na posição correta

```
while(not acertou and not enforcou):
    chute = input('Qual letra? ')

    posicao = 0
    for letra in palavra_secreta:
        if (chute == letra):
            letras_acertadas[posicao] = letra
        posicao += 1
```

7. Modifique a condição do `if` para considerarmos apenas letras maiúsculas utilizando a função `upper` de *strings*. E atualize a variável chute

```
if (chute.upper() == letra.upper()):
```

8. Agora precisamos incrementar a variável *erros* caso o jogador não acerte a letra. Se o chute é uma letra dentro de *palavra_secreta*, quer dizer que o jogador acertou, caso contrário incrementamos a variável *erros*.Para isso, teremos mais um bloco `if/else` :

```
if(chute in palavra_secreta):
    posicao = 0
    for letra in palavra_secreta:
        if(chute.upper() == letra.upper()):
            letras_acertadas[index] = letra
        posicao += 1
else:
    erros += 1
```

9. Atualize a variável `acertou` . Se todas as letras ainda não foram acertadas, quer dizer que o usuário ainda não finalizou o jogo, ou seja, letras*acertas ainda contém espaços vazios (''*). Dentro o *loop* `while` , após o comando `for` , vamos atualizar a variável *acertou* e utilizar o operador `not in` :

```
acertou = '_' not in letras_acertadas.
```

Podemos ler o código acima como "'_' não está contido em letras_acertadas".

10. Vamos também atualizar a variável `enforcou` . Vamos considerar que se o jogador errar 7 vezes ele perde o jogo. Como a variável inicia com 0 (zero), quer dizer que quando ela for igual a 6 ele se enforca. Atualize a variável dentro do *loop* `while` após a variável `acertou` :

```
acertou = "_" not in letras_acertadas.
enforcou = erros == 6
```

11. Para que o jogador acompanhe o resultado a cada chute que ele der, ao final do laço `while` imprima também a lista `letras_acertadas` para que ele veja como ele está indo no jogo:

```
while(not enforcou and not acertou):

    #código omitido

    print(letras_acertadas)
```

12. E claro, para dar uma dica ao nosso jogador de quantas letras a palavra tem, vamos colocar acima do while um print inicial para que ele veja de início qual o tamanho da palavra:

```
print(letras_acertadas)

while (not acertou and not enforcou):
    ...
```

13. Por fim, vamos imprimir uma mensagem de "Você ganhou!" se o usuário adivinhar a letra e "Você perdeu" caso tenha cometido 7 erros. Após o laço `while` , fora dele, acrescente:

```
if(acertou):
    print('Você ganhou!!')
else:
    print('Você perdeu!!')

print('Fim do jogo')
```

14. Faça o teste e veja na resposta se seu código funcionando.

```
$ python3 jogos/forca.py
```

No final, seu código deve estar parecido com este:

```
def jogar():
    print('*********************************')
    print('***Bem vindo ao jogo da Forca!***')
    print('*********************************')

    palavra_secreta = 'banana'
    letras_acertadas = ['_', '_', '_', '_', '_', '_']

    enforcou = False
    acertou = False
    erros = 0

    print(letras_acertadas)

    while(not enforcou and not acertou):

        chute = input("Qual letra? ")

        if(chute in palavra_secreta):
            posicao = 0
            for letra in palavra_secreta:
                if(chute.upper() == letra.upper()):
                    letras_acertadas[posicao] = letra
                posicao = index + 1
        else:
            erros += 1

        enforcou = erros == 6
        acertou = '_' not in letras_acertadas
        print(letras_acertadas)
```

```
if(acertou):
    print('Você ganhou!!')
else:
    print('Você perdeu!!')
print('Fim do jogo')
```


Editora Casa do Código com livros de uma forma diferente

Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.
Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.]

4.2 SEQUÊNCIAS

Desenvolvemos um novo jogo e conhecemos um novo tipo de dado do Python que são as listas. Uma lista é uma sequência de valores e o Python possui outros tipos de dados que também são sequências. Neste momento, conheceremos um pouco de cada uma delas.

Sequências são *containers*, um tipo de dado que contém outros dados. Existem três tipos básicos de sequência: `list` (lista) , `tuple` (tupla) e *`range`* (objeto de intervalo). Outro tipo de sequência famoso que já vimos são as *strings* que são sequências de texto.

Sequências podem ser mutáveis ou imutáveis. Sequências imutáveis não podem ter seus valores modificados. Tuplas, *strings* e *ranges* são sequências imutáveis, enquanto listas são sequências mutáveis.

As operações na tabela a seguir são suportadas pela maioria dos tipos de sequência, mutáveis e imutáveis. Na tabela abaixo, s e t são sequências do mesmo tipo, n , i , j e k são inteiros e x é um objeto arbitrário que atende a qualquer tipo e restrições de valor impostas por s .

Operação	Resultado
x in s	True se um item de s é igual a *x*
x not in s	False se um item de s é igual a *x*
s + t	Concatenação de s e s
s *n ou n* s	Equivalente a adicionar s a si mesmo n vezes
s[i]	Elemento na posição i de s

s[i:j]	Fatia s de i para j
s[i:j:k]	Fatia s de i para j com o passo k
len(s)	Comprimento de s
min(s)	Menor item de s
max(s)	Maior item de s
s.count(x)	Número total de ocorrências de x em s

- **Listas**

Uma lista é uma sequência de valores onde cada valor é identificado por um índice iniciado por 0. São similares a *strings* (coleção de caracteres) exceto pelo fato de que os elementos de uma lista podem ser de qualquer tipo. A sintaxe é simples, listas são delimitadas por colchetes e seus elementos separados por vírgula:

```
>>> lista1 = [1, 2, 3, 4]
>>> lista1
[1, 2, 3, 4]
>>>
>>>lista2 = ['python', 'java', 'c#']
>>> lista2
['python', 'java', 'c#']
```

O primeiro exemplo é uma lista com 4 inteiros, o segundo é uma lista contendo três *strings*. Mas listas não precisam necessariamente conter elementos de mesmo tipo. Podemos ter listas heterogêneas:

```
>>> lista = [1, 2, 'python', 3.5, 'java']
>>> lista
[1, 2, 'python', 3.5, 'java']
```

Nossa *lista* possui elementos do tipo `int` , `float` e `str` . Se queremos selecionar um elemento específico utilizamos o operador `[ ]` passando a posição:

```
>>> lista = [1, 2, 3, 4]
>>> lista[0]
1
>>> lista[1]
2
>>> lista[2]
3
>>> lista[3]
4
>>> lista[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Quando tentamos acessar a posição 4 fazendo `lista[4]` , aparece o erro `IndexError` dizendo que a lista excedeu seu limite já que não há um quinto elemento (índice 4).

O Python permite passar valores negativos como índice que vai devolver o valor naquela posição de forma reversa:

```
>>> lista = [1, 2, 3, 4]
>>> lista[-1]
4
>>> lista[-2]
3
```

Também é possível usar a função `list()` para criar uma lista passando um tipo que pode ser iterável como uma *string*:

```
>>> lista = list('python')
>>> lista
['p', 'y', 't', 'h', 'o', 'n']
```

Listas são muito úteis, por exemplo:

```
meses = ['Janeiro', 'Fevereiro', 'Março', 'Abril', 'Junho', 'Julho', 'Agosto', 'Setembro', 'Outub
ro', 'Novembro', 'Dezembro']

n = 1

while(n < 4):
    mes = input("Escolha um mês (1-12): ")
    if 1 <= mes <= 12:
        print('O mês é {}'.format(mes[mes-1]))
    n += 1
```

E testamos:

```
>>> Escolha um mês (1-12): 4
O mês é Abril
>>> Escolha um mês (1-12): 11
O mês é Novembro
>>> Escolha um mês (1-12): 6
O mês é Junho
```

Primeiro criamos um lista, relacionamos seus índices com os meses do ano, recuperamos seus valores através da entrada do usuário e imprimimos na tela o mês escolhido (mes[mes-1]) indexado a lista a partir do zero. Usamos um laço `while` que faz nosso programa entrar em *loop* e ser rodado 3 vezes.

Além de acessar um valor específico utilizando o índice, podemos acessar múltiplos valores através do fatiamento. Também utilizamos colchetes para o fatiamento. Suponha que queremos acessar os dois primeiros elementos de uma lista:

```
>>> lista = [2, 3, 5, 7, 11]
>>> lista[0:2]
[2, 3]
```

Podemos ter o mesmo comportamento fazendo:

```
>>> lista[:2]
[2, 3]
```

Se queremos todos os valores excluindo os dois primeiros, fazemos:

```
>>> lista[2:]
[5, 7, 11]
```

Ou utilizamos ínidces negativos:

```
>>> lista[-3:]
[5, 7, 11]
```

Também podemos fatiar uma lista de modo a pegar elementos em um intervalo específico:

```
>>> lista[2:4]
[5, 7]
```

As listas também possuem funcionalidades prontas e podemos manipulá-las através de funções embutidas. A lista tem uma função chamada `append()` que adiciona um dado na lista:

```
>>> lista = []
>>> lista.append('zero')
>>> lista.append('um')
>>> lista
['zero', 'um']
```

A função `append()` só consegue inserir um elemento por vez. Se quisermos inserir mais elementos podemos somar ou multiplicar listas, ou então utilizar a função `extend()` :

```
>>> lista = ['zero', 'um']
>>> lista.extend(['dois', 'três',])
>>> lista += ['quatro', 'cinco']
>>> lista + ['seis']
['zero', 'um', 'dois', 'três', 'quatro', 'cinco', 'seis']
>>> lista * 2
['zero', 'um', 'dois', 'três', 'quatro', 'cinco', 'seis', 'zero', 'um', 'dois', 'três', 'quatro',
'cinco', 'seis']
```

Isso é possível já que listas são sequências **mutáveis**, ou seja conseguimos adicionar, remover e modificar seus elementos. Para imprimir o conteúdo de uma lista utilizamos o comando `for` :

```
for valor in lista:
... print(valor)
...
zero
um
dois
três
quatro
cinco
```

- **Tuplas**

Uma tupla é uma lista **imutável**, ou seja, uma tupla é uma sequência que não pode ser alterada depois de criada. Uma tupla é definida de forma parecida com uma lista com a diferença do delimitador. Enquanto listas utilizam colchetes como delimitadores, as tuplas usam parênteses:

```
>>> dias = ('domingo', 'segunda', 'terça', 'quarta', 'quinta', 'sexta', 'sabado')
>>> type(dias)
<class 'tuple'>
```

Podemos omitir os parênteses e inserir os elementos separados por vírgula:

```
>>> dias = 'domingo', 'segunda', 'terça', 'quarta', 'quinta', 'sexta', 'sabado'
```

```
>>> type(dias)
>>> <class 'tuple'>
```

Note que, na verdade, é a vírgula que faz uma tupla, não os parênteses. Os parênteses são opcionais, exceto no caso da tupla vazia, ou quando são necessários para evitar ambigüidade sintática.

Assim como as listas, também podemos usar uma função para criar uma tupla passando um tipo que pode ser iterável como uma *string* ou uma lista. Essa função é a `tuple()` :

```
>>> texto = 'python'
>>> tuple(texto)
('p', 'y', 't', 'h', 'o', 'n')
>>> lista = [1, 2, 3, 4]
>>> tuple(lista)
(1, 2, 3, 4)
```

As regras para os índices são as mesmas das listas, exceto para elementos também imutáveis. Como são imutáveis, uma vez criadas não podemos adicionar nem remover elementos de uma tupla. O método `append()` da lista não existe na tupla:

```
>>> dias.append('sabado2')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
>>>
>>> dias[0]
'domingo'
>>>
>>> dias[0] = 'dom'
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Não é possível atribuir valores aos itens individuais de uma tupla, no entanto, é possível criar tuplas que contenham objetos mutáveis, como listas.

```
>>> lista = [3, 4]
>>> tupla = (1, 2, lista)
>>> tupla
(1, 2, [3, 4])
>>> lista = [4, 4]
>>> tupla
(1, 2, [4, 4])
>>> tupla[2] = [3, 4]
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

As tuplas são imutáveis e geralmente contêm uma sequência heterogênea de elementos já as listas são mutáveis e seus elementos geralmente são homogêneos e são acessados pela iteração da lista, mas isso não é uma regra.

Quando é necessário armazenar uma coleção de dados que não pode ser alterada, prefira usar tuplas a listas. Outra vantagem é que tuplas podem ser usadas como chaves de dicionários.

Tuplas são frequentemente usadas em programas Python. Um uso bastante comum são em funções

que recebem múltiplos valores. As tuplas implementam todas as operações de sequência comuns.

- **Range**

O range é um tipo de seqüência imutável de números e é comumente usado para *looping* de um número específico de vezes em um comando `for` já que representam um intervalo. O comando **range** gera um valor contendo números inteiros sequenciais, obedecendo a sintaxe:

```
range(inicio, fim)
```

O número finalizador, o *fim*, não é incluído na sequência. Vejamos um exemplo:

```
>>> sequencia = range(1, 3)
>>> print(sequencia)
range(1, 3)
```

O range não imprime os elementos da sequência, ele apenas armazena seu início e seu final. Para imprimir seus elementos precisamos de um laço `for` :

```
>>> for valor in range(1, 3):
...     print(valor)
...
1
2
```

Observe que ele não inclui o segundo parâmetro da função range na sequência. Outra característica deste comando é a de poder controlar o passo da sequência adicionando um terceiro parâmetro, isto é, a variação entre um número e o seu sucessor:

```
>>> for valor in range(1, 10, 2):
...     print(valor)
...
1
3
5
7
9
```

Os intervalos implementam todas as operações de seqüência comuns, exceto concatenação e repetição (devido ao fato de que objetos de intervalo só podem representar sequências que seguem um padrão estrito e a repetição e a concatenação geralmente violam esse padrão).

4.3 CONJUNTOS

O Python também inclui um tipo de dados para conjuntos. Um conjunto, diferente de uma sequência, é uma coleção **não ordenada** e que **não admite elementos duplicados**.

Chaves ou a função `set()` podem ser usados para criar conjuntos.

```
>>> frutas = {'laranja', 'banana', 'uva', 'pera', 'laranja', 'uva', 'abacate'}
>>> frutas
>>> {'uva', 'abacate', 'pera', 'banana', 'laranja'}
>>> type(frutas)
```

```
<class 'set'>
```

Usos básicos incluem testes de associação e eliminação de entradas duplicadas. Os objetos de conjunto também suportam operações matemáticas como união, interseção, diferença e diferença simétrica. Podemos transformar um texto em um conjunto com a frunção `set()` e testar os operações:

```
>>> a =  set('abacate')
>>> b = set('abacaxi')
>>> a
{'a', 'e', 'c', 't', 'b'}
>>> b
{'a', 'x', 'i', 'c', 'b'}
>>> a - b                                # diferença
{'e', 't'}
>>> a | b                                # união
{'c', 'b', 'i', 't', 'x', 'e', 'a'}
>>> a & b                                # interseção
{'a', 'c', 'b'}
>>> a ^ b                                # diferença simétrica
{'i', 't', 'x', 'e'}
```

Note que para criar um conjunto vazio você tem que usar `set()` , não `{}` ; o segundo cria um dicionário vazio, uma estrutura de dados que discutiremos na próxima seção.

```
>>> a = set()
>>> a
set()
>>> b = {}
>>> b
{}
>>> type(a)
<class 'set'>
>>> type(b)
<class 'dict'>
```

4.4 DICIONÁRIOS

Vimos que `list` , `tuple` , `range` e `str` são sequências ordenadas de objetos e *sets* são coleções de elementos não ordenados. Dicionário é outra estrutura de dados em Python e seus elementos, como os conjuntos, não são ordenados. Essa não é a principal diferença com as listas, os dicionários são estruturas poderosas e muito utilizadas já que podemos acessar seus elementos através de chaves e não de sua posição. Em outras linguagens este tipo é conhecido como "matrizes associativas".

Qualquer chave de um dicionário é associada (ou mapeada) a um valor. Os valores podem ser qualquer tipo de dado do Python. Portanto, os dicionários são pares de chave-valor não ordenados.

Os dicionários pertencem ao tipo de mapeamento integrado e não sequenciais como as listas, tuplas e *strings*. Vamos ver como isso funciona no código e criar um dicionário com dados de uma pessoa:

```
>>> pessoa = {'nome': 'João', 'idade': 25, 'cidade': 'São Paulo'}
>>> pessoa
'nome': 'João', 'idade': 25, 'cidade': 'São Paulo'
```

Os dicionários são delimitados por chaves ({}) e suas chaves ('nome', 'idade' e 'cidade') por aspas. Já os valores podem ser de qualquer tipo, no exemplo acima temos duas *strings* e um `int`.

O que será que acontece se tentarmos acessar seu primeiro elemento?

```
>>> pessoa[0]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: 0
```

Não é possível acessar um elemento de um dicionário por um índice como na lista. Devemos acessar por sua chave:

```
>>> pessoa['nome']
'João'
>>> pessoa['idade']
25
```

Se precisarmos adicionar algum elemento, por exemplo, o país, basta fazermos:

```
>>> pessoa1['país'] = 'Brasil'
>>> pessoa1
{'nome': 'João', 'idade': 25, 'cidade': 'São Paulo', 'país': 'Brasil'}
```

Como sempre acessamos seus elementos através de chaves, o dicionário possui um método chamado `keys()` que devolve o conjunto de suas chaves:

```
>>> pessoa1.keys()
dict_keys(['nome', 'idade', 'cidade', 'pais'])
```

Assim como um método chamado `values()` que retorna seus valores:

```
>>> pessoa1.values()
dict_values(['João', 25, 'São Paulo', 'Brasil'])
```

Note que as chaves de um dicionário não podem ser iguais para não causar conflito. Além disso, somente tipos de dados imutáveis podem ser usados como chaves, ou seja, nenhuma lista ou dicionário pode ser usado. Caso isso aconteça, recebemos um erro:

```
>>> dic = {[1, 2, 3]: 'valor'}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Já as tuplas, como chaves, são permitidas:

```
>>> dic = {(1, 2, 3): 'valor'}
>>> dic
{(1, 2, 3): 'valor'}
```

Também podemos criar dicionários utilizando a função **dict()**:

```
>>> a = dict(um=1, dois=2, três=3)
>>> a
{'três': 3, 'dois': 2, 'um': 1}
```

Já conhece os cursos online Alura?

alura

A **Alura** oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex-aluno da Caelum tem 15% de desconto neste link!

Conheça os cursos online Alura.

4.5 EXERCÍCIOS: ESTRUTURA DE DADOS

1. Dada a lista = [12, -2, 4, 8, 29, 45, 78, 36, -17, 2, 12, 8, 3, 3, -52] faça um programa que:

   a) imprima o maior elemento

   b) imprima o menor elemento

   c) imprima os números pares

   d) imprima o número de ocorrências do primeiro elemento da lista

   e) imprima a média dos elementos

   f) imprima a soma dos elementos de valor negativo

2. Faça um programa que leia dados do usuário (nome, sobrenome, idade) adicione em uma lista e imprima seus elementos na tela.

3. Faça um programa que leia 4 notas, mostre as notas e a média na tela.

4. Faça um programa utilizando um `dict` que leia dados de entrada do usuário. O usuário deve entrar com os dados de uma pessoa como nome, idade e cidade onde mora (fique livre para acrescentar outros). Após isso, você deve imprimir os dados como o exemplo abaixo:

   ```
   nome: João
   idade: 20
   cidade: São Paulo
   ```

5. (Opcional) Utilize o exercício anterior e adicione a pessoa em uma lista. Pergunte ao usuário se ele deseja adicionar uma nova pessoa. Após adicionar dados de algumas pessoas, você deve imprimir todos os dados de cada pessoa de forma organizada.

CAPÍTULO 5
FUNÇÕES

Objetivos:

- entender o conceito de função
- saber e usar algumas funções embutidas da linguagem
- criar uma função

5.1 O QUE É UMA FUNÇÃO?

O conceito de função é um dos mais importantes na matemática. Em computação, uma função é uma sequência de instruções que computa um ou mais resultados que chamamos de parâmetros. No capítulo anterior utilizamos algumas funções já prontas do Python como o **print()**, **input()**, **format()** e **type()**.

Também podemos criar nossas próprias funções. Por exemplo, quando queremos calcular a razão do espaço pelo tempo podemos definir uma função recebendo estes parâmetros:

```
f(espaco, tempo) = espaco/tempo
```

Essa razão do espaço pelo tempo é o que chamamos de velocidade média na física. Podemos então dar este nome a nossa função:

```
velocidade(espaco, tempo) = espaco/tempo
```

Se um carro percorreu uma distância de 100 metros em 20 segundos podemos calcular sua velocidade média:

```
velocidade(100, 20) = 100/20 =     5 m/s
```

O Python permite definirmos funções como essa da velocidade média. A sintaxe é muito parecida com a da matemática. Para definirmos uma função no Python utilizamos o comando **def**:

```
def velocidade(espaco, tempo):
    pass
```

Logo após o `def` vem o nome da função e entre parêntese vêm os seus parâmetros. Uma função também tem um escopo, um bloco de instruções em que colocamos os cálculos e estes devem seguir a identação padrão do Python (4 espaços a direita).

Como nossa função ainda não faz nada, utilizamos a palavra chave **pass** para dizer ao interpretador

que definiremos os cálculo depois. A palavra `pass` não é usada apenas em funções, podemos usar em qualquer bloco de comandos como nas instruções `if`, `while` e `for`, por exemplo.

Vamos substituir a palavra `pass` pelos cálculos que nossa função deve executar:

```
def velocidade(espaco, tempo):
    v = espaco/tempo
    print('velocidade: {} m/s'.format(v))
```

Nossa função faz o cálculo da velocidade média e utiliza a função `print()` do Python para imprimir na tela. Vamos testar nossa função:

```
>>> velocidade(100, 20)
velocidade: 5 m/s
```

De maneira geral, uma função é um estrutura para agrupar um conjunto de instruções que podem ser reutilizadas. Agora qualquer parte do nosso programa pode chamar a função velocidade quando precisar calcular a velocidade média de um veículo, por exemplo. E podemos chamá-la mais de uma vez, o que significa que não precisamos escrever o mesmo código novamente.

Funções são conhecidas por diversos nomes em linguagens de programação como subrotinas, rotinas, procedimentos, métodos e subprogramas.

Podemos ter funções sem parâmetros. Por exemplo, podemos ter uma função que diz 'oi' na tela:

```
>>> def diz_oi():
...     print("oi")
>>>
>>> diz_oi()
oi
```


Já conhece os cursos online Alura?

A **Alura** oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex-aluno da Caelum tem 15% de desconto neste link!

Conheça os cursos online Alura.

5.2 PARÂMETROS DE FUNÇÃO

Um conjunto de parâmetros consiste em uma lista com nenhum ou mais elementos que podem ser

obrigatórios ou opcionais. Para um parâmetro ser opcional atribuímos um valor padrão (default) para ele - o mais comum é utilizar **None**. Por exemplo:

```
def dados(nome, idade=None):
    print('nome: {}'.format(nome))
    if(idade is not None):
        print('idade: {}'.format(idade))
    else:
        print('idade: não informada')
```

O código da função acima recebe uma idade como parâmetro e faz uma verificação com uma instrução **if**: se a idade for diferente de *None* ela vai imprimir a idade, caso contrário vai imprimir *idade não informada*. Vamos testar passando os dois parâmetros e depois apenas o nome:

```
>>> dados('joão', 20)
nome: joão
idade: 20
```

Agora passando apenas o nome:

```
>>> dados('joão')
nome: joão
idade: não informada
```

E o que acontece se passarmos apenas a idade?

```
>>> dados(20)
nome: 20
idade: não informada
```

Veja que o Python obedece a ordem dos parâmetros. Nossa intenção era passar o número 20 como `idade` mas o interpretador vai entender que estamos passando o `nome` porque não avisamos isso à ele. Caso queiramos passar apenas a `idade` , devemos nomear o parâmetro:

```
>>> dados(idade=20)
 File "<stdin>", line 1, in <module>
TypeError: dados() missing 1 required positional argument: 'nome'
```

O interpretador vai acusar um erro já que não passamos o atributo obrigatório `nome` .

5.3 FUNÇÃO COM RETORNO

E se ao invés de apenas mostrar o resultado, quisermos utilizar a velocidade média para fazer outro cálculo como calcular a aceleração? Da maneira como está, nossa função `velocidade()` não conseguimos utilizar seu resultado final para cálculos.

```
Exemplo:
aceleracao = velocidade(parametros) / tempo
```

Para conseguirmos este comportamento, precisamos que nossa função **retorne** o valor calculado por ela. No Python, utilizamos o comando **return**:

```
def velocidade(espaco, tempo):
    v = espaco/tempo
```

```
    return v
```

Testando:

```
>>> velocidade(100, 20)
5.0
```

Ou ainda, podemos atribuir a uma variável:

```
>>> resultado = velocidade(100, 20)
>>> resultado
5.0
```

E conseguimos utilizar no cálculo da aceleração:

```
>>> aceleracao = velocidade(100, 20)/20
0.25
```

Uma função pode conter mais de um comando **return**. Por exemplo, nossa função `dados()` que imprime o nome e a idade, pode agora retornar uma *string*. Repare que, neste caso, temos duas situações possíveis: a que a idade é passada por parâmetro e a que ela não é passada. Aqui, teremos dois comandos **return**:

```
def dados(nome, idade=None):
    if(idade is not None):
        return ('nome: {} \nidade: {}'.format(nome, idade))
    else:
        return ('nome: {} \nidade: não informada'.format(nome))
```

Apesar da função possuir dois comandos **return**, ela tem apenas um retorno -- vai retornar um ou o outro. Quando a função encontra um comando **return** ela não executa mais nada que vier depois dele dentro de seu escopo.

5.4 RETORNANDO MÚLTIPLOS VALORES

Apesar de uma função executar apenas um retorno, em Python podemos retornar mais de uma valor. Vamos fazer uma função calculadora que vai retornar os resultados de operações básicas entre dois números: adição(+) e subtração(-), nesta ordem.

Para retornar múltiplos valores, retornamos os resultados separados por virgula:

```
def calculadora(x, y):
    return x+y, x-y

>>> calculadora(1, 2)
(3, -1)
```

Qual será o tipo de retorno desta função? Vamos perguntar ao interpretador através da função type:

```
>>> type(calculadora(1,2))
<class 'tuple'>
```

Da maneira que definimos o retorno, a função devolve uma tupla. Neste caso específico, poderíamos

retornar um dicionário e usar um laço `for` para imprimir os resultados:

```
>>> def calculadora(x, y):
...        return {'soma':x+y, 'subtração':x-y}
...
>>> resultados = calculadora(1, 2)
>>> for key in resultados:
...     print('{}: {}'.format(key, resultados[key]))
...
soma: 3
subtração: -1
```

Você pode também fazer o curso PY-14 dessa apostila na Caelum

Querendo aprender ainda mais sobre? Esclarecer dúvidas dos exercícios? Ouvir explicações detalhadas com um instrutor?
A Caelum oferece o **curso PY-14** presencial nas cidades de São Paulo, Rio de Janeiro e Brasília, além de turmas incompany.

Consulte as vantagens do curso *Python e Orientação a Objetos*

5.5 EXERCÍCIOS: FUNÇÕES

1. Defina uma função chamada velocidade_media() em um script chamado `funcoes.py` que recebe dois parâmetros: a distância percorrida (em metros) e o tempo (em segundos) gasto.

   ```
   def velocidade_media(distancia, tempo):
       pass
   ```

2. Agora vamos inserir as instruções, ou seja, o que a função deve fazer. Vamos inserir os comandos para calcular a velocidade média e guardar o resultado em uma variável velocidade:

   ```
   def velocidade_media(distancia, tempo):
       velocidade = distancia/tempo
   ```

3. Vamos fazer a função imprimir o valor da velocidade média calculada:

   ```
   def velocidade_media(distancia, tempo):
       velocidade = distancia/tempo
       print(velocidade)
   ```

4. Teste o seu código chamando a função para os valores abaixo e compare os resultados com seus colegas:

   - distância: 100, tempo = 20
   - distância: 150, tempo = 22

- distância: 200, tempo = 30
- distância: 50, tempo = 3

5. Modifique a função `velocidade_media()` de modo que ela retorne o resultado calculado.
6. Defina uma função `soma()` que recebe dois números como parâmetros e calcula a soma entre eles.
7. Defina uma função `subtracao()` que recebe dois números como parâmetros e calcula a diferença entre eles.
8. Agora faça uma função `calculadora()` que recebe dois números como parâmetros e retorna o resultado da soma e da subtração entre eles.
9. Modifique a função `calculadora()` do exercício anterior e faça ela retornar também o resultado da multiplicação e divisão dos parâmetros.
10. Chame a função `calculadora()` com alguns valores.
11. (opcional) Defina uma função `divisao()` que recebe dois números como parâmetros, calcula e retorna o resultado da divisão do primeiro pelo segundo. Modifique a função `velocidade_media()` utilizando a função `divisao()` para calcular a velocidade. Teste o seu código chamando a função `velocidade_media()` com o valores abaixo: a. distância: 100, tempo = 20 b. distância: -20, tempo = 10 c. distância: 150, tempo = 0

5.6 NÚMERO ARBITRÁRIO DE PARÂMETROS (*ARGS)

Podemos passar um número arbitrário de parâmetros em uma função. Utilizamos as chamadas variáveis mágicas do Python: ***args** e ****kwargs**. Muitos programadores tem dificuldades em entender essas variáveis. Vamos entender o que elas são.

Não é necessário utilizar exatamente estes nomes: `*args` e `**kwargs` . Apenas o asterisco(*), ou dois deles(**), será necessário. Podemos optar, por exemplo, em escrever `*var` e `**vars` . Mas `*args` e `**kwargs` é uma convenção entre a comunidade que também seguiremos.

Primeiro aprenderemos a usar o `*args` . É usado, assim como o `**kwargs` , em definições de funções. `*args` e `**kwargs` permitem passar um número variável de argumentos de uma função. O que a variável significa é que o programador ainda não sabe de antemão quantos argumentos serão passados para sua função, apenas que são muitos. Então, neste caso usamos a palavra chave `*args` .

Veja um exemplo:

```
def teste(arg, *args):
    print('primeiro argumento normal: {}'.format(arg))
    for arg in args:
        print('outro argumento: {}'.format(arg))

teste('python', 'é', 'muito', 'legal')
```

Que vai gerar a saída:

```
primeiro argumento normal:  python
outro argumento:  é
outro argumento:  muito
outro argumento:  legal
```

O parâmetro `arg` é como qualquer outro parâmetro de função, já o `*args` recebe múltiplos parâmetros. Viu como é fácil? Também poderíamos conseguir o mesmo resultado passando um `list` ou `tuple` de argumentos, acrescido do asterisco:

```
lista = ["é", "muito", "legal"]
teste('python', *lista)
```

Ou ainda:

```
tupla = ("é", "muito", "legal")
teste('python', *tupla)
```

O `*args` então é utilizado quando não sabemos de antemão quantos argumentos queremos passar para uma função. O asterisco (`*`) executa um empacotamento dos dados para facilitar a passagem de parâmetros, e a função que recebe este tipo de parâmetro é capaz de fazer o desempacotamento.

5.7 NÚMERO ARBITRÁRIO DE CHAVES (**KWARGS)

O `**kwargs` permite que passemos o tamanho variável da palavra-chave dos argumentos para uma função. Você deve usar o `**kwargs` se quiser manipular argumentos nomeados em uma função. Veja um exemplo:

```
def minha_funcao(**kwargs):
    for key, value in kwargs.items():
        print('{0} = {1}'.format(key, value))

>>> minha_funcao(nome='caelum')
nome = caelum
```

Também podemos passar um dicionário acrescido de dois símbolos asterisco já que se trata de chave e valor:

```
dicionario = {'nome': 'joao', 'idade': 25}
minha_funcao(**dicionario)
idade = 25
nome = joao
```

A diferença é que o `*args` espera uma tupla de argumentos posicionais enquanto o `**kwargs` um dicionário com argumentos nomeados.

Seus livros de tecnologia parecem do século passado?

Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.
Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.]

5.8 EXERCÍCIO - *ARGS E **KWARGS

1. Crie um arquivo com uma função chamada `teste_args_kwargs()` que recebe três argumentos e imprime cada um deles:

```
def teste_args_kwargs(arg1, arg2, arg3):
    print("arg1: ", arg1)
    print("arg2: ", arg2)
    print("arg3: ", arg3)
```

2. Agora vamos chamar a função utilizando o ***args**:

```
args = ('um', 2, 3)
teste_args_kwargs(*args)
```

   Que gera a saída:

```
arg1: um
arg2: 2
arg3: 3
```

3. Teste a mesma função usando o ****kargs**. Para isso criaremos um dicionário com três argumentos:

```
kwargs = {'arg3': 3, 'arg2': 'dois', 'arg1': 'um'}
teste_args_kwargs(**kwargs)
```

   Que deve gerar a saída:

```
arg1: um
arg2: dois
arg3: 3
```

4. **(Opcional)** Tente chamar a mesma função mas adicionando um quarto argumento na variável args e kwargs dos exercícios anteriores. O que acontece se a função recebe mais do que 3 argumentos?

5. De que maneira você resolveria o problema do exercício anterior?

Discuta com o instrutor e seus colegas quando usar ***args** e ****kwargs**.

5.9 EXERCÍCIO - FUNÇÃO JOGAR()

1. Vamos começar definindo uma função jogar que conterá toda lógica do jogo da forca. Abra o arquivo forca.py e coloque o código do jogo em uma função `jogar()` :

```
def jogar():
    #código do jogo aqui
```

2. Vamos tentar executar nosso jogo pelo terminal. Navegue até a pasta jogos e execute forca.py através do comando python3:

```
$ cd jogos
$ python3 forca.py
$
```

   Veja que nada aconteceu já que precisamos chamar a função `jogar()` do arquivo forca.py. Para fazer isso temos que importar o arquivo forca.py dentro do interpretador do python através do comando **import**:

```
$ python3.6
>>> import forca
```

   E chamar a função através do arquivo 'forca':

```
>>> forca.jogar()
"*********************************"
"***Bem vindo ao jogo da Forca!***"
"*********************************"
['_', '_', '_', '_', '_', '_']
Qual letra?
```

   Agora nosso jogo funciona como esperado.

3. Faça o mesmo com o jogo da adivinhação e execute o jogo.

5.10 MÓDULOS E O COMANDO IMPORT

Ao importar o arquivo forca.py estamos importando um **módulo** de nosso programa, que nada mais é do que um arquivo. Vamos verificar o tipo de `forca` :

```
>>> type(forca)
<class 'module'>
```

Veja que o tipo é um **módulo**. Antes de continuarmos com nosso jogo, vamos aprender um pouco mais sobre arquivos e módulos. Vamos melhorar ainda mais nosso jogo da Forca e utilizar o que aprendemos de funções para organizar nosso código.

Agora é a melhor hora de aprender algo novo

Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

Conheça a Alura Cursos Online.

CAPÍTULO 6

ARQUIVOS

Uma funcionalidade que ainda nos atrapalha no jogo da forca é a palavra secreta, que atualmente está fixa. Se queremos que a palavra seja diferente, devemos modificá-la no código.

A nossa ideia é ler palavras de um arquivo de texto, e dentre elas escolhemos uma palavra aleatoriamente, que será a palavra secreta do jogo

6.1 ESCRITA DE UM ARQUIVO

Para abrir um arquivo, o Python possui a função `open()` . Ela recebe dois parâmetros: o primeiro é o nome do arquivo a ser aberto, e o segundo parâmetro é o modo que queremos trabalhar com esse arquivo - se queremos ler ou escrever. O modo é passado através de uma *string*: "w" para escrita e "r" para leitura.

No jogo, faremos a leitura de um arquivo. Antes, vamos testar como funciona a escrita no terminal do Python 3:

```
>>> arquivo = open('palavras.txt', 'w')
```

O modo é opcional e o modo padrão é o "r" de leitura (*reading*) que veremos mais adiante.

O arquivo criado se chama 'palavras.txt' e está no modo de escrita. É importante saber que o modo de escrita sobrescreve o arquivo, se o mesmo existir. Se a intenção é apenas adicionar conteúdo ao arquivo, utilizamos o modo "a" (abreviação para *append*).

Agora que temos o arquivo vamos aprender a escrever algum conteúdo nele. Basta chamar a partir do arquivo a função `write()` , passando para ela o que se quer escrever no arquivo:

```
>>> arquivo.write('banana')
6
>>> arquivo.write('melancia')
8
```

O retorno dessa função é o número de caracteres de cada texto adicionado no arquivo.

Agora é a melhor hora de aprender algo novo

Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

Conheça a Alura Cursos Online.

6.2 FECHANDO UM ARQUIVO

Quando estamos trabalhando com arquivos, devemos nos preocupar em fechá-lo. Para fechá-lo usamos a função `close()` :

```
>>> arquivo.close()
```

Após isso, podemos verificar o conteúdo do arquivo. Repare que ele foi criado na mesma pasta em que o comando para abrir o console do Python 3 foi executado. Se você tentar fechar um arquivo que já está fechado não vai surtir efeito algum, nem mesmo um erro. Abra o arquivo na pasta criada e verifique seu conteúdo:

```
bananamelancia
```

As palavras foram escritas em uma mesma linha. Mas como escrever uma nova linha?

6.3 ESCREVENDO PALAVRAS EM NOVAS LINHAS

A primeira coisa que devemos fazer é abrir o arquivo novamente, dessa vez utilizando o modo *'a'*, de `append` :

```
arquivo = open('palavras.txt', 'a')
```

Vamos escrever novamente no arquivo, mas dessa vez com a preocupação de criar uma nova linha após cada conteúdo escrito. Para representar uma nova linha em código adicionamos o *\n* ao final do que queremos escrever:

```
>>> arquivo.write('morango\n')
8
>>> arquivo.write('manga\n')
```

Ao fechar o arquivo e verificar novamente o seu conteúdo, vemos:

```
bananamelanciamorango
```

```
manga
```

A palavra morango ainda ficou na mesma linha, mas como especificamos na sua adição que após a palavra deverá ter uma quebra de linha, a palavra manga foi adicionada abaixo, em uma nova linha.

Por fim, vamos mover este arquivo para nosso projeto e ajeitar suas palavras quebrando as linhas.

6.4 EXERCICIOS

1. Vamos abrir o terminal e navegar até nossa pasta *jogos* dentro de *home* e iniciar o interpretador do Python 3:

   ```
   $ cd jogos
   $ python3
   ```

   Lembrando que nossa estrutura de arquivos está assim:

   ```
   |_ home
       |_ jogos
           |_ advinhacao.py
           |_ forca.py
   ```

2. Crie um arquivo chamado *palavras.txt* no modo escrita. Insira o código logo após a mensagem de abertura:

   ```
   >>> arquivo = open("palavras.txt", "w")
   ```

   Abra a pasta `\home\jogos` e veja se o arquivo foi criado como esperado:

   ```
   |_ home
       |_ jogos
           |_ advinhacao.py
           |_ forca.py
           |_ palavras.txt
   ```

3. Vamos começar a escrever no nosso arquivo utilizando a função `write()` as palavras que usaremos no nosso jogo da forca:

   ```
   >>> arquivo.write('banana\n')
   7
   >>> arquivo.write('melancia\n')
   9
   >>> arquivo.write('morango\n')
   8
   >>> arquivo.write('manga\n')
   6
   ```

   Note que ao final de cada palavra temos que acrescentar o "\n" para a quebra de linha, que vai facilitar na hora da leitura.

4. É uma boa prática fechar o arquivo depois de utilizá-lo, assim outros programas ou processos podem ter acesso ao arquivo e ele não fica preso apenas ao nosso programa Python.

   ```
   >>> arquivo.close()
   ```

Para saber mais

Além do **r**, **w** e **a** existe o modificador **b** que é utilizado quando se deseja trabalhar no modo binário. Para abrir uma imagem no modo leitura devemos usar:

```
imagem = open('foto.jpg', 'rb')
```

Editora Casa do Código com livros de uma forma diferente

Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.
Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

Casa do Código, ebook com preço de ebook.

6.5 LENDO UM ARQUIVO

Ainda no terminal do Python 3, veremos o funcionamento da leitura de um arquivo. Como agora o arquivo *palavras.txt* está na pasta do projeto `jogos` , devemos executar o comando que abre o terminal do Python 3 na pasta do projeto.

```
$ cd jogos
$ python3
```

Vamos então abrir o arquivo no modo de leitura, basta passar o nome do arquivo e a letra "r" para a função `open()` , como já visto anteriormente.

```
arquivo = open('palavras.txt', 'r')
```

Diferente do modo "w", abrir um arquivo que não existe no modo "r" não vai criar um arquivo. Se "palavras.txt" não existir, o Python vai lançar o erro `FileNotFoundError` :

```
>>> arquivo.open('frutas.txt', 'r')
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'frutas.txt'
```

Como o arquivo *frutas.txt* não existe na pasta jogos, o Pyhton não consegue encontrar e acusa que

não existe nenhum arquivo ou diretório com este nome na pasta raiz.

Como abrimos o arquivo no modo de leitura, a função `write()` não é suportada:

```
>>> arquivo.write("oi")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
io.UnsupportedOperation: not writable
```

Para ler o arquivo inteiro, utilizamos a função `read()` :

```
>>> arquivo.read()
'banana\nmelancia\nmorango\nmanga\n'
```

Mas ao executar a função novamente, será retornado uma *string* vazia:

```
>>> arquivo.read()
''
```

Isso acontece porque o arquivo é como um fluxo de linhas, que começa no início do arquivo como se fosse um cursor. Ele vai descendo e lendo o arquivo. Após ler tudo, ele fica posicionado no final do arquivo. E quando chamamos a função `read()` novamente não há mais conteúdo pois ele todo já foi lido.

Portanto, para ler o arquivo novamente, devemos fechá-lo e abrí-lo outra vez:

```
>>> arquivo.close()
>>> arquivo = open('palavras.txt', 'r')
>>> arquivo.read()
```

6.6 LENDO LINHA POR LINHA DO ARQUIVO

Não queremos ler todo o conteúdo do arquivo mas ler linha por linha. Como já foi visto, um arquivo é um fluxo de linhas, ou seja, uma sequência de linhas. Sendo uma sequência podemos utilizar um laço `for` para ler cada linha do arquivo:

```
>>> arquivo = open('palavras.txt', 'r')
>>> for linha in arquivo:
...     print(linha)
...
banana

melancia

morango

manga
```

Repare que existe uma linha vazia entre cada fruta. Isso acontece porque estamos utilizando a função `print()` que também acrescenta, por padrão, um `\n` . Agora vamos utilizar outra função, a `readline()` , que lê apenas uma linha do arquivo:

```
>>> arquivo = open('palavras.txt', 'r')
>>> linha = arquivo.readline()
```

```
>>> linha
'banana\n'
```

Há um `\n` ao final de cada linha, de cada palavra, mas queremos somente a palavra. Para tirar espaços em branco no início e no fim da *string*, basta utilizar a função `strip()`, que também remove caracteres especiais, como o `\n` - para mais informações consulte a documentação de *strings*. Sabendo disso tudo, já podemos implementar a funcionalidade de leitura de arquivo no nosso jogo:

```
arquivo = open('palavras.txt', 'r')
palavras = []

for linha in arquivo:
    linha = linha.strip()
    palavras.append(linha)

arquivo.close()
```

Agora já temos todas as palavras na lista, mas como selecionar uma delas aleatoriamente?

6.7 GERANDO UM NÚMERO ALEATÓRIO

Sabemos que cada elemento da lista possui uma posição e vimos no treinamento anterior como gerar um número aleatório. A biblioteca que sabe gerar um número aleatório é a random. Vamos testá-la no terminal do Python 3, primeiro importando-a:

```
>>> import random
```

Para gerar o número aleatório utilizamos a biblioteca e chamamos a função `randrange()`, que recebe o intervalo de valores que o número aleatório deve estar. Então vamos passar o valor 0 (equivalente à primeira posição da nossa lista) e 4 (lembrando que o número é exclusivo, ou seja, o número aleatório será entre 0 e 3, equivalente à última posição da nossa lista):

```
>>> import random
>>> random.randrange(0, 4)
0
>>> random.randrange(0, 4)
1
>>> random.randrange(0, 4)
3
>>> random.randrange(0, 4)
1
>>> random.randrange(0, 4)
3
```

Sabendo disso, vamos implementar esse código no nosso jogo.

Já conhece os cursos online Alura?

A **Alura** oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex-aluno da Caelum tem 15% de desconto neste link!

Conheça os cursos online Alura.

6.8 EXERCÍCIOS - LEITURA DE ARQUIVOS

1.Vamos começar abrindo o arquivo "palavras.txt" no nosso script `forca.py` que criamos no exercício anterior. É importante já acrescentar o comando para fechá-lo para não esquecer no futuro:

```
def jogar():
    print('*********************************')
    print('***Bem vindo ao jogo da Forca!***')
    print('*********************************')

    arquivo = open('palavras.txt', 'r')
    arquivo.close()

    #restante do código
```

1. Agora vamos criar uma lista chamada `palavras` e fazer um laço `for` para acessar cada linha para guardar na lista:

```
arquivo = open('palavras.txt', 'r')
palavras = []

for linha in arquivo:
    palavras.append(linha)

arquivo.close()
```

2. Como precisamos remover o **\n** ao final da linha, usaremos a função `strip()` em cada linha:

```
def jogar():

    arquivo = open('palavras.txt', 'r')
    palavras = []

    for linha in arquivo:
        linha - linha.strip()
        palavras.append(linha)

    arquivo.close()
```

3. Devemos importar a biblioteca *random* para gerar um número que vai de 0 até a quantidade de palavras da nossa lista. Usaremos a função `len()` para saber o tamanho da lista e a `randrange()` para gerar um número randômico de um intervalo específico:

```
import random

print('*********************************')
print('***Bem vindo ao jogo da Forca!***')
print('*********************************')

arquivo = open('palavras.txt', 'r')
palavras = []

for linha in arquivo:
    linha = linha.strip()
    palavras.append(linha)

arquivo.close()

numero = random.randrange(0, len(palavras))
```

4. Agora que temos o número aleatório, vamos utilizá-lo como índice para acessar a lista e atribuir essa palavra à variável palavra_secreta:

```
import random

print("*********************************")
print("***Bem vindo ao jogo da Forca!***")
print("*********************************")

arquivo = open("palavras.txt", "r")
palavras = []

for linha in arquivo:
    linha = linha.strip()
    palavras.append(linha)

arquivo.close()

numero = random.randrange(0, len(palavras))

palavra_secreta = palavras[numero].upper()
letras_acertadas = ['_' for letra in palavra_secreta]
```

5. Por fim, temos que deixar nossa variável `letras_acertadas` dinâmica, com número de letras de acordo com nossa `palavra_secreta` . Vamos utilizar um `for` dentro da lista para gerar um '_' para cada letra de acordo com o tamanho da palavra_secreta:

```
letras_acertadas = ['_' for letra in palavra_secreta]
```

6. Como já garantimos que a `palavra_secreta` está toda em letras maiúsculas com o código `palavras[numero].upper()` , modificaremos o `chute` para o primeiro **if** continuar funcionando

```
chute = input('Qual a letra? ')
chute = chute.strip().upper()
```

Podemos executar o jogo e notar que a palavra é selecionada aleatoriamente!

Mas agora a nossa função cresceu bastante, com várias funcionalidades e responsabilidades. Então, no próximo capítulo, organizaremos melhor o nosso código, separando-o em funções e deixando-o mais fácil de entender.

6.9 PARA SABER MAIS - COMANDO WITH

Já falamos da importância de fechar o arquivo, certo? Veja o código abaixo que justamente usa a função `close()` :

```
arquivo = open('palavras.txt', 'r')
palavras = logo.read()
arquivo.close()
```

Imagine que algum problema aconteça na hora da leitura quando a função read() é chamada. Será que o arquivo é fechado quando o erro ocorre?

Se for algum erro grave, o programa pode parar a execução sem ter fechado o arquivo e isto seria bastante ruim. Para evitar esse tipo de situação existe no Python uma sintaxe especial para abertura de arquivo:

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

Repare o comando `with` usa a função `open()` mas não a função `close()` . Isso não será mais necessário já que o comando `with` vai se encarregar de fechar o arquivo para nós mesmo que aconteça algum erro no código dentro de seu escopo. Muito melhor não?

6.10 MELHORANDO NOSSO CÓDIGO

Nos capítulos anteriores criamos dois jogos, avançamos no jogo da Forca e implementamos leitura de palavras em um arquivo. Agora vamos utilizar o que aprendemos de funções para encapsular nosso código e deixá-lo mais organizado. Vamos começar, com os conhecimentos que temos até aqui, para estruturar nosso jogo da Forca.

A função `jogar()` possui um código muito complexo, com muitas funcionalidades e responsabilidades.

Entre as funcionalidades que o código possui, está a apresentação do jogo, leitura do arquivo e inicialização da palavra secreta, entre outras. Vamos então separar as responsabilidades do código em funções, melhorando a sua legibilidade e organização.

Vamos começar com a mensagem de apresentação do nosso jogo e exportar o código para a função `imprime_mensagem_abertura()` . Não podemos nos esquecer de chamar essa função no início da função `jogar()` :

```
import random

def jogar():
    imprime_mensagem_abertura()

    #código omitido

def imprime_mensagem_abertura():
    print('*********************************')
    print('***Bem vindo ao jogo da Forca!***')
    print('*********************************')
```

Aqui não importa o local da função, ela pode ser declarada antes ou depois da função jogar().

O que fizemos foi **refatorar** nosso código. Refatoração é o processo de modificar um programa para melhorar a estrutura interna do código sem alterar seu comportamento externo. Veja que se executarmos nosso jogo da Forca, tudo funciona como antes:

```
$ python3
>>> import forca
>>> forca.jogar()
*********************************)
***Bem vindo ao jogo da Forca!***
*********************************
['_', '_', '_', '_', '_', '_']
Qual letra?
```

No próximo exercício vamos refatorar as demais partes do nosso código

Você pode também fazer o curso PY-14 dessa apostila na Caelum

Querendo aprender ainda mais sobre? Esclarecer dúvidas dos exercícios? Ouvir explicações detalhadas com um instrutor?
A Caelum oferece o **curso PY-14** presencial nas cidades de São Paulo, Rio de Janeiro e Brasília, além de turmas incompany.

Consulte as vantagens do curso *Python e Orientação a Objetos*

6.11 EXERCÍCIO - REFATORANDO O JOGO DA FORCA

1. Crie a função imprime_mensagem_abertura que vai isolar a mensagem de abertura do jogo:

```
import random

def jogar():
    imprime_mensagem_abertura()

    #código omitido
```

```python
def imprime_mensagem_abertura():
    print('*********************************')
    print('***Bem vindo ao jogo da Forca!***')
    print('*********************************')
```

2. Agora, vamos separar o código que realiza a leitura do arquivo e inicializa a palavra secreta na função `carrega_palavra_secreta()` :

```python
def carrega_palavra_secreta():
    arquivo = open('palavras.txt', 'r')
    palavras = []

    for linha in arquivo:
        linha = linha.strip()
        palavras.append(linha)

    arquivo.close()

    numero = random.randrange(0, len(palavras))
    palavra_secreta = palavras[numero].upper()
```

Só que a função `jogar()` irá reclamar que a palavra_secreta não existe. O que queremos é que, ao executar a função `carrega_palavra_secreta()` , que ela retorne a palavra secreta para nós, assim poderemos guardá-la em uma variável:

```python
import random

def jogar():

    imprime_mensagem_abertura()

    palavra_secreta = carrega_palavra_secreta()

    letras_acertadas = ["_" for letra in palavra_secreta]

    # restante do código omitido
```

Só que como faremos a função `carrega_palavra_secreta()` retornar um valor, no caso a `palavra_secreta` ? A `palavra_secreta` já existe, mas só dentro da função `carrega_palavra_secreta()` . Para que ela seja retornada, utilizamos a palavra-chave return:

```python
def carrega_palavra_secreta():
    arquivo = open('palavras.txt', 'r')
    palavras = []

    for linha in arquivo:
        linha = linha.strip()
        palavras.append(linha)

    arquivo.close()

    numero = random.randrange(0, len(palavras))
    palavra_secreta = palavras[numero].upper()

    return palavra_secreta
```

3. Agora vamos criar uma função que inicializa a lista de letras acertadas com o caractere '_'. Criaremos

a função `inicializa_letras_acertadas()`:

```
import random

def jogar():

    imprime_mensagem_abertura()

    palavra_secreta = carrega_palavra_secreta()

    letras_acertadas = inicializa_letras_acertadas()

    # código omitido

def inicializa_letras_acertadas():
    return ['_' for letra in palavra_secreta]
```

4. Mas a função `inicializa_letras_acertadas()` precisa ter acesso à palavra_secreta, pois ela não existe dentro da função, já que uma função define um escopo, e as variáveis declaradas dentro de uma função só estão disponíveis dentro dela. Então, ao chamar a função `inicializa_letras_acertadas()`, vamos passar palavra_secreta para ela por parâmetro:

```
import random

def jogar():

    imprime_mensagem_abertura()

    palavra_secreta = carrega_palavra_secreta()

    letras_acertadas = inicializa_letras_acertadas(palavra_secreta)

    # restante do código omitido

def inicializa_letras_acertadas(palavra):
    return ["_" for letra in palavra]
```

5. Vamos continuar refatorando nosso código. Criaremos a função `pede_chute()`, que ficará com o código que pede o chute do usuário, remove os espaços antes e depois, e o coloca em caixa alta. Não podemos nos esquecer de retornar o chute:

```
def jogar():
    # código omitido

    while (not acertou and not enforcou):
        chute = pede_chute()
        #código omitido

    #código omitido

def pede_chute():
    chute = input('Qual letra? ')
    chute = chute.strip().upper()
    return chute
```

6. Ainda temos o código que coloca o chute na posição correta, dentro da lista. Vamos colocá-lo dentro da função `marca_chute_correto()`:

```python
    while (not acertou and not enforcou):

        chute = pede_chute()

        if (chute in palavra_secreta):
            marca_chute_correto()
        else:
            erros += 1

        enforcou = erros == 6
        acertou = '_' not in letras_acertadas
        print(letras_acertadas)

    #código omitido

def marca_chute_correto():
    posicao = 0
    for letra in palavra_secreta:
        if (chute == letra):
            letras_acertadas[posicao] = letra
        posicao += 1
```

Mas a função `marca_chute_correto()` precisa ter acesso a três valores: `palavra_secreta` , `chute` e `letras_acertadas` . Então vamos passar esses valores por parâmetro

```python
if (chute in palavra_secreta):
    marca_chute_correto(chute, letras_acertadas, palavra_secreta)
```

E modificamos nossa função para receber esses parâmetros: ```python def marca_chute_correto(chute, letras_acertadas, palavra_secreta):

```
posicao = 0
for letra in palavra_secreta:
    if (chute == letra):
        letras_acertadas[posicao] = letra
    posicao += 1


1. Por fim, vamos remover a mensagem de fim de jogo e exportar os códigos que imprimem as mensagens d
e vencedor e perdedor do jogo:
```python
    if (acertou):
        imprime_mensagem_vencedor()
    else:
        imprime_mensagem_perdedor()
```

E criar as funções:

```python
    def imprime_mensagem_vencedor():
        print('Você ganhou!')

    def imprime_mensagem_perdedor():
        print('Você perdeu!')
```

Agora o nosso código está muito mais organizado e legível. Ao chamar todas as funções dentro da função `jogar()` , nosso código ficará assim:

```python
def jogar():
    imprime_mensagem_abertura()
    palavra_secreta = carrega_palavra_secreta()
```
```

```
letras_acertadas = inicializa_letras_acertadas(palavra_secreta)
print(letras_acertadas)

enforcou = False
acertou = False
erros = 0

while(not enforcou and not acertou):

    chute = pede_chute()

    if(chute in palavra_secreta):
        marca_chute_correto(chute, letras_acertadas, palavra_secreta)
    else:
        erros += 1

    enforcou = erros == 6
    acertou = "_" not in letras_acertadas

    print(letras_acertadas)

if(acertou):
    imprime_mensagem_vencedor()
else:
    imprime_mensagem_perdedor()
```

Por fim, podemos executar o nosso código, para verificar que o mesmo continua funcionando normalmente.

```
$ python3
>>> import forca
>>> forca.jogar()
*********************************
***Bem vindo ao jogo da Forca!***
*********************************
['_', '_', '_', '_', '_', '_']
Qual letra?
```

1. (opcional) Com a melhor organização do nosso código, vamos melhorar a exibição, a apresentação da forca, deixando o jogo mais amigável. Vamos começar com a mensagem de perdedor, alterando a função `imprime_mensagem_perdedor`. Ela ficará assim:

```
def imprime_mensagem_perdedor(palavra_secreta):
 print('Puxa, você foi enforcado!')
 print('A palavra era {}'.format(palavra_secreta))
 print("    _______________         ")
 print("   /               \        ")
 print("  /                 \       ")
 print("//                   \/\    ")
 print("\|   XXXX     XXXX   | /    ")
 print(" |   XXXX     XXXX   |/     ")
 print(" |   XXX       XXX   |      ")
 print(" |                   |      ")
 print(" \__      XXX      __/      ")
 print("   |\     XXX     /|        ")
 print("   | |           | |        ")
 print("   | I I I I I I I |        ")
 print("   |  I I I I I I  |        ")
 print("   \_             _/        ")
```

```
 print("     \_         _/          ")
 print("       \_______/            ")
```

Precisamos passar a `palavra_secreta` para função `imprime_mensagem_perdedor()`:

```
 if(acertou):
     imprime_mensagem_vencedor()
 else:
     imprime_mensagem_perdedor(palavra_secreta)
```

E modificamos o código de `imprime_mensagem_vencedor()` pra:

```
def imprime_mensagem_vencedor():
 print('Parabéns, você ganhou!')
 print("       ___________      ")
 print("      '._==_==_=_.'      ")
 print("      .-\\:      /-.    ")
 print("     | (|:.     |) |    ")
 print("      '-|:.     |-'     ")
 print("        \\::.    /      ")
 print("         '::. .'        ")
 print("           ) (          ")
 print("         _.' '._        ")
 print("        '-------'       ")
```

Vá até a pasta do curso e copie o código destas funções que estão em um arquivo chamado *funcoes_forca.py*.

2. (opcional) Por fim, crie a função `desenha_forca()`, que irá desenhar uma parte da forca, baseado nos erros do usuário. Como ela precisa acessar os erros, passe-o por parâmetro para a função:

```
 def desenha_forca(erros):
     pass
```

   E copie o conteúdo do código da função `desenha_forca()` do arquivo funcoes *forca.py* na pasta do curso para sua função.

3. Para finalizar, chame a função `desenha_forca` quando o jogador errar e aumente o limite de erros para 7.

```
 if(chute in palavra_secreta):
     marca_chute_correto(chute, letras_acertadas, palavra_secreta)
 else:
     erros += 1
     desenha_forca(erros)

 enforcou = erros == 7
 acertou = "_" not in letras_acertadas
```

4. Tente fazer o mesmo com o jogo da adivinhação, refatore partes do código e isole em funções. Além disso, use sua criatividade para customizar mensagens para o usuário do seu jogo.

Neste exercício praticamos bastante do que aprendemos no capítulo de função e finalizamos o jogo da forca.

Você pode estar se perguntando por que encapsulamos uma simples linha de código em uma função. Fizemos isso somente para deixar claro o que estamos fazendo, melhorando a **legibilidade do código**. Mas precisamos tomar cuidado com a criação de funções, pois criar funções desnecessariamente pode aumentar a complexidade do código.

Capítulo 7

ORIENTAÇÃO A OBJETOS

Considere um programa para um banco financeiro. É fácil perceber que uma entidade importante para o nosso sistema será uma conta. Primeiramente suponha que você tem uma conta nesse banco com as seguintes características: titular, número, saldo e limite. Vamos começar inicializando essas características:

```
>>> numero = '123-4'
>>> titular = "João"
>>> saldo = 120.0
>>> limite = 1000.0
```

E se a necessidade de representar mais de uma conta surgir? Vamos criar mais uma:

```
>>> numero1 = '123-4'
>>> titular1 = "João"
>>> saldo1 = 120.0
>>> limite1 = 1000.0
>>>
>>> numero2 = '123-5'
>>> titular2 = "José"
>>> saldo2 = 200.0
>>> limite2 = 1000.0
```

Nosso banco pode vir a crescer e ter milhares de contas e, da maneira que está o programa, seria muito trabalhoso dar manutenção.

E como utilizar os dados de uma determinada conta em outro arquivo? Podemos utilizar a estrutura do dicionário que aprendemos anteriormente e agrupar essas características. Isso vai ajudar a acessar os dados de uma conta específica:

```
conta = {"numero": '123-4', "titular": "João", "saldo": 120.0, "limite": 1000.0}
```

Agora é possível acessar os dados de uma conta pelo nome da chave:

```
>>> conta['numero']
'123-4'
>>> conta['titular']
'João'
```

Para criar uma segunda conta, crie outro dicionário:

```
conta2 = {"numero": '123-5', "titular": "José", "saldo": 200.0, "limite": 1000.0}
```

Avançamos em agrupar os dados de uma conta, mas ainda precisamos repetir seguidamente essa linha de código a cada conta criada. Podemos isolar esse código em uma função responsável por criar

uma conta:

```
def cria_conta():
    conta = {"numero": '123-4', "titular": "João", "saldo": 120.0, "limite": 1000.0}
    return conta
```

Mas ainda não é o ideal já que queremos criar contas com outros valores e tornar a criação dinâmica. Vamos, então, receber esse valores como parâmetros da função e por fim retornamos a conta:

```
def cria_conta(numero, titular, saldo, limite):
    conta = {"numero": numero, "titular": titular, "saldo": saldo, "limite": limite}
    return conta
```

Desta maneira é possível criar várias contas com dados diferentes:

```
>>> conta1 = cria_conta('123-4', 'João', 120.0, 1000.0)
>>> conta2 = cria_conta('123-5', 'José', 200.0, 1000.0)
```

Para acessar o número de cada uma delas, fazemos:

```
>>> conta1['numero']
'123-4'
>>> conta2['numero']
'123-5'
```

7.1 FUNCIONALIDADES

Já descrevemos as características de uma conta e nosso próximo passo será descrever suas funcionalidades. O que fazemos com uma conta? Ora, podemos depositar um valor em uma conta, por exemplo. Vamos criar uma função para representar esta funcionalidade. Além do valor a ser depositado, precisamos saber qual conta receberá este valor:

```
def deposita(conta, valor):
    conta['saldo'] = conta['saldo'] + valor
```

Veja que estamos repetindo conta['saldo'] duas vezes nessa linha de código. O Python permite escrever a mesma coisa de uma maneira mais elegante utilizando o '+=':

```
def deposita(conta, valor):
    conta['saldo'] += valor
```

Podemos fazer algo semelhante com a função `saca()` :

```
def saca(conta, valor):
    conta['saldo'] -= valor
```

Antes de testar essas funcionalidades, crie outra que mostra o extrato da conta:

```
def extrato(conta):
    print("numero: {} \nsaldo: {}".format(conta['numero'], conta['saldo']))
```

O extrato imprime as informações da conta utilizando a função `print()` . Agora podemos testar o código (supondo que o mesmo esteja em um arquivo chamado `teste.py`):

```
>>> from teste import cria_conta, deposita, saca, extrato
```

```
>>>
>>> conta = cria_conta('123-4', 'João', 120.0, 1000.0)
>>> deposita(conta, 15.0)
>>> extrato(conta)
numero: '123-4'
saldo: 135.0
>>> saca(conta, 20.0)
>>> extrato(conta)
numero: '123-4'
saldo 115.0
```

Ótimo! Nosso código funcionou como o esperado. Aplicamos algumas funções como `deposita()` e `saca()` e ao final pudermos checar o saldo final com a função `extrato()`.

Você pode também fazer o curso PY-14 dessa apostila na Caelum

Querendo aprender ainda mais sobre? Esclarecer dúvidas dos exercícios? Ouvir explicações detalhadas com um instrutor?
A Caelum oferece o **curso PY-14** presencial nas cidades de São Paulo, Rio de Janeiro e Brasília, além de turmas incompany.

Consulte as vantagens do curso *Python e Orientação a Objetos*

7.2 EXERCÍCIO: CRIANDO UMA CONTA

1. Crie uma pasta chamada `oo` em sua workspace e crie um arquivo chamado *teste_conta.py*

2. Crie a função chamada `cria_conta()`, que recebe como argumento `numero`, `titular`, `saldo` e `limite`:

```
def cria_conta(numero, titular, saldo, limite):
```

3. Dentro de `cria_conta()`, crie uma variável do tipo dicionário chamada `conta` com as chaves recebendo os valores dos parâmetros (`numero`, `titular`, `saldo` e `limite`) e ao final retorne a `conta`:

```
def cria_conta(numero, titular, saldo, limite):
    conta = {"numero": numero, "titular": titular, "saldo": saldo, "limite": limite}
    return conta
```

4. Crie uma função chamada `deposita()` no mesmo arquivo `teste_conta.py` que recebe como argumento uma `conta` e um `valor`. Dentro da função adicione o `valor` ao `saldo` da conta:

```
def deposita(conta, valor):
    conta['saldo'] += valor
```

5. Crie outra função chamada `saca()` que recebe como argumento uma `conta` e um `valor`. Dentro da função subtraia o `valor` do `saldo` da conta:

```
def saca(conta, valor):
    conta['saldo'] -= valor
```

6. E por fim, crie uma função chamada `extrato()`, que recebe como argumento uma `conta` e imprime o `numero` e o `saldo`:

```
def extrato(conta):
    print("numero: {} \nsaldo: {}".format(conta['numero'], conta['saldo']))
```

7. Navegue até a pasta `oo` pelo terminal, abra o console do Python3, importe o script e testes as funcionalidades:

```
>>>from teste_conta import cria_conta, deposita, saca, extrato
>>> conta = cria_conta('123-7', 'João', 500.0, 1000.0)
>>> deposita(conta, 50.0)
>>> extrato(conta)
numero: '123-7'
saldo: 550.0
>>> saca(conta, 20.0)
>>> extrato(conta)
numero: '123-7'
saldo 530.0
```

8. (Opcional) Acrescente uma documentação para o seu módulo `teste_conta.py` e utilize a função `help()` para testá-la.

Neste exercício criamos uma conta e juntamos suas características (número, titular, limite, saldo) e funcionalidades (sacar, depositar, tirar extrato) num mesmo arquivo. Mas o que fizemos até agora foi baseado no conhecimento procedural que tínhamos do Python3.

Por mais que tenhamos agrupado os dados de uma conta, essa ligação é frágil no mundo procedural e se mostra limitada. Precisamos pensar sobre o que escrevemos para não errar. O paradigma orientado a objetos vem para sanar essa e outras fragilidades do paradigma procedural que veremos a seguir.

7.3 CLASSES E OBJETOS

Ninguém deveria ter acesso ao saldo diretamente. Além disso, nada nos obriga a validar esse valor e podemos esquecer disso cada vez que utilizá-lo. Nosso programa deveria obrigar o uso das funções `saca()` e `deposita()` para alterar o saldo e não permitir alterar o valor diretamente:

```
conta3['saldo'] = 100000000.0
```

ou então:

```
conta3['saldo'] = -3000.0
```

Devemos manipular os dados através das funcionalidades `saca()` e `deposita()` e proteger os dados da conta. Pensando no mundo real, ninguém pode modificar o saldo de sua conta quando quiser,

a não ser quando vamos fazer um saque ou um depósito. A mesma coisa deve acontecer aqui.

Para isso, vamos primeiro entender o que é `classe` e `objeto`, conceitos importantes do paradigma **orientado a objetos** e depois veremos como isso funciona na prática.

Quando preparamos um bolo, geralmente, seguimos uma receita que define os ingredientes e o modo de preparação. A nossa conta é um objeto concreto assim como o bolo que também precisa de uma receita pré-definida. E a "receita" no mundo OO recebe o nome de **classe**. Ou seja, antes de criarmos um objeto definiremos uma classe.

Outra analogia que podemos fazer é entre o projeto de uma casa (a planta da casa) e a casa em si. O projeto é a **classe** e a casa, construída a partir desta planta, é o **objeto**. O projeto da conta, isto é, a definição da conta, é a classe. O que podemos construir (instanciar) a partir dessa classe, as contas de verdade, damos o nome de objetos.

Pode parecer óbvio, mas a dificuldade inicial do paradigma da orientação a objetos é justamente saber distinguir o que é classe e o que é objeto. É comum o iniciante utilizar, obviamente de forma errada, essas duas palavras como sinônimos.

O próximo passo será criar nossa classe `Conta` dentro de um novo arquivo Python, que receberá o nome de *conta.py*. Criar uma classe em Python é extremamente simples em termos de sintaxe. Vamos começar criando uma classe vazia. Depois criaremos uma instância, um objeto dessa classe, e utilizaremos a função `type()` para analisar o resultado:

```
class Conta:
    pass
```

```
>>> from conta import Conta
>>> conta = Conta()
>>> type(conta)
<class 'conta.Conta'>
```

Vemos porque estamos utilizando o modo interativo pelo terminal e o módulo onde se encontra a classe `Conta` é `conta` ou o arquivo `conta.py`. Agora temos uma classe `Conta`.

Como Python é uma linguagem dinâmica, podemos modificar esse objeto `conta` em tempo de execução. Por exemplo, podemos acrescentar **atributos** a ele:

```
>>> conta.titular = "João"
>>> print(conta.titular)
'João'
>>> conta.saldo = 120.0
>>> print(conta.saldo)
120.0
```

Mas o problema do código é que ainda não garantimos que toda instância de `Conta` tenha um atributo `titular` ou `saldo`. Portanto queremos uma forma padronizada da conta de maneira que possamos criar objetos com determinadas configurações iniciais.

Em linguagens orientadas a objetos existe uma maneira padronizada de criar atributos de um objeto. Geralmente fazemos isso através de uma função construtora - algo parecido com nossa função `cria_conta()` do exercício anterior.

7.4 CONSTRUTOR

Em Python, alguns nomes de métodos estão reservados para o uso da própria linguagem. Um desses métodos é o `__init__()` que vai inicializar o objeto. Seu primeiro parâmetro, assim como todo método de instância, é a própria instância. Por convenção chamamos este argumento de **self**. Vejamos um exemplo:

```
class Conta:
    def __init__(self, numero, titular, saldo, limite):
        self.numero = numero
        self.titular = titular
        self.saldo = saldo
        self.limite = limite
```

Agora, quando uma classe é criada, todos os seus atributos serão inicializados pelo método `__init__()`. Apesar de muitos programadores chamarem este método de construtor, ele não cria um objeto conta. Existe outro método, o `__new__()` que é chamado antes do `__init_()` pelo interpretador do Python. O método `__new__()` é realmente o construtor e é quem realmente cria uma instância de `Conta`. O método `__init__()` é responsável por inicializar o objeto, tanto é que já recebe a própria instância (self) criada pelo construtor como argumento. E dessa maneira garantimos que toda instância de uma `Conta` tenha os atributos que definimos.

Agora, se executarmos a linha de código abaixo, vai acusar um erro:

```
>>> from conta import Conta
>>> conta = Conta()
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: __init__() missing 4 required positional arguments: 'numero', 'titular', 'saldo', and 'lim
ite
```

O erro acusa a falta de 4 argumentos na hora de criar uma `Conta`. A classe `Conta` agora nos obriga a passar 4 atributos (numero, titular, saldo e limite) para criar uma conta:

```
>>> conta = Conta('123-4', 'João', 120.0, 1000.0)
```

Veja que em nenhum momento chamamos o método `__init__()`. Quem está fazendo isso por debaixo dos panos é o próprio Python quando executa `conta = Conta()`. Não só, como vimos, ele chama o método `__new__()` que devolve um instância do objeto e em seguida chama o método `__init__()` toda vez que criamos uma conta. Podemos ver isto funcionando imprimindo uma mensagem dentro do método `__init__()`:

```
def __init__(self, titular, numero, saldo, limite):
    print("inicializando uma conta")
    self.titular = titular
```

```
    self.numero = numero
    self.saldo = saldo
    self.limite = limite
```

e testar novamente:

```
>>> from conta import Conta
>>> conta = Conta('123-4', 'João', 120.0, 1000.0)
inicializando uma conta
```

Ao criar uma `Conta` , estamos pedindo para o Python criar uma nova instância de `Conta` na memória, ou seja, o Python alocará memória suficiente para guardar todas as informações da `Conta` dentro da memória do programa. O `__new__()` , portanto, devolve uma **referência**, uma seta que aponta para o objeto em memória e é guardada na variável `conta` .

Para manipularmos nosso objeto `conta` e acessar seus atributos utilizamos o operador "." (ponto):

```
>>> conta.titular
'João'
>>> conta.saldo
120.0
```

Como o **self** é a referência do objeto, ele chama `self.titular e self.saldo` da classe `Conta` .

Agora, além de funcionar como esperado, nosso código não permite criar uma conta sem os atributos que definimos anteriormente. Discuta com seus colegas e instrutor as vantagens da orientação a objetos até aqui.

Seus livros de tecnologia parecem do século passado?

Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.
Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

Casa do Código, Livros de Tecnologia.

7.5 MÉTODOS

Como vimos, além dos atributos, nossa conta deve possuir funcionalidades. No arquivo *teste_conta.py* criamos as funções `saca()` , `deposita()` e `extrato()` . No paradigma orientado a objetos as funcionalidades de um objeto são chamados de **métodos** - do ponto de vista do código, são as

funções dentro de uma classe.

Vamos criar o método `deposita()` na classe `Conta`. Aqui, assim como o método `__init__()`, o método `deposita()` deve receber a instância do objeto (self) além do valor a ser depositado:

```
class Conta:

    # método __init__() omitido

    def deposita(self, valor):
        self.saldo += valor
```

Isso acontece porque o método precisa saber qual objeto conta ele deve manipular, qual conta vai depositar um determinado valor - e podemos ter muitas contas criadas no nosso sistema.

Utilizamos o operador '.' (ponto) através do objeto `conta` para chamar o método `deposita`:

```
>>> conta.deposita(20.0)
```

O interpretador, ao ler esse código, associa o objeto `conta` ao argumento `self` do método - note que não precisamos passar a `conta` como argumento, isso é feito por debaixo dos panos pelo Python.

Faremos o mesmo para os métodos `saca()` e `extrato()`:

```
class Conta:

    # outros métodos omitidos

    def saca(self, valor):
        self.saldo -= valor

    def extrato(self):
        print("numero: {} \nsaldo: {}".format(self.numero, self.saldo))
```

Agora vamos testar nossos métodos:

```
>>> from conta import Conta
>>>
>>> conta = Conta('123-4', 'João', 120.0, 1000.0)
>>> conta.deposita(20.0)
>>> conta.extrato()
numero: '123-4'
saldo: 140.0
>>> conta.saca(15)
>>> conta.extrato()
numero: '123-4'
saldo: 125.0
```

O saldo inicial era de 120 reais. Depositamos 20 reais, sacamos 15 reais e tiramos o extrato que resultou em 125 reais.

Por fim, o código de nossa `Conta` vai ficar assim:

```
class Conta:

    def __init__(self, numero, titular, saldo, limite):
        self.numero = numero
```

```
        self.titular = tituar
        self.saldo = saldo
        self.limite = limite

    def deposita(self, valor):
        self.saldo += valor

    def saca(self, valor):
        self.saldo -= valor

    def extrato(self):
        print("numero: {} \nsaldo: {}".format(self.numero, self.saldo))
```

7.6 MÉTODOS COM RETORNO

Em outras linguagens como C++ e Java, um método sempre tem que definir o que retorna, nem que defina que não há retorno. Como vimos no capítulo sobre funções, no Python isso não é necessário - mas podemos retornar algo no método `saca()` , por exemplo, indicando se a operação foi bem sucedida ou não. Neste caso podemos retornar um valor booleano:

```
def saca(self, valor):
    if (self.saldo < valor):
        return False
    else:
        self.saldo -= valor
        return True
```

Veja que a declaração do método não mudou mas agora ele nos retorna algo (um *boolean*). A palavra chave **return** indica que o método vai terminar ali, retornando tal informação.

Exemplo de uso:

```
>>> from conta import Conta
>>> minha_conta.saldo = 1000
>>> consegui = minha_conta.saca(2000)
>>> if(consegui):
...    print(“consegui sacar”)
... else:
...    print(“não consegui sacar”)
>>>
'não consegui sacar'
```

Ou então, podemos eliminar a variável temporária, se desejado:

```
>>> from conta import Conta
>>> minha_conta.saldo = 1000
>>> if(minha_conta.saca(2000)):
...    print(“consegui sacar”)
... else:
...    print(“não consegui sacar”)
>>>
'não consegui sacar'
```

Mais adiante, veremos que algumas vezes é mais interessante lançar uma exceção (*exception*) nesses casos.

7.7 OBJETOS SÃO ACESSADOS POR REFERÊNCIA

O programa pode manter na memória não apenas uma `Conta`, mas mais de uma:

```
>>> from conta import Conta
>>> minha_conta = Conta()
>>> minha_conta.saldo = 1000
>>>
>>> meu_sonho = Conta()
>>> meu_sonho.saldo = 1500000
```

Quando criamos uma variável para associar a um objeto, na verdade, essa variável não guarda o objeto, e sim uma maneira de acessá-lo, chamada de **referência** (o self).

```
>>> c1 = Conta()
```

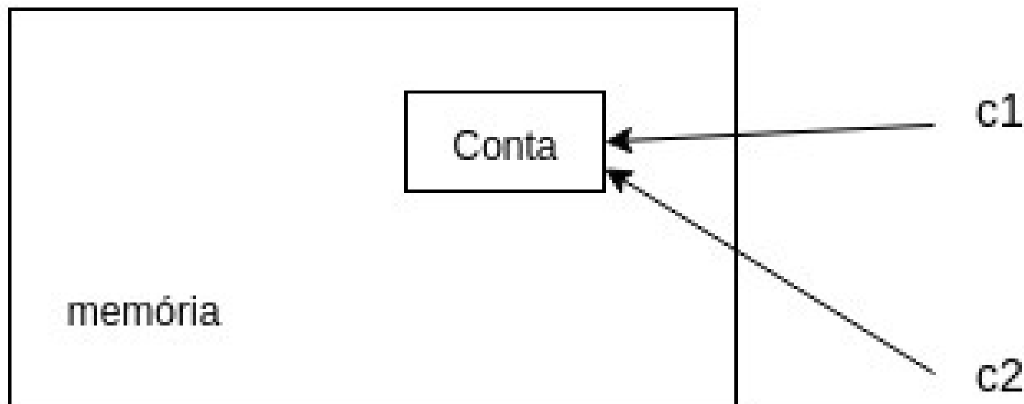
Ao fazer isso, já sabemos que o Python está chamando os métodos mágicos `__new__()` e `__init__()` que são responsáveis por construir e iniciar um objeto do tipo `Conta`.

O correto é dizer que `c1` se refere a um objeto. Não é correto dizer que `c1` é um objeto, pois `c1` é uma variável referência, apesar de, depois de um tempo, os programadores falarem "tenho um objeto `c1` do tipo `Conta`", mas apenas para encurtar a frase "Tenho uma referência `c1` a um objeto tipo `Conta`".

Vamos analisar o código abaixo:

```
>>> from conta import Conta
>>> c1 = Conta('123-4', 'João', 120.0, 1000.0)
>>> c2 = c1
>>> c2.saldo
120.0
>>> c1.deposita(100.0)
>>> c1.saldo
220.0
>>> c2.deposita(30.0)
>>> c2.saldo
250.0
>>> c1.saldo
250.0
```

O que aconteceu aqui? O operador "=" copia o valor de uma variável. Mas qual é o valor da variável `c1`? É o objeto? Não. Na verdade, o valor guardado é a referência (endereço) de onde o objeto se encontra na memória principal.



Ao fazer `c2 = c1`, `c2` passa a fazer referência para o mesmo objeto que `c1` referencia nesse instante. Quando utilizamos `c1` ou `c2`, neste código, estamos nos referindo ao MESMO objeto – são duas referências que apontam para o mesmo objeto.

Podemos notar isso através da função interna `id()` que retorna a referência de um objeto:

```
>>> id(c1)
140059774918104
>>> id(c2)
140059774918104
```

Internamente, `c1` e `c2` vão guardar um número que identifica em que posição da memória aquela `Conta` se encontra. Dessa maneira, ao utilizarmos o "." (ponto) para navegar, o Python vai acessar a `Conta` que se encontra naquela posição de memória, e não uma outra conta. Para quem conhece, é parecido com um ponteiro, porém você não pode manipulá-lo.

Outra maneira de notar esse comportamento é que o interpretador Python chamou os métodos `__new__()` e `__init__()` apenas uma vez (na linha `c1 = Conta('123-4', 'João', 120.0, 1000.0)`), então só pode haver um objeto `Conta` na memória. Compará-las com o operador "==" vai nos retornar `True`, pois o valor que elas carregam é o mesmo:

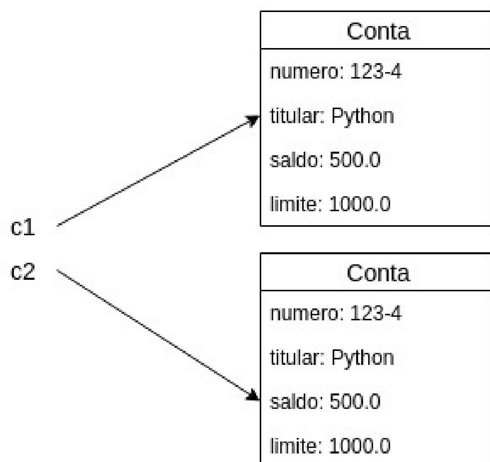
```
>>> id(c1) == id(c2)
True
>>> c1 == c2
True
```

Podemos então ver outra situação:

```
>>> c1 = Conta("123-4", "Python", 500.0, 1000.0)
>>> c2 = Conta("123-4", "Python", 500.0, 1000.0)
>>> if(c1 == c2):
...     print("contas iguais")
>>>
```

O operador "==" compara o conteúdo das variáveis, mas essas variáveis não guardam o objeto, e sim

o endereço em que ele se encontra. Como em cada uma dessas variáveis guardamos duas contas criadas diferentemente, elas estão em espaços diferentes da memória, o que faz o teste no `if` valer `False`. As contas podem ser equivalentes no nosso critério de igualdade, porém elas não são o mesmo objeto. Quando se trata de objetos, pode ficar mais fácil pensar que o "==" compara se os objetos (referências, na verdade) são o mesmo, e não se possuem valores iguais.



Para saber se dois objetos têm o mesmo conteúdo, você precisa comparar atributo por atributo. Futuramente, veremos uma solução mais elegante para isso também.

Agora é a melhor hora de aprender algo novo

Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

Conheça a Alura Cursos Online.

7.8 MÉTODO TRANSFERE

E a funcionalidade que transfere dinheiro entre duas contas? Podemos ficar tentados a criar um método que recebe dois parâmetros: `conta1` e `conta2` do tipo `Conta`. Cuidado: já sabemos que os métodos de nossa classe `Conta` sempre recebem a referência, o **self** - portanto o método recebe apenas um parâmetro do tipo `Conta`, a conta destino (além do valor):

```
class Conta:

    # código omitido

    def transfere(self, destino, valor):
        self.saldo -= valor
        destino.saldo += valor
```

Para deixar o código mais robusto, poderíamos verificar se a conta possui a quantidade a ser transferida disponível. Para ficar ainda mais interessante, você pode chamar os métodos `deposita` e `saca` já existentes para fazer essa tarefa:

```
class Conta:

    # código omitido

    def transfere(self, destino, valor):
        retirou = self.saca(valor)
        if (retirou == False):
            return False
        else:
            destino.deposita(valor)
            return True
```

Quando passamos uma `Conta` como argumento, o que será que acontece na memória? Será que o objeto é clonado?

No Python, a passagem de parâmetro funciona como uma simples atribuição como no uso do "=". Então, esse parâmetro vai copiar o valor da variável do tipo `Conta` que for passado como argumento para a variável `destino` . E qual é o valor de uma variável dessas? Seu valor é um endereço, uma referência, nunca um objeto. Por isso não há cópia de objetos aqui.

Esse último código poderia ser escrito com uma sintaxe muito sucinta. Como?

Transfere Para

Perceba que o nome deste método poderia ser `transfere_para()` ao invés de só `transfere()` . A chamada do método fica muito mais natural, é possível ler a frase em português que ela tem um sentido:

conta1.transfere_para(conta2, 50.0):

A leitura deste código seria "conta1 transfere para conta2 50 reais".

7.9 CONTINUANDO COM ATRIBUTOS

Os atributos de uma classe podem receber um valor padrão - assim como os argumentos de uma função. Nosso banco pode ter um valor de limite padrão para todas as contas e apenas em casos

específicos pode atribuir um valor limite diferente.

Para aplicarmos essa regra de negócio, podemos atribuir um valor padrão ao limite, por exemplo, 1000.0 reais:

```
class Conta:

    def __init__(self, numero, titular, saldo, limite=1000.0):
        self.numero = numero
        self.titular = titular
        self.saldo = saldo
        self.limite = limite
```

E podemos inicializar uma conta:

```
>>> conta = Conta('123-4', 'joão', 120.0)
```

Veja que agora não somos obrigados a passar o valor do `limite` já que ele possui um valor padrão de 1000.0 e podemos acessá-lo pela `conta` :

```
>>> conta.limite
1000.0
```

Quando declaramos as variáveis na classe `Conta` , aprendemos que podemos atribuir um valor padrão para cada uma delas. Imagine que comecemos a aumentar nossa classe `Conta` e adicionar nome, sobrenome e cpf do titular da conta. Começaríamos a ter muitos atributos... e, se você pensar direito, uma `Conta` não tem nome, nem sobrenome nem cpf, quem tem esses atributos é um **cliente**. Então podemos criar uma nova classe e fazer uma **agregação** - agregar um cliente a nossa conta. Portanto, nossa classe `Conta` **tem um** `Cliente` .

O atributos de uma `Conta` também podem ser referências para outras classes. Suponha a seguinte classe `Cliente` :

```
class Cliente:

    def __init__(self, nome, sobrenome, cpf):
        self.nome = nome
        self.sobrenome = sobrenome
        self.cpf = cpf

class Conta:

    def __init__(self, numero, cliente, saldo, limite):
        self.numero = numero
        self.titular = cliente
        self.saldo = saldo
        self.limite = limite
```

E quando criarmos um `Conta` , precisamos passar um `Cliente` como `titular` :

```
>>> from conta import Conta, Cliente
>>> cliente = Cliente('João', 'Oliveira', '1111111111-1')
>>> minha_conta = Conta('123-4', cliente, 120.0, 1000.0)
```

Aqui aconteceu uma atribuição, o valor da variável `cliente` é copiado para o atributo `titular` do

objeto ao qual `minha_conta` se refere. Em outras palavras, `minha_conta` tem uma referência ao mesmo `Cliente` que `cliente` se refere, e pode ser acessado através de `minha_conta.titular` .

Você pode realmente navegar sobre toda estrutura de informação, sempre usando o ponto:

```
>>> minha_conta.titular
<__main__.Cliente object at 0x7f83dac31dd8>
```

Veja que a saída é a referência a um objeto do tipo `Cliente` , mas podemos acessar seus atributos de uma forma mais direta e até mais elegante:

```
>>> minha_conta.titular.nome
'João'
```

7.10 TUDO É OBJETO

Python é uma linguagem totalmente orientada a objetos. Tudo em Python é um objeto! Sempre que utilizamos uma função ou método que recebe parâmetros estamos passando objetos como argumentos. Não é diferente com nossas classes. Quando uma conta recebe um cliente como titular, ele está recebendo uma instância de `Cliente` , ou seja, um objeto.

O mesmo acontece com `numero` , `saldo` e `limite` . *Strings* e números são classes no Python. Por este motivo que aparece a palavra **class** quando pedimos para o Python nos devolver o tipo de uma variável através da função **type**:

```
>>> type(conta.numero)
<class 'str'>
>>> type(conta.saldo)
<class 'float'>
>>> type(conta.titular)
<class '__conta__.Cliente'>
```

Um sistema orientado a objetos é um grande conjunto de classes que vai se comunicar, delegando responsabilidades para quem for mais apto a realizar determinada tarefa. A classe `Banco` usa a classe `Conta` que usa a classe `Cliente` , que usa a classe `Endereco` , etc... Dizemos que esses objetos colaboram, trocando mensagens entre si. Por isso acabamos tendo muitas classes em nosso sistema, e elas costumam ter um tamanho relativamente curto.

Editora Casa do Código com livros de uma forma diferente

Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.
Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

Casa do Código, ebook com preço de ebook.

7.11 COMPOSIÇÃO

Fizemos, no ponto anterior, uma **agregação**. Agora nossa classe `Conta` **tem um** `Cliente` e associamos estas duas classes. Mas nossa classe `Cliente` existe independente da classe `Conta` . Suponha agora que nossa `Conta` possua um histórico, contendo a data de abertura da conta e suas transações. Podemos criar uma classe para representar o histórico, como no exemplo abaixo:

```
import datetime

class Historico:

    def __init__(self):
        self.data_abertura = datetime.datetime.today()
        self.transacoes = []

    def imprime(self):
        print("data abertura: {}".format(self.data_abertura))
        print("transações: ")
        for t in self.transacoes:
            print("-", t)
```

Agora precisamos modificar nossa classe `Conta` de modo que ela tenha um `Historico` . Mas aqui, diferente da relação do cliente com uma conta, a existência de um histórico depende da existência de uma `Conta` :

```
class Conta:

    def __init__(self, numero, cliente, saldo, limite=1000.0):
        self.numero = numero
        self.cliente = cliente
        self.saldo = saldo
        self.limite = limite
        self.historico = Historico()
```

E podemos, em cada método para manipular uma `Conta` , acrescentar a operação nas transações de seu `Historico` :

```
class Conta:

    #código omitido

    def deposita(self, valor):
        self.saldo += valor
        self.historico.transacoes.append("depósito de {}".format(valor))

    def saca(self, valor):
        if (self.saldo < valor):
            return False
        else:
            self.saldo -= valor
            self.historico.transacoes.append("saque de {}".format(valor))

    def extrato(self):
        print("numero: {} \nsaldo: {}".format(self.numero, self.saldo))
        self.historico.transacoes.append("tirou extrato - saldo
        de {}".format(self.saldo))

    def transfere_para(self, destino, valor):
        retirou = self.saca(valor)
        if (retirou == False):
            return False
        else:
            destino.deposita(valor)
            self.historico.transacoes.append("transferencia de {}
            para conta {}".format(valor, destino.numero))
            return True
```

E testamos:

```
$python3.6
>>> from conta import Conta, Cliente
>>> cliente1 = Cliente('João', 'Oliveira', '11111111111-11')
>>> cliente2 = Cliente('José', 'Azevedo', '222222222-22')
>>> conta1 = Conta('123-4', cliente1, 1000.0)
>>> conta2 = Conta('123-5', cliente2, 1000.0)
>>> conta1.deposita(100.0)
>>> conta1.saca(50.0)
>>> conta1.transfere_para(conta2, 200.0)
>>> conta1.extrato
numero: 123-4
saldo: 850.0
>>> conta1.historico.imprime()
data abertura: 2018-05-10 19:44:07.406533
transações:
- depósito de 100.0
- saque de 50.0
- saque de 200.0
- transferencia de 200.0 para conta 123-5
- tirou extrato - saldo de 850.0
>>> conta2.historico.imprime()
data abertura: 2018-05-10 19:44:07.406553
transações:
- depósito de 200.0
```

Quando a existência de uma classe depende de outra classe, como é a relação da classe Histórico com a classe Conta, dizemos que a classe `Historico` compõe a classe `Conta` . Esta associação chamamos **Composição**.

Mas, e se dentro da nossa `Conta` não colocássemos `self.historico = Historico()` e tentasse acessá-lo diretamente? Faz algum sentido fazer `historico = Historico()`?

Quando o objeto é inicializado, ele vai receber o valor default que definimos na classe:

```
class Conta:

    def __init__(self, numero, cliente, saldo, limite):
        #iniciando outros parâmetros
        self.historico = Historico()
```

Com esse código, toda nova `Conta` criada já terá um novo `Historico` associado, sem necessidade de instanciá-lo logo em seguida da instanciação de uma `Conta` .

Atenção: para quem não está acostumado com referências, pode ser bastante confuso pensar sempre em como os objetos estão na memória para poder tirar as conclusões de o que ocorrerá ao executar determinado código, por mais simples que ele seja. Com o tempo, você adquire a habilidade de rapidamente saber o efeito de atrelar as referências, sem ter de gastar muito tempo para isso. É importante, nesse começo, você estar sempre pensando no estado da memória. E realmente lembrar que, no Python "uma variável nunca carrega um objeto, e sim uma referência para ele" facilita muito.

7.12 PARA SABER MAIS: OUTROS MÉTODOS DE UMA CLASSE

O interpretador adiciona alguns atributos especiais somente para leitura a vários tipos de objetos de uma classe e um deles é o `__dict__` .

Isso acontece porque a classe `Conta` possui alguns métodos, dentre eles o `__init__()` e o `__new__()` que são chamados para criar e inicializar um objeto desta classe, respectivamente. Caso você queira saber quais outros métodos são implementados pela classe `Conta` você pode usar a função embutida `dir()` que vai listar todos métodos e atributos que a classe possui.

```
>>> dir(Conta)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '_
_le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__se
tattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'extrato', 'deposita', 'limite'
, 'numero', 'saca', 'saldo', 'transfere_para', 'titular']
```

Dessa lista, já conhecemos o `__init__()` , o `__new__()` e os métodos e atributos que definimos quando construímos a classe `Conta` . Na verdade, quando usamos a função `dir()` , o interpretador chama o atributo `__dir__` dessa lista. Um outro atributo bastante útil é o `__dict__` que retorna um dicionário com os atributos da classe

```
>>> cliente = Cliente('João', 'Oliveira', '111111111-11')
>>> conta = Conta('123-4', cliente, 1000.0)
>>> conta.__dict__
{'saldo': 1000.0, 'numero': '123-4', 'titular': <__main__.Cliente object at 0x7f0b6d028f28>, 'limite'
: 1000.0}
```

Mas não é comum acessá-lo dessa maneira. Estes métodos iniciados e terminados com dois *underscores* são chamados pelo interpretador e são conhecidos como métodos mágicos. Existe outra função embutida do Python, a função `vars()` , que chama exatamente o `__dict__` de uma classe. Obtemos o mesmo resultado usando `vars(conta)` :

```
>>> vars(conta)
{'saldo': 1000.0, 'numero': '123-4', 'titular': <__main__.Cliente object at 0x7f0b6d028f28>, 'limite'
: 1000.0}
```

Repare que o `__dict__` e o `vars()` retornam exatamente um dicionário de atributos de uma conta como tínhamos modelado no início deste capítulo. Portanto, nossas classes utilizam dicionários para armazenar informações da própria classe.

Os demais métodos mágicos estão disponíveis para uso e não utilizaremos por enquanto. Voltaremos a falar deles em um outro momento.

7.13 EXERCÍCIO: PRIMEIRA CLASSE PYTHON

1. Crie um arquivo chamado `conta.py` na pasta `oo` criada no exercício anterior.

2. Crie a classe Conta sem nenhum atributo e salve o arquivo.

```
class Conta:
    pass
```

3. Abra o terminal e vá até a pasta onde se encontra o arquivo `conta.py` . Abra o console do Python3 no terminal e importe a classe Conta do módulo conta.

```
>>> from conta import Conta
```

4. Crie uma instância (objeto) da classe `Conta` e utilize a função `type()` para verificar o tipo do objeto:

```
>>> conta = Conta()
>>> type(conta)
<class 'conta.Conta'>
```

Além disso, crie alguns atributos e tente acessá-los.

1. Abra novamente o arquivo conta.py e escreva o método `__init__()` recebendo os atributos anteriormente definidos por nós que toda conta deve ter (numero titular, saldo e limite):

```
class Conta:

    def __init__(self, numero, titular, saldo, limite):
        self.numero = numero
        self.titular = titular
        self.saldo = saldo
        self.limite = limite
```

2. Reinicie o Python3 no terminal e importe novamente a classe `Conta` do módulo conta para

testarmos nosso código:

```
>>> from conta import Conta
```

3. Tente criar uma conta sem passar qualquer argumento no construtor:

```
>>> conta = Conta()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __init__() missing 4 required positional arguments: 'numero', 'titular', 'saldo', and
'limite
```

   Note que o interpretador acusou um erro. O método `__init__()` exige 4 argumentos 'numero', 'titular', 'saldo' e 'limite'.

4. Agora vamos seguir o exigido pela classe, pela receita de uma conta:

```
>>> conta = Conta('123-4', 'João', 120.0, 1000.0)
```

5. O interpretador não acusou nenhum erro. Vamos imprimir o `numero` e `titular` da conta:

```
>>> conta.numero
'123-4'
>>> conta.titular
'João'
```

6. Crie o método `deposita()` dentro da classe `Conta`. Esse método deve receber uma referência do próprio objeto e o valor a ser adicionado ao saldo da conta.

```
def deposita(self, valor):
    self.saldo += valor
```

7. Crie o método `saca()` que recebe como argumento uma referência do próprio objeto e o valor a ser sacado. Esse método subtrairá o valor do saldo da conta.

```
def saca(self, valor):
    self.saldo -= valor
```

8. Crie o método `extrato()`, que recebe como argumento uma referência do próprio objeto. Esse método imprimirá o saldo da conta:

```
def extrato(self):
    print("numero: {} \nsaldo: {}".format(self.numero, self.saldo))
```

9. Modifique o método `saca() fazendo retornar um valor que representa se a operação foi ou não bem sucedida. Lembre que não é permitido sacar um valor menor do que o saldo.

```
def saca(self, valor):
    if (self.saldo < valor):
        return False
    else:
        self.saldo -= valor
        return True
```

10. Crie o método `transfere_para()` que recebe como argumento uma referência do próprio objeto, uma `Conta` destino e o valor a ser transferido. Esse método deve sacar o valor do próprio objeto e

depositar na conta destino:

```
def transfere_para(self, destino, valor):
    retirou = self.saca(valor)
    if (retirou == false):
        return False
    else:
        destino.deposita(valor)
        return True
```

11. Abra o Python no terminal, importe o módulo conta, crie duas contas e teste os métodos criados.

12. (Opcional) Crie uma classe para representar um cliente do nosso banco que deve ter `nome` , `sobrenome` e `cpf` . Instancie uma `Conta` e passe um cliente como `titular` da conta. Modifique o método `extrato()` da classe `Conta` para imprimir, além do número e o saldo, os dados do cliente. Podemos criar uma `Conta` sem um `Cliente` ? E um `Cliente` sem uma `Conta` ?

13. (Opcional) Crie uma classe que represente uma data, com dia, mês e ano. Crie um atributo `data_abertura` na classe `Conta` . Crie uma nova conta e faça testes no console do Python.

14. (Desafio) Crie uma classe `Historico` que represente o histórico de uma `Conta` seguindo o exemplo da apostila. Faça testes no console do Python criando algumas contas, fazendo operações e por último mostrando o histórico de transações de uma `Conta` . Faz sentido criar um objeto do tipo `Historico` sem uma `Conta?`

Agora, além de funcionar como esperado, nosso código não permite criar uma conta sem os atributos que definimos anteriormente. Discuta com seus colegas e instrutor as vantagens da orientação a objetos até aqui.

Já conhece os cursos online Alura?

A **Alura** oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex-aluno da Caelum tem 15% de desconto neste link!

Conheça os cursos online Alura.

Capítulo 8

MODIFICADORES DE ACESSO E MÉTODOS DE CLASSE

Um dos problemas mais simples que temos no nosso sistema de contas é que o método `saca()` permite sacar mesmo que o saldo seja insuficiente. A seguir você pode lembrar como está a classe `Conta` :

```
class Conta:

    def __init__(self, numero, titular, saldo, limite=1000.0):
        self.numero = numero
        self.titular = titular
        self.saldo = saldo
        self.limite = limite

    # outros métodos

    def saca(self, valor):
        this.saldo -= valor
```

Abrimos o terminal e testamos nosso código:

```
minha_conta = Conta('123-4', 'joão', 1000.0, 2000.0)
minha_conta.saca(500000)
```

O limite de saque é ultrapassado. Podemos incluir um `if` dentro do método `saca()` para evitar a situação que resultaria em uma conta em estado inconsistente, com seu saldo menor do que zero. Fizemos isso no capítulo de orientação a objetos básica.

Apesar de melhorar bastante, ainda temos um problema mais grave: ninguém garante que o usuário da classe vai sempre utilizar o método para alterar o saldo da conta. O código a seguir altera o saldo diretamente:

```
minha_conta = Conta('123-4', 'João', 1000.0)
minha_conta.saldo = -200
```

Como evitar isso? Uma ideia simples seria testar se não estamos sacando um valor maior que o saldo toda vez que formos alterá-lo.

```
minha_conta = Conta('123-4', 'joão', 1000.0)
novo_saldo = -200

if(novo_saldo < 0):
    print("saldo inválido")
else:
    minha_conta.saldo = novo_saldo
```

Esse código iria se repetir ao longo de toda nossa aplicação e, pior, alguém pode esquecer de fazer essa comparação em algum momento, deixando a conta em uma situação inconsistente. A melhor forma de resolver isso seria forçar quem usa a classe `Conta` a invocar o método `saca()` e não permitir o acesso direto ao atributo.

Em linguagens como Java e C# basta declarar que os atributos não podem ser acessados de fora da classe utilizando a palavra chave **private**. Em orientação a objetos, é prática quase que obrigatória proteger seus atributos com **private**. Cada classe é responsável por controlar seus atributos, portanto ela deve julgar se aquele novo valor é válido ou não. E esta validação não deve ser controlada por quem está usando a classe e sim por ela mesma, centralizando essa responsabilidade e facilitando futuras mudanças no sistema.

O Python não utiliza o termo **private**, que é um **modificador de acesso** e também chamado de **modificador de visibilidade**. No Python inserimos dois *underscores* ('__') ao atributo para adicionar esta característica:

```
class Pessoa:

    def __init__(self, idade):
        self.__idade = idade
```

Dessa maneira não conseguimos acessar o atributo `idade` de um objeto do tipo `Pessoa` fora da classe:

```
>>> pessoa = Pessoa(20)
>>> pessoa.idade
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Pessoa' object has no attribute 'idade'
```

O interpretador acusa que o atributo `idade` não existe na classe `Pessoa`. Mas isso não garante que ninguém possa acessá-lo. No Python não existem atributos realmente privados, ele apenas alerta que você não deveria tentar acessar este atributo, ou modificá-lo. Para acessá-lo, fazemos:

```
>>> p._Pessoa__idade
```

Ao colocar o prefixo __ no atributo da classe, o Python apenas renomeia '**__nome_do_atributo**' para '**_nome*da_Classe*_nome_do_atributo**', como fez em **__idade** para **_Pessoa__idade**. Qualquer pessoa que saiba que os atributos privados não são realmente privados, mas "desconfigurados", pode ler e atribuir um valor ao atributo "privado" diretamente. Mas fazer `pessoa._Pessoa__idade = 20` é considerado má prática e pode acarretar em erros.

Podemos utilizar a função `dir` para ver que o atributo `_Pessoa__idade` pertence ao objeto:

```
>>> dir(pessoa)
['_Pessoa__idade', '__class__', '__delattr__', '__dict__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__le__', '__lt__',
'__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
```

```
'__weakref__']
```

Repare que não existe nenhum atributo `__idade` no objeto pessoa. Agora vamos tentar atribuir um valor para `__idade` :

```
>>> pessoa.__idade = 25
```

Epa, será que o Python deveria deixar isso ocorrer? Vamos acessar a variável novamente e ver se a modificação realmente aconteceu:

```
>>> pessoa._Pessoa__idade
20
```

O que aconteceu aqui é que o Python criou um novo atributo **__idade** para o objeto `pessoa` já que é uma linguagem dinâmica. Vamos utilizar a função `dir` novamente para ver isso:

```
>>> dir(pessoa)
['_Pessoa__idade', '__class__', '__delattr__', '__dict__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__idade', '__init__', '__init_subclass__', '__le__', '__lt_'
'__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__weakref__']
```

Note que um novo atributo `__idade` apareceu, já que foi inicializado em tempo de execução e é diferente do `__idade` da classe. Isso pode gerar muita confusão e erros! O Python também tem uma maneira de lidar com este problema através da variável **__slots__** onde definimos um número limitado de atributos que veremos a seguir.

Nenhum atributo é realmente privado em Python já que podemos acessá-lo pelo seu nome 'desfigurado'. Muitos programadores Python não gostam dessa sintaxe e preferem usar apenas um *underscore* '_' para indicar quando um atributo deve ser protegido. Ou seja, deve ser explícita essa desconfiguração do nome - feita pelo programador e não pelo interpretador - já que oferece o mesmo resultado. E argumentam que '__' são obscuros.

O prefixo com apenas um *underscore* não tem significado para o interpretador quando usado em nome de atributos, mas entre programadores Python é uma convenção que deve ser respeitada. O programador alerta que esse atributo não deve ser acessado diretamente:

```
def __init__(self, idade):
    self._idade = idade
```

Um atributo com apenas um underscore é chamados de protegido, mas quando usado **sinaliza** que deve ser tratado como um atributo "privado" e acessá-lo diretamente pode ser perigoso.

As mesmas regras de acesso aos atributos valem para os métodos. É muito comum, e faz todo sentido, que seus atributos sejam privados e quase todos seus métodos sejam públicos (não é uma regra!). Desta forma, toda conversa de um objeto com outro é feita por troca de mensagens, isto é, acessando seus métodos. Algo muito mais educado que mexer diretamente em um atributo que não é

seu.

Melhor ainda! O dia que precisarmos mudar como é realizado um saque na nossa classe `Conta`, adivinhe onde precisaríamos modificar? Apenas no método `saca()`, o que faz pleno sentido.

8.1 ENCAPSULAMENTO

O que começamos a ver nesse capítulo é a ideia de encapsular, isto é, 'esconder' todos os membros de uma classe (como vimos acima), além de esconder como funcionam as rotinas (no caso métodos) do nosso sistema.

Encapsular é fundamental para que seu sistema seja suscetível a mudanças: não precisamos mudar uma regra de negócio em vários lugares, mas sim em apenas um único lugar, já que essa regra está encapsulada. O conjunto de métodos públicos de uma classe é também chamado de **interface da classe**, pois esta é a única maneira a qual você se comunica com objetos dessa classe.

O *underscore* _ alerta que ninguém deve modificar, nem mesmo ler, o atributo em questão. Com isso, temos um problema: como fazer para mostrar o saldo de uma `Conta`, já que não devemos acessá-lo para leitura diretamente?

Precisamos então arranjar uma maneira de fazer esse acesso. Sempre que precisamos arrumar uma maneira de fazer alguma coisa com um objeto, utilizamos métodos! Vamos então criar um método, digamos `pega_saldo()`, para realizar essa simples tarefa:

```
class Conta:

    # outros métodos

    def pega_saldo(self):
        return self._saldo
```

Para acessarmos o saldo de uma conta, podemos fazer:

```
>>> minha_conta = Conta('123-4', 'joão', 1000.0)
>>> minha_conta.deposita(100)
>>> minha_conta.pega_saldo()
1100
```

Para permitir o acesso aos atributos (já que eles são 'protegidos') de uma maneira controlada, a prática mais comum é criar dois métodos, um que retorna o valor e outro que muda o valor. A convenção para esses métodos em muitas linguagens orientadas a objetos é colocar a palavra **get** ou **set** antes do nome do atributo. Por exemplo, uma conta com saldo e titular fica assim, no caso de desejarmos dar acesso a leitura e escrita a todos os atributos:

```
class Conta:

    def __init__(self, titular, saldo):
        self._titular = titular
        self._saldo = saldo
```

```
    def get_saldo(self):
        return self._saldo

    def set_saldo(self, saldo):
        self._saldo = saldo

    def get_titular(self):
        return self._titular

    def set_titular(self, titular):
        self._titular = titular
```

Getters e *setters* são usados em muitas linguagens de programação orientada a objetos para garantir o princípio do encapsulamento de dados. O encapsulamento de dados é visto como o agrupamento de dados com os métodos que operam nesses dados. Esses métodos são, obviamente, o *getter* para recuperar os dados e o *setter* para alterar os dados. De acordo com esse princípio, os atributos de uma classe são tornados privados para ocultá-los e protegê-los de outro código.

Infelizmente, é crença generalizada que uma classe Python adequada deve encapsular atributos privados usando *getters* e *setters*. Assim que um desses programadores introduzir um novo atributo, ele fará com que seja uma variável privada e criará "automaticamente" um *getter* e um *setter* para esses atributos.

Os programadores de Java irão torcer o nariz quando lerem o seguinte: A maneira *pythônica* de introduzir atributos é torná-los públicos. Vamos explicar isso mais tarde. Primeiro, demonstramos no exemplo a seguir, como podemos projetar uma classe, da mesma maneira usada no Java, com *getters* e *setters* para encapsular um atributo protegido:

```
class Conta:

    def __init __(self, saldo):
        self._saldo = saldo

    def get_saldo(self):
        retorno self._saldo

    def set_saldo(self, saldo):
        self._saldo = saldo
```

E podemos ver como trabalhar com essa classe e os métodos:

```
>>> conta1 = Conta(200.0)
>>> conta2 = Conta(300.0)
>>> conta3 = Conta(-100.0)
>>> conta1.get_saldo()
200.0
>>> conta2.get_saldo()
300.0
>>> conta3.set_saldo(conta1.get_saldo() + conta2.get_saldo())
>>> conta3.get_saldo()
500.0
```

O que você acha da expressão "conta3.set_saldo(conta1.get_saldo() + conta2.get_saldo())"? É feio,

não é? É muito mais fácil escrever uma expressão como a seguinte:

```
conta3.saldo = conta1.saldo + conta2.saldo
```

Tal atribuição é mais fácil de escrever e, acima de tudo, mais fácil de ler do que a expressão com *getters* e *setters*. Vamos reescrever a classe `Conta` de um modo Pythônico, sem *getter* e sem *setter*:

```
class Conta:

    def __init __(self, saldo):
        self.saldo = saldo
```

Mas neste caso não há encapsulamento e não seria um problema. Mas o que acontece se quisermos mudar a implementação no futuro? O leitor atento deve ter reparado que no exemplo anterior declaramos uma variável do tipo `Conta` com saldo negativo e isso não deveria acontecer. Temos que evitar essa situação e o *setter*, neste caso, se justifica para acrescentar esta validação:

```
class Conta:

    def __init __(self, saldo):
        self.saldo = saldo

    def set_saldo(self, saldo):
        if(saldo < 0):
            print("saldo não pode ser negativo")
        else:
            self.saldo = saldo
```

Podemos abrir o interpretador e testar:

```
>>> conta1 = Conta(200.0)
>>> conta1.saldo
200.0
>>> conta2 = Conta(300.0)
>>> conta2.saldo
300.0
>>> conta3 = Conta(100.0)
>>> conta3.set_saldo(-100.0)
"saldo não pode ser negativo"
>>> conta3.saldo
100.0
```

Mas há um problema, caso projetemos nossa classe com atributo público e sem métodos quebramos a interface:

```
conta1 = Conta(100.0)
conta.saldo = -100.0
```

É por isso que em Java recomenda-se que as pessoas usem somente atributos privados com *getters* e *setters*, para que possam alterar a implementação sem precisar alterar a interface. O Python oferece uma solução bastante parecida para este problema. A solução é chamada de **properties**. Mantemos nossos atributos protegidos e decoramos nossos métodos com um *decorator* chamado *property*.

A classe com uma propriedade fica assim:

```
class Conta:

    def __init__(self, saldo=0.0):
        self._saldo = saldo

    @property
    def saldo(self):
        return self._saldo

    @saldo.setter
    def saldo(self, saldo):
        if(self._saldo < 0):
            print("saldo não pode ser negativo")
        else:
            self._saldo = saldo
```

Um método que é usado para obter um valor (o *getter*) é decorado com `@property`, isto é, colocamos essa linha diretamente acima da declaração do método que recebe o nome do próprio atributo. O método que tem que funcionar como *setter* é decorado com `@saldo.setter`. Se a função tivesse sido chamada de "func", teríamos que anotá-la com `@func.setter`.

PARA SABER MAIS: DECORATOR

Um decorador, ou **decorator** é um padrão de projeto de software que permite adicionar um comportamento a um objeto já existente em tempo de execução, ou seja, agrega dinamicamente responsabilidades adicionais a um objeto. Esta solução traz uma flexibilidade maior, em que podemos adicionar ou remover responsabilidades sem que seja necessário editar o código-fonte.

Um decorador é um objeto invocável, uma função que aceita outra função como parâmetro (a função decorada). O decorador pode realizar algum processamento com a função decorada e devolvê-la ou substituí-la por outra função. O *property* é um decorador que possui métodos extras como um *getter* e um *setter* e ao ser aplicado a um objeto, retorna uma cópia dele com essas funcionalidades:

```
@property
def foo(self):
   return self._foo
```

é equivalente a:

```
def foo(self)
   return self._foo

foo = property(foo)
```

Portanto, a função `foo()` é substituída pela propriedade `property(foo)`. Então, se você usa `@foo.setter()`, o que você está fazendo é chamar o método `property().setter`

Desta maneira, podemos chamar esses métodos sem os parênteses, como se fossem atributos

públicos. É uma forma mais elegante de encapsular nossos atributos. Vamos testar criar uma conta e depois atribuir um valor negativo ao saldo:

```
>>> conta = Conta(1000.0)
>>> conta.saldo = -300.0
"saldo não pode ser negativo"
```

Veja que temos um resultado muito melhor do que usar *getters* and *setters* diretamente. Chamamos o atributo pela suas propriedades, que podem conter validações, e nossos atributos estão sinalizados como 'protegidos' através do '_'.

Mas ainda podemos modificar o saldo e isto deveria ser feito através dos métodos públicos `saca()` e `deposita()` . Então, a necessidade de um `@saldo.setter` é questionável. Devemos apenas manipular o saldo através dos métodos `saca()` e `deposita()` , não precisamos da *property* `saldo.setter` . Isso é uma decisão de negócio específico. O programador deve ficar alerta quanto as propriedades *setters* de seus atributos, nem sempre elas são necessárias.

É uma má prática criar uma classe e, logo em seguida, criar as propriedades para todos seus atributos. Você só deve criar *properties* se tiver real necessidade. Repare que nesse exemplo, a propriedade *setter* do saldo não deveria ter sido criada já que queremos que todos usem os métodos `deposita()` e `saca()` .

Você pode também fazer o curso PY-14 dessa apostila na Caelum

Querendo aprender ainda mais sobre? Esclarecer dúvidas dos exercícios? Ouvir explicações detalhadas com um instrutor?
A Caelum oferece o **curso PY-14** presencial nas cidades de São Paulo, Rio de Janeiro e Brasília, além de turmas incompany.

Consulte as vantagens do curso *Python e Orientação a Objetos*

8.2 ATRIBUTOS DE CLASSE

Nosso banco também quer controlar a quantidade de contas existentes no sistema. Como poderíamos fazer isso? Bom, a cada instância criada deveríamos incrementar esse total:

```
total_contas = 0
conta = Conta(300.0)
total_contas = total_contas + 1
conta2 = Conta(100.0)
total_contas = total_contas + 1
```

Aqui Volta o problema de repetir um mesmo código para toda aplicação, além de ter que lembrar de incrementar a variável `total_contas` toda vez após instanciar uma `Conta`. Como `total_contas` tem vínculo com a classe `Conta`, ele deve ser um atributo controlado pela **classe** que deve incrementá-lo toda vez que instanciamos um objeto, ou seja, quando chamamos o método `__init__()`:

```
class Conta:

    def __init__(self, saldo):
        self._saldo = saldo
        self._total_contas = self._total_contas + 1
```

Mas onde inicializamos a variável `_total_contas`? Não faz sentido recebermos por parâmetro no `__init__()` já que é a classe que deve controlar esse número e não o objeto. Seria interessante que essa variável fosse própria da classe, fosse única e compartilhada por todos os objetos dessa classe. Dessa maneira, quando mudasse através de um objeto, o outro enxergaria o mesmo valor. Para fazer isso, vamos inicializar a variável na classe, portanto, fora do método `__init__()`:

```
class Conta:

    total_contas = 0

    def __init__(self, saldo):
        self._saldo = saldo
        self.total_contas += 1
```

Veja que `saldo` é um **atributo de instância** e `total_contas` um **atributo de classe**. Vamos fazer um teste para ver se nosso `total_contas` funciona como esperado:

```
>>> c1 = Conta(100.0)
>>> c1.total_contas
1
>>> c2 = Conta(200.0)
>>> c2.total_contas
1
```

Criamos duas instâncias e mesmo assim o `total_contas` não mudou. Isso acontece por conta do `self.total_contas += 1`. `self.total_contas` é diferente de `total_contas` da classe. Como `total_contas` é uma variável da classe, devemos chamá-la pela classe:

```
class Conta:

    total_contas = 0

    def __init__(self, saldo):
        self._saldo = saldo
        Conta.total_contas += 1
```

E testamos:

```
>>> c1 = Conta(100.0)
>>> c1.total_contas
1
>>> c2 = Conta(200.0)
>>> c2.total_contas
2
```

Agora obtemos o resultado esperado. Também é possível acessar este atributo direto da classe:

```
>>> Conta.total_contas
2
```

Mas não queremos que ninguém venha a acessar nosso atributo `total_contas` e modificá-lo. Portanto vamos torná-lo 'protegido' acrescentando um '_':

```
class Conta:

    _total_contas = 0
```

Dessa maneira avisamos os usuários de nossa classe que esse atributo deve ser considerado 'privado' e não modificado. Mas como acessá-lo então? Veja que agora, ao acessar pela classe obtemos um erro:

```
>>> Conta.total_contas
Traceback (most recent call last):
  File <stdin>, line 23, in <module>
    Conta.total_contas
AttributeError: 'Conta' object has no attribute 'total_contas'
```

Precisamos criar um método para acessar o atributo. Vamos criar o `get_total_contas` :

```
class Conta:

    _total_contas = 0

    # __init__ e outros métodos

    def get_total_contas(self):
        return Conta._total_contas
```

Funciona quando chamamos este método por um instância, mas quando fazemos `Conta.get_total_contas()` o interpretador reclama pois não passamos a instância:

```
>>> c1 = Conta(100.0)
>>> c1.get_total_contas()
1
>>> c2 = Conta(200.0)
>>> c2.get_total_contas()
2
>>> Conta.get_total_contas()
Traceback (most recent call last):
  File <stdin>, line 17, in <module>
    Conta.get_total_contas()
TypeError: get_total_contas() missing 1 required positional argument: 'self'
```

Veja que o erro avisa que falta passar o argumento `self` . Não podemos chamá-lo pois não está vinculado a qualquer instância de `Conta` . E um método quer uma instância como seu primeiro argumento:

```
>>> c1 = Conta(100.0)
>>> c2 = Conta(200.0)
>>> Conta.get_total_contas(c1)
2
```

Passamos a instância `c1` de `Conta` e funcionou. Mas essa não é a melhor maneira de se chamar um

método. A chamada não é clara e leva um tempo para ler e entender o que a terceira linha desse código realmente faz. Vamos então deixar de passar o 'self' como argumento de `get_total_contas` :

```
def get_total_contas():
    return Conta._total_contas
```

Mas dessa maneira não conseguimos acessar o método já que todo método exige o argumento `self` :

```
>>> c1 = Conta(100.0)
>>> c1.get_total_contas()
Traceback (most recent call last):
  File <stdin> in <module>
    c1.get_total_contas()
TypeError: get_total_contas() takes 0 positional arguments but 1 was given
```

E agora, o que fazer? Queremos um método que seja chamado via classe e via instância sem a necessidade de passar a referência deste objeto. O Python resolve isso usando **métodos estáticos**.

Métodos estáticos não precisam de uma referência, não recebem um primeiro argumento especial (self). É como uma função simples que, por acaso, reside no corpo de uma classe em vez de ser definida no nível do módulo.

Para que um método seja considerado estático basta adicionarmos um decorador, como fizemos com as propriedades no capítulo anterior. O decorador se chama *@staticmethod*:

```
@staticmethod
def get_total_contas():
    return Conta._total_contas
```

Testando, vemos que funciona tanto chamado por um instância quanto pela classe:

```
>>> c1 = Conta(100.0)
>>> c1.get_total_contas()
1
>>> c2 = Conta(200.0)
>>> c2.get_total_contas()
2
>>> Conta.get_total_contas()
2
```

8.3 MÉTODOS DE CLASSE

Métodos estáticos não devem ser confundidos com métodos de classe. Como os métodos estáticos, métodos de classe não são ligados às instâncias, mas sim a classe. O primeiro parâmetro de um método de classe é uma referência para a classe, isto é, um objeto do tipo *class* que por convenção nomeamos como 'cls'. Eles podem ser chamados via instância ou pela classe e utilizam um outro decorar, o *@classmethod*:

```
class Conta:

    _total_contas = 0
```

```
    def __init__(self):
        type(self)._total_contas += 1

    @classmethod
    def get_total_contas(cls):
        return cls._total_contas
```

E podemos testar:

```
>>> c1 = Conta(100.0)
>>> c1.get_total_contas()
1
>>> c2 = Conta(200.0)
>>> c2.get_total_contas()
2
>>> Conta.get_total_contas()
2
```

No início pode parecer confuso qual usar: `@staticmethod` ou `@classmethod` ? Isso não é trivial. Métodos de classe servem para definir um método que opera na classe, e não em instâncias. Já os métodos estáticos utilizamos quando não precisamos receber a referência de um objeto especial (seja da classe ou de uma instância) e funciona como uma função comum, sem relação.

Isso ficará mais claro quando avançarmos no aprendizado. No próximo capítulo discutiremos Herança, um conceito fundamental em Orientação a Objetos. Veremos que classes podem ter filhas e aproveitar o código das classes mães. Um método de classe pode mudar a implementação, ou seja, pode ser reescrito pela classe filha. Já os métodos estáticos não podem ser reescritos pelas filhas, já que são imutáveis e não dependem de um referência especial.

@CLASSMETHOD X @STATICMETHOD

Alguns programadores não veem muito sentido em usar métodos estáticos, já que se você escrever uma função que não vai interagir com a classe, basta defini-la no módulo. Outros já contra argumentam em outra via, considerando herança de classes que veremos em outro capítulo. Indicamos a leitura do artigo '*The Definitive Guide on How to Use Static, Class and Abstract Methods in Python*' de Julien Danjou que pode ser acessado pelo link: https://julien.danjou.info/guide-python-static-class-abstract-methods/ .

8.4 PARA SABER MAIS - SLOTS

Aprendemos sobre encapsulamento e vimos que é uma boa prática proteger nossos atributos incluindo o prefixo *underscore* em seus nomes, seguindo a convenção utilizada pelos programadores. Além disso, utilizamos *properties* para acessar e modificar nossos atributos. Mas como Python é uma linguagem dinâmica, nada impede que usuários de nossa classe `Conta` criem atributos em tempo de

execução, fazendo, por exemplo:

```
>>> conta.nome = "minha conta"
```

Esse código não acusa erro e nossa conta fica aberta a modificações ferindo a segurança da classe. Para evitar isso podemos utilizar uma variável embutida no Python chamada `__slots__` que pode guardar uma lista de atributos da classe definidos por nós:

```
class Conta:

    __slots__ = ['_numero', '_titular', '_saldo', '_limite']

    def __init__(self, numero, titular, saldo, limite=1000.0):
        # inicialização dos atributos

    # código omitido
```

Agora, quando tentamos adicionar um atributo na classe recebemos um erro:

```
>>> conta.nome = "minha_conta"
Traceback (most recent call last):
    File <stdin>, line 1, in <module>
AttributeError: 'Conta' object has no attribute '__dict__'
```

```
class Conta:

    __slots__ = ['_numero', '_titular', '_saldo', '_limite']

    def __init__(self, numero, titular, saldo, limite=1000.0):
        self.numero = numero
        self.titular = titular
        self.saldo = saldo
        self.limite = limite

    # restante do código

conta.nome = "minha_conta"
```

Repare que o erro acusa que a classe `Conta` não possui o atributo `__dict__` . Ao atribuir um valor para `__slots__` , o interpretador do Python vai entender que queremos excluir o `__dict__` da classe `Conta` não sendo possível criar atributos, ou seja, impossibilitando adicionar atributos ao dicionário da classe que é responsável por armazenar atributos de instância. Portanto, tentar chamar `vars(conta)` também vai gerar um erro:

```
>>> vars(conta)
Traceback (most recent call last):
  File <stdin>, line 1, in <module>
TypeError: vars() argument must have __dict__ attribute
```

Embora `__slots__` seja muito utilizado para não permitir que usuários de nossas classes criem outros atributos, essa não é sua principal função nem o motivo de sua existência. O que acontece é que o `__dict__` desperdiça muita memória. Imagine um sistema grande, com milhões de instâncias de `Conta` - teríamos, consequentemente, milhões de dicionários de classe armazenando seus atributos de instância. O Python não pode simplesmente alocar uma quantidade estática de memória na criação de objetos para armazenar todos os atributos. Por isso, consome muita memória RAM se você criar muitos

objetos.

Para contornar este problema é que se usa o `__slots__` e este é seu principal propósito. O `__slots__` avisa o Python para não usar um dicionário e apenas alocar espaço para um conjunto fixo de atributos.

Programadores viram uma redução de quase 40 a 50% no uso de RAM usando essa técnica.

Seus livros de tecnologia parecem do século passado?

Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.
Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

Casa do Código, Livros de Tecnologia.

8.5 EXERCÍCIOS:

1. Adicione o modificador de visibilidade privado (dois *underscores*: `__`) para cada atributo e método da sua classe `Conta` . Tente criar uma `Conta` e modificar ou ler um de seus atributos "privados". O que acontece?

2. Sabendo que no Python não existem atributos privados, como podemos modificar e ler esses atributos? É uma boa prática fazer isso?

3. Modifique o acesso para 'protegido' seguindo a convenção do Python e modifique o prefixo `__` por apenas um *underscore* `_` . Crie métodos de acesso em sua classe `Conta` através do *decorator* `@property` .

4. Crie novamente uma conta e acesse e modifique seus atributos. O que mudou?

5. Modifique sua classe `Conta` de modo que não seja permitido criar outros atributos além dos definidos anteriormente utilizando `__slots__` .

6. (Opcional) Adicione um atributo `identificador` na classe `Conta` . Esse identificador deve ter um valor único para cada instância do tipo `Conta` . A primeira Conta instanciada tem identificador 1, a segunda 2, e assim por diante. Você deve utilizar os recursos aprendidos aqui para resolver esse problema.

CAPÍTULO 9

PYCHARM

9.1 IDE

Quando começamos a programar, uma das principais dúvidas de iniciantes é: "qual ferramenta vou utilizar para escrever código?". A maioria dos códigos das principais linguagens de programação permitem desenvolver em um arquivo utilizando um editor de texto comum.

Alguns editores de texto possuem ferramentas mais sofisticadas que dão maior auxílio na hora de desenvolver como: indentação de código, diferenciação de funções, autocompletamento de código, dentre outras.

Outra ferramenta, mais utilizada para desenvolver código, é o que chamamos de Ambiente Integrado de Desenvolvimento ou IDE (sigla em inglês para *Integrated Development Enviroment*). Uma IDE é um software com muitas funcionalidades que auxiliam no desenvolvimento de código além de possuir a capacidade de rodar o código.

Neste capítulo apresentaremos a IDE Pycharm e suas principais ferramentas.

Seus livros de tecnologia parecem do século passado?

Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.
Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

Casa do Código, Livros de Tecnologia.

9.2 PYCHARM

O Pycharm é multiplataforma com versões para Windows, MacOS e Linux. O PyCharm é desenvolvido pela empresa JetBrains e fornece análise de código, depurador gráfico, autocompletamento

de código e capacidades de navegação que facilitam a escrita de código.

IDE's foram desenvolvidas para criar código mais rapidamente e com maior eficiência. Veremos aqui os principais recursos do PyCharm. Você perceberá que ele evita ao máximo atrapalhar e apenas gera trechos de códigos óbvios, sempre ao seu comando.

No site oficial há guias e tutoriais para iniciantes. Se você se interessar, recomendamos usar o guia inicial neste link: https://www.jetbrains.com/help/pycharm/meet-pycharm.html

Com o PyCharm você pode desenvolver em Python. A versão Profissional dá suporte para desenvolvimento de aplicações web com Django, Flask e Pyramid. O Pycharm também suporta HTML, CSS, JavaScript e XML. Suporte para outras linguagens também podem ser adicionadas baixando plugins.

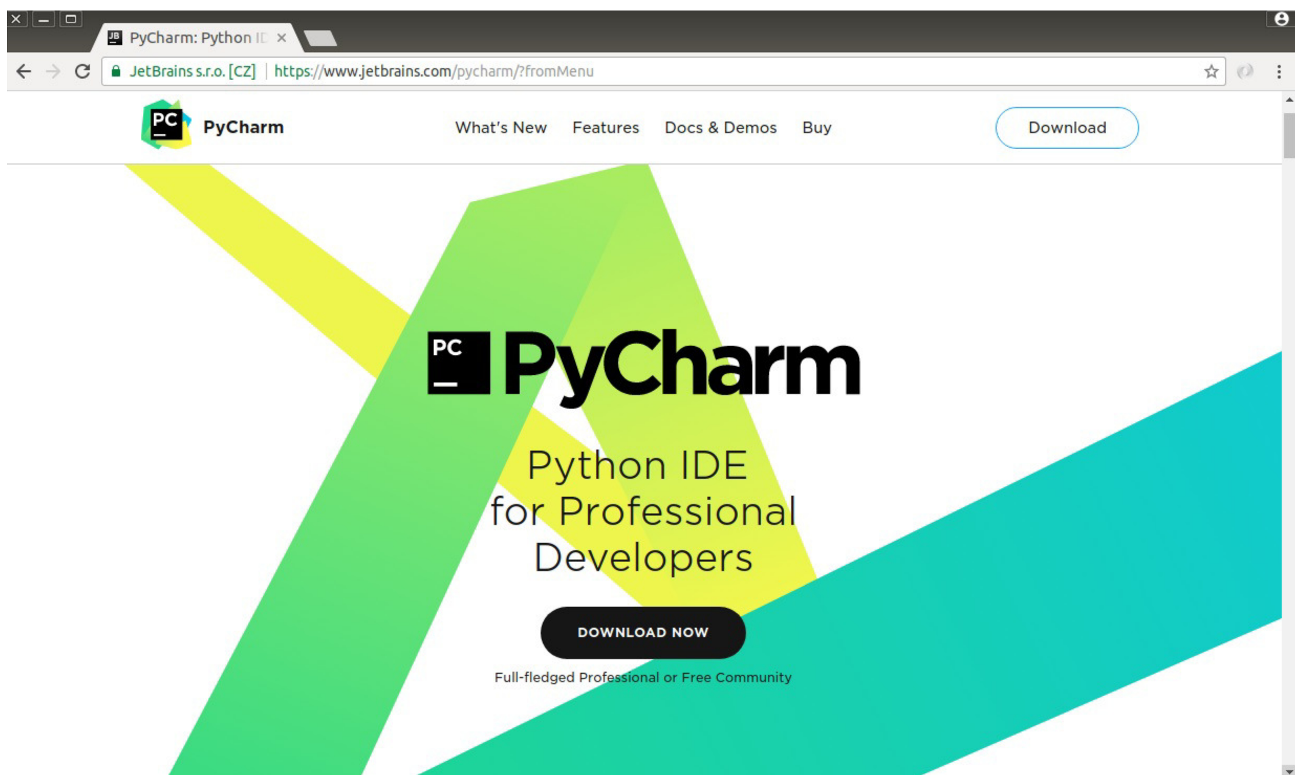
Outras IDEs

Outra IDE famosa no Python é o IDLE que possui bem menos recursos do que o PyCharm mas também é bastante utilizado pela comunidade.

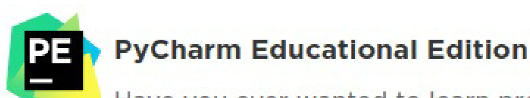
No site oficial da Python Brasil existem uma lista imensa de outras IDEs: https://wiki.python.org.br/IdesPython.

9.3 DOWNLOAD E INSTALAÇÃO DO PYCHARM

Se você ainda não possui o PyCharm, faça o download nesta página: https://www.jetbrains.com/pycharm/ .



Existem duas versões, a **Professional** e a **Commmunity**. A versão paga (a *Professional*) possui funcionalidades extras como suporte para desenvolvimento de aplicações web e integração com banco de dados. Para o curso, a versão **Community** será suficiente. Para o *download*, siga as instruções dependendo de seu sistema operacional.



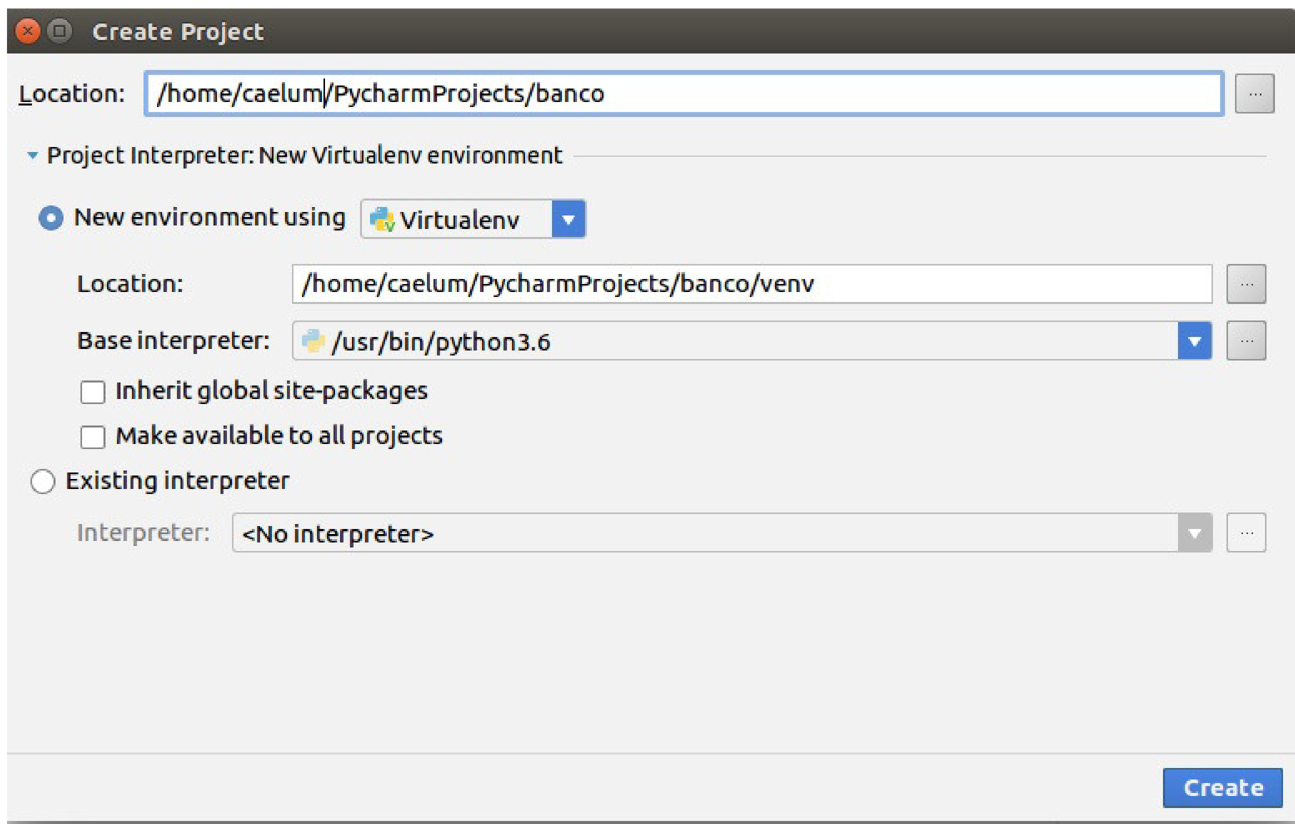
Have you ever wanted to learn programming with Python? Or maybe you're using Python to teach programming?
Then we have the perfect tool for you.

CHECK IT OUT!

Se você precisar de ajuda para fazer a instalação, consulte as instruções de instação neste link: https://www.jetbrains.com/help/pycharm/install-and-set-up-pycharm.html

9.4 CRIANDO UM PROJETO

Ao abrir o PyCharm pela primeira vez, uma janela chamada `Create Project` aparecerá. É nela que definimos todas as configurações necessárias.



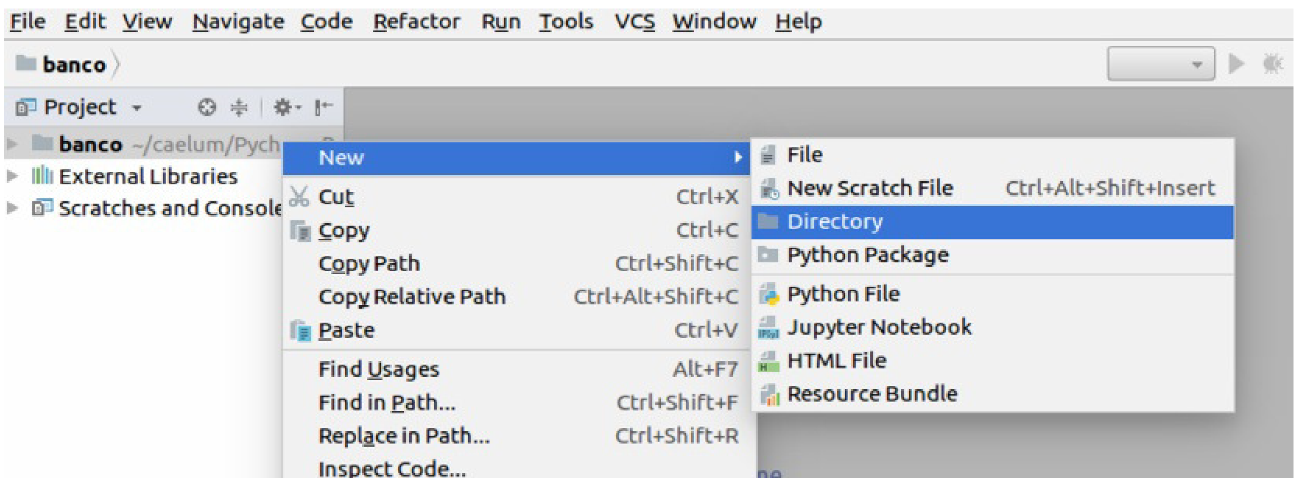
Podemos criar um novo projeto a qualquer momento. Para fazer isso, basta clicar em `File -> New Project` no menu superior da janela principal do PyCharm.

Primeiro, especificamos o nome do projeto - no nosso caso será apenas *banco*. Note que o PyCharm sugere um local padrão para salvar o projeto. Você pode aceitar este local ou configurar manualmente no campo `Location` . Vamos optar pelo caminho padrão. Ao fazer isso, a IDE vai criar uma pasta chamada `PyCharmProjects` na sua pasta home.

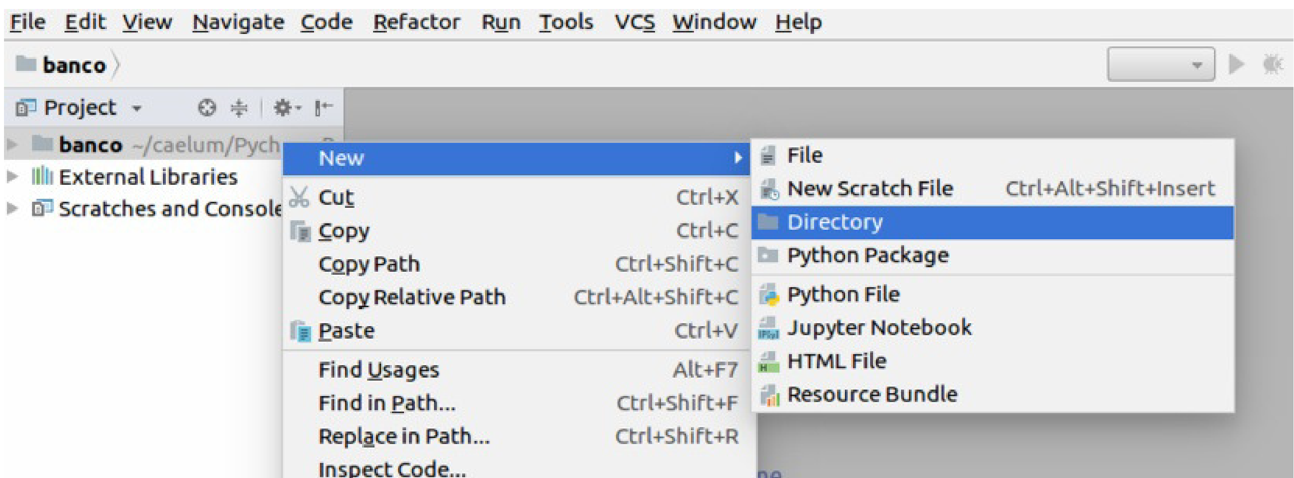
Após isso, escolhemos a versão do interpretador que usaremos no projeto. O PyCharm cria um ambiente isolado da instalação padrão do sistema operacional (no caso do Linux e MacOS). Isso é muito importante e não causa concorrência com outras bibliotecas instaladas em seu computador. Por fim, clicamos em `Create` e nosso projeto é criado.

Nosso projeto tem uma estrutura padrão. A pasta `venv` é o ambiente isolado do sistema operacional. Nela contém a versão do interpretador Python que selecionamos na criação do projeto e seus módulos embutidos (*builtins*) - você pode checar isso na pasta `lib` . A qualquer momento podemos incluir novas bibliotecas ao projeto.

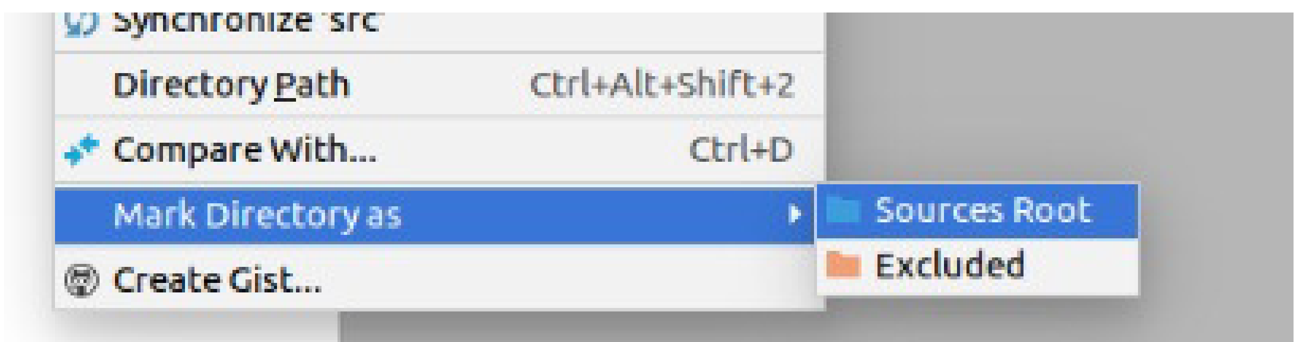
Vamos iniciar nosso projeto criando a classe `Conta` . Antes, precisamos criar uma pasta raiz do projeto. Vá em `File -> New -> Directory` .



Definimos um nome para nosso diretório e clicamos em 'OK'. Vamos criar um diretório chamado `src` :



Para defini-lo como pasta raiz, vamos clicar com o botão direito do mouse no diretório, navegar até `Mark Directory as` e escolher `Sources Root` . Você notará que a pasta muda da cor cinza para azul.



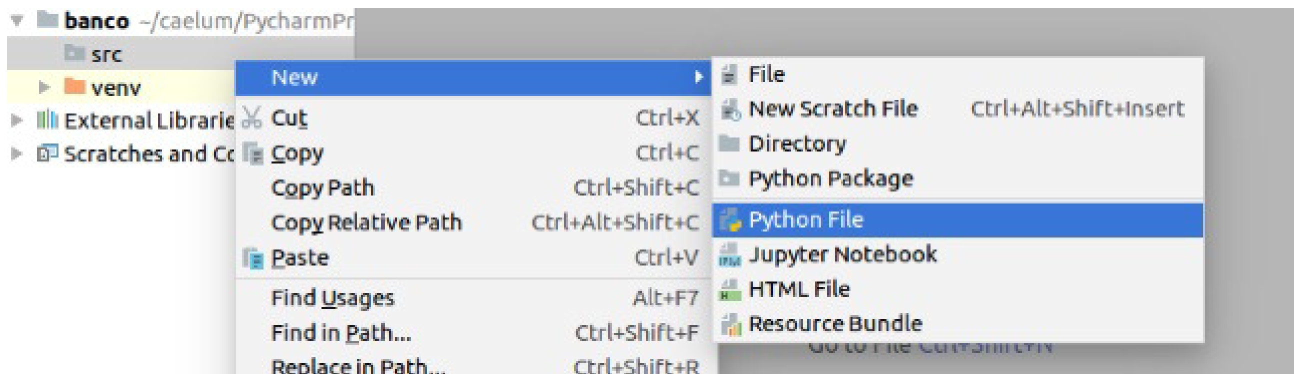
Agora é a melhor hora de aprender algo novo

alura Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

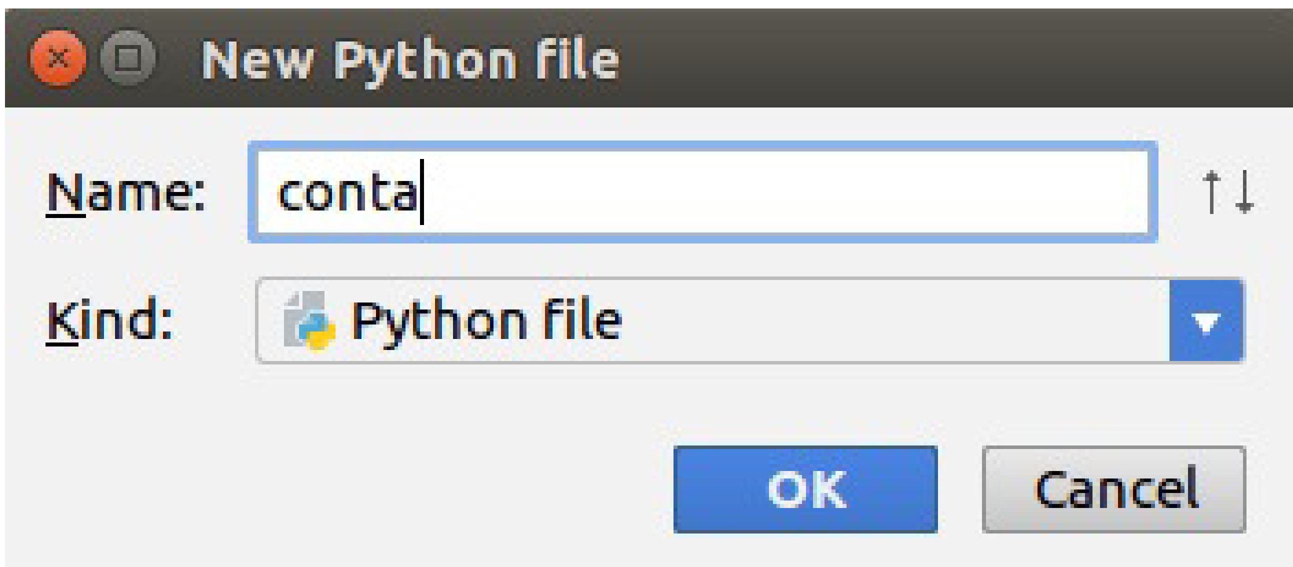
Conheça a Alura Cursos Online.

9.5 CRIANDO UMA CLASSE

Agora vamos criar o arquivo `conta.py` que conterá nossa classe `Conta` . Para isso, clique com o botão direito do mouse em `src` e vá em `New -> Python File` :

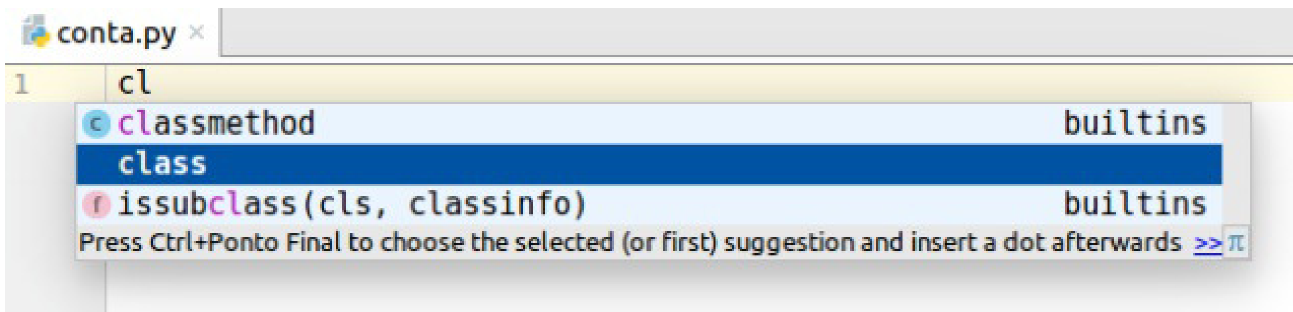


Digite o nome do arquivo e clique em 'OK'. Vamos nomear o arquivo como `conta` :

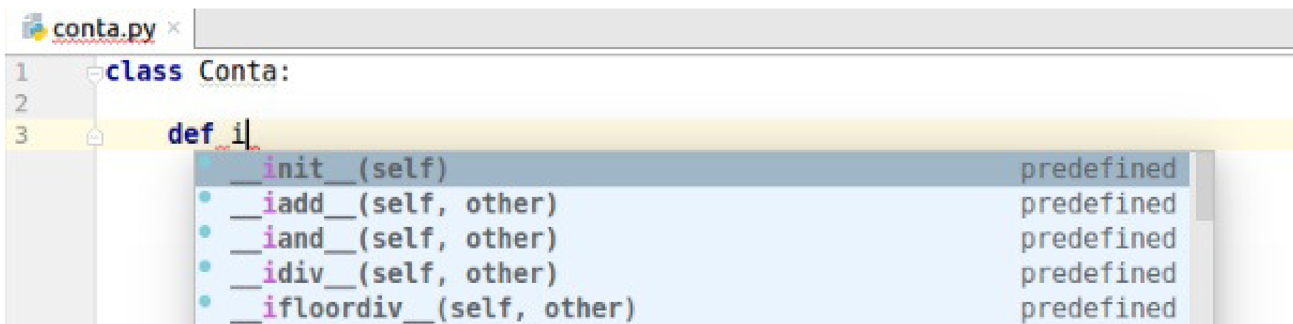


Outra maneira de criar um arquivo python é clicar na pasta src e digitar o atalho `ALT + Insert` e a janela para você entrar com o nome do arquivo vai aparecer.

Uma nova aba vai aparecer com o nome do módulo que criamos, à direita do menu de navegação do projeto. Vamos começar a escrever o código de nossa classe `Conta`. Você vai notar que quando começamos a digitar a palavra `class` o PyCharm vai te oferecer sugestões para você escolher. Comece escrevendo a palavra `class` (exemplo: 'cl') e ele vai terminar de digitar pra você:



Vamos agora escrever o método `__init__()`. Aqui também o Pycharm vai nos ajudar. Escreva apenas `"def i"` e o PyCharm vai te dar novamente sugestões:



Escolha a primeira opção `__init__(self)` e aperte `ENTER` . Vai gerar o código:

```
class Conta:

    def __init__(self):
```

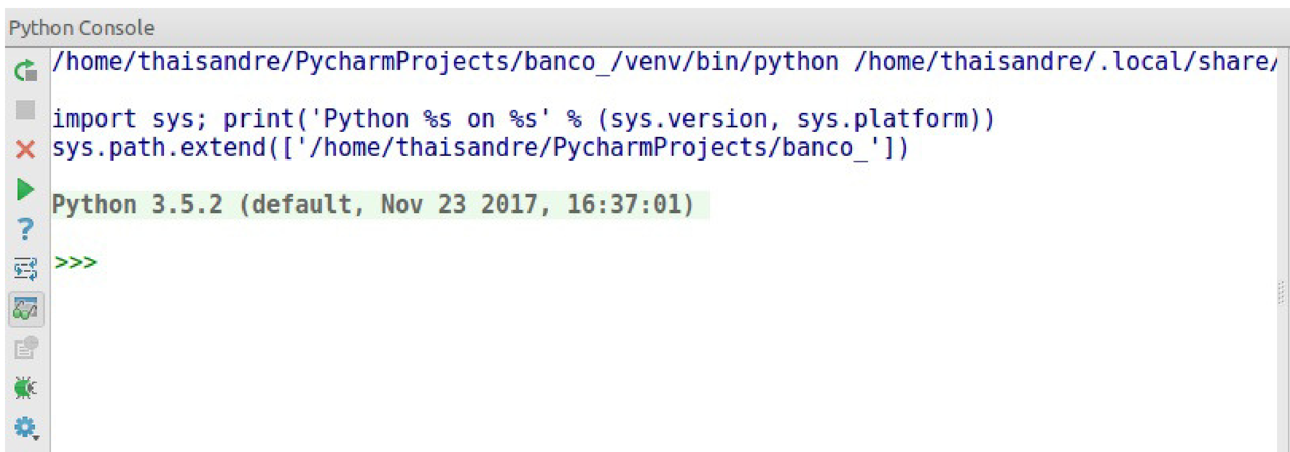
Faça o método inicializador receber como parâmetro os valores do atributos de uma `Conta` , ou seja, receber o `numero` , `titular` , `saldo` e `limite` :

```
class Conta:

    def __init__(self, numero, titular, saldo, limite=1000.0):
        self.numero = numero
        self.titular = titular
        self.saldo = saldo
        self.limite = limite
```

9.6 EXECUTANDO CÓDIGO

Agora vamos testar nossa classe. O Pycharm possui um console do Python embutido, para abrí-lo vá em `Tools -> Python Console` . Você vai notar que a janela do console vai abrir abaixo do arquivo conta:



Vamos importar o módulo `conta` com o comando `from src.conta import Conta` , instanciar uma `Conta` e acessar seus atributos:

Python Console

```
sys.path.extend(['/home/thaisandre/PycharmProjects/banco_'])

Python 3.5.2 (default, Nov 23 2017, 16:37:01)
>>> from src.conta import Conta
>>> conta = Conta('123-4', 'João', 1000.0)
>>> conta.numero
'123-4'
>>> conta.titular
'João'
>>> conta.saldo
1000.0
>>> conta.limite
1000.0
```

Repare que o console também possui a ferramenta de autocomplete. Para reiniciá-lo basta clicar no primeiro ícone do menu esquerdo do console.

A IDE também permite abrir o terminal e usar o modo interativo para testes. O atalho para abrir o terminal é `ALT + F12`. O Python Console do Pycharm é mais aconselhável para isso e o terminal é mais utilizado para instalar novas libs ao seu projeto.

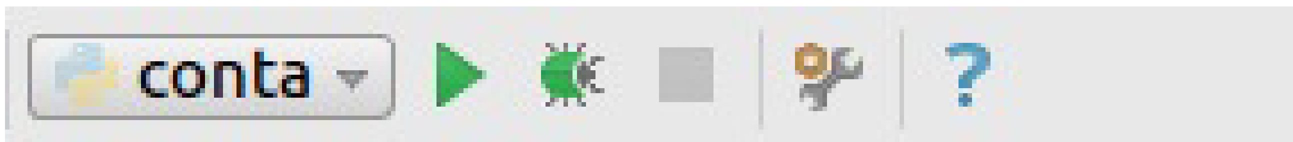
Mas usaremos o console apenas para testes. Vamos executar o código diretamente do nosso arquivo conta. Para isso precisamos acrescentar a condicional `if __name__ == '__main__':` e fazer os mesmos testes que fizemos no console dentro desta condição `if`. Basta você escrever "main" e digitar `ENTER` que o Pycharm cria a condicional para você:

```
if __name__ == '__main__':
```

Agora vamos instanciar e imprimir os atributos de uma `Conta` como fizemos utilizando o Python Console. Não esqueça de utilizar a função *print* na hora de mostrar os atributos já que não estamos mais no modo interativo:

```
if __name__ == '__main__':
    conta = Conta('123-4', 'João', 1000.0)
    print(conta.numero)
    print(conta.titular)
    print(conta.saldo)
    print(conta.limite)
```

Para executar vá em `Run -> Run` ou clique com o botão direito do mouse no interior do arquivo *conta* e escolha a opção `Run 'conta'`. Ou ainda, digite o atalho `CTRL+Shift+F10` que vai ter o mesmo efeito. Depois de ter rodado pela primeira vez, para você rodar novamente basta clicar no ícone de uma pequena seta verde no menu superior da IDE:



9.7 CRIANDO MÉTODOS

Vamos criar nosso primeiro método. Primeiro, dentro da condicional `main` , vamos digitar a seguinte linha de código:

```
conta.deposita(100.0)
```

Se executarmos esse código o nosso programa quebra já que a classe `Conta` ainda não possui o método `deposita()` . Para criá-lo, coloque o cursor do mouse na palavra "deposita" e use o atalho `ALT + ENTER` , várias sugestões do Pycharm irão aparecer. Clique em `Add method deposita() to class Conta` :

```
if __name__ == '__main__':
    conta = Conta('123-4', 'João', 1000.0)
    print(conta.numero)
    print(conta.titular)
    print(conta.saldo)
    print(conta.limite)

    conta.deposita(50.0)
```

Add method deposita() to class Conta
Ignore unresolved reference 'src.conta.Conta.deposita'
Mark all unresolved attributes of 'src.conta.Conta' as ignored

E o Pycharm vai criar a declaração do método para você:

```
class Conta:

    def __init__(self, numero, titular, saldo, limite=1000.0):
        self.numero = numero
        self.titular = titular
        self.saldo = saldo
        self.limite = limite

    def deposita(self, param):
        pass
```

Viu como é fácil e rápido? Agora basta trocar a palavra `param` por `valor` , apagar a palavra `pass`

e adicionar a implementação do método:

```
def deposita(self, valor):
    self._saldo += valor
```

No próximo exercício vamos criar nosso primeiro projeto do banco e nossa classe `Conta` utilizando o Pycharm e praticar o que aprendemos de orientação a objetos e sobre a IDE até aqui.

Editora Casa do Código com livros de uma forma diferente

Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.
Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.]

9.8 PRINCIPAIS ATALHOS

O Pycharm possui muitos atalhos úteis na hora de desenvolver. Abaixo estão os principais deles para você conhecer e praticar:

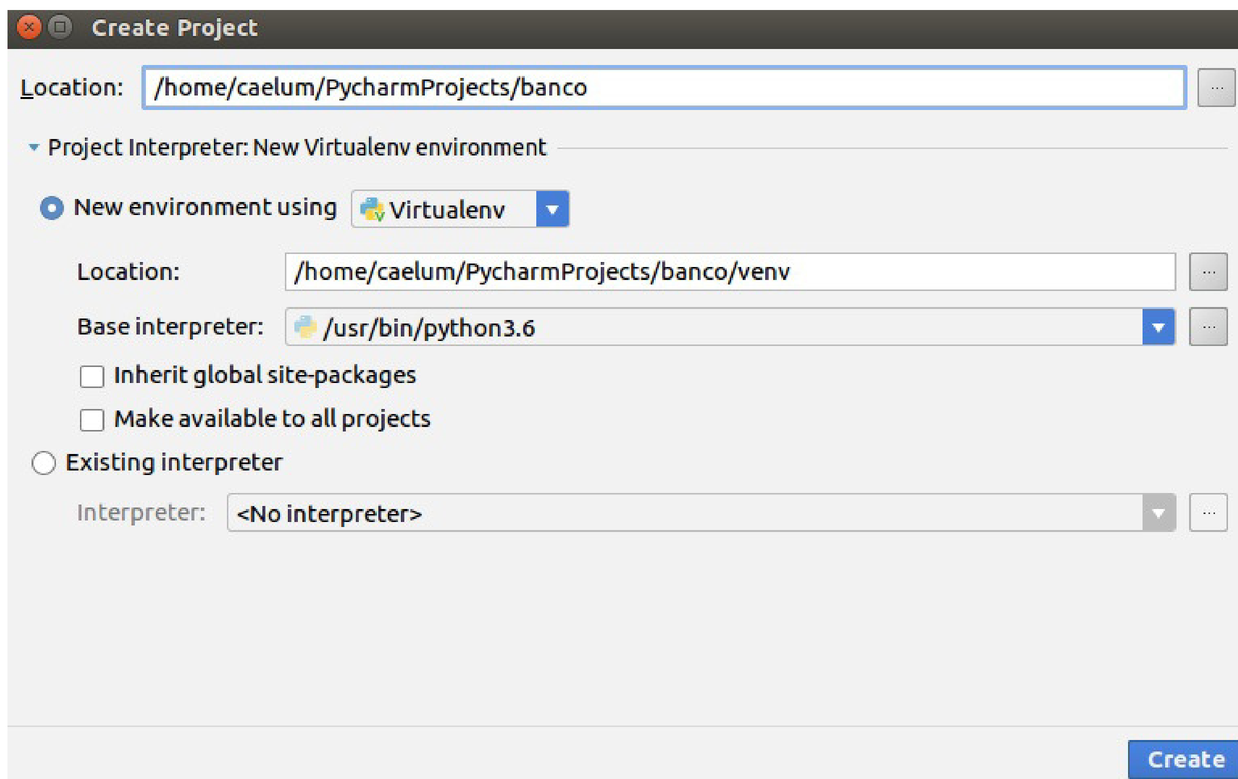
- ALT + INSERT: cria novo arquivo
- ALT + ENTER: sugere ações e conserta erros rápidos
- CTRL + ESPAÇO: autocomplete
- CTRL + N: busca classes
- CTRL + SHIFT + N: busca arquivos
- CTRL + ALT + M: extrair código para um método
- CTRL + ALT + V: extrair para uma variável
- CTRL + ALT + SHIFT + T: refatoração (renomear)
- CTRL + A: seleciona tudo
- CTRL + SHIFT + LEFT/RIGHT: seleciona parte do texto a esquerda ou a direita.

- CTRL + SHIFT + PAGE DOWN/PAGE UP - move linha para cima ou para baixo
- ALT + J: procura próxima palavra selecionada

A JetBrains fez uma tabela com todos os atalhos que você pode checar neste link: resources.jetbrains.com/storage/products/pycharm/docs/PyCharm_ReferenceCard.pdf

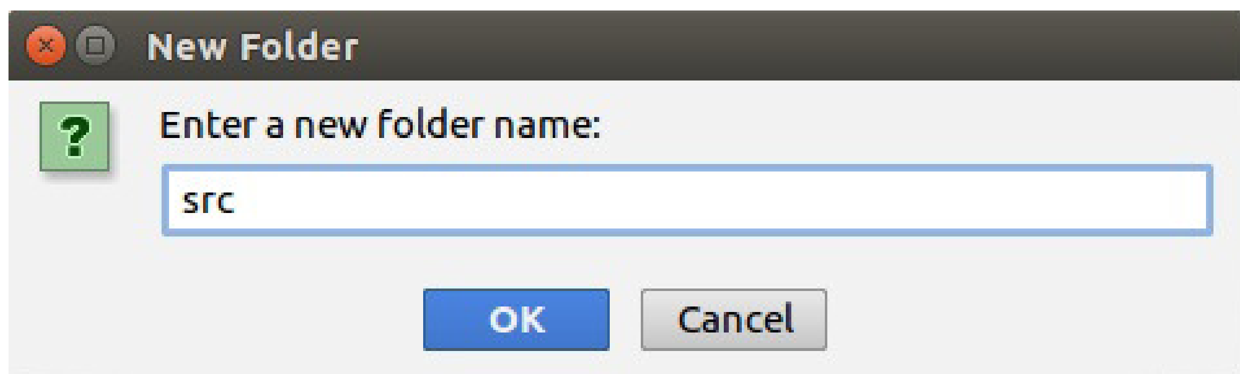
9.9 EXERCÍCIO - CRIANDO PROJETO BANCO NO PYCHARM

1. Abra o Pycharm e vá em `File -> New Project`. A janela abaixo vai aparecer. Troque a palavra **untitled** pelo nome do nosso projeto que será **banco**:

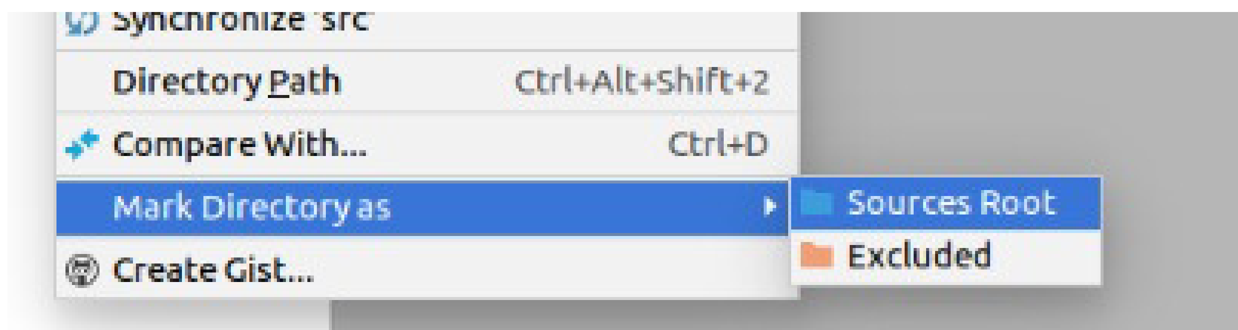


Verifique se a versão do Python está correta em **Base interpreter** e clique em `OK`.

2. No menu esquerdo vai aparecer a estrutura do projeto. Vamos definir uma pasta raiz onde ficarão nossos arquivos de código python. Clique com o botão direito na pasta **banco** e escolha a opção *New Folder*. Uma nova janela vai aparecer para você entrar com o nome do diretório, digite *src* e `OK`:



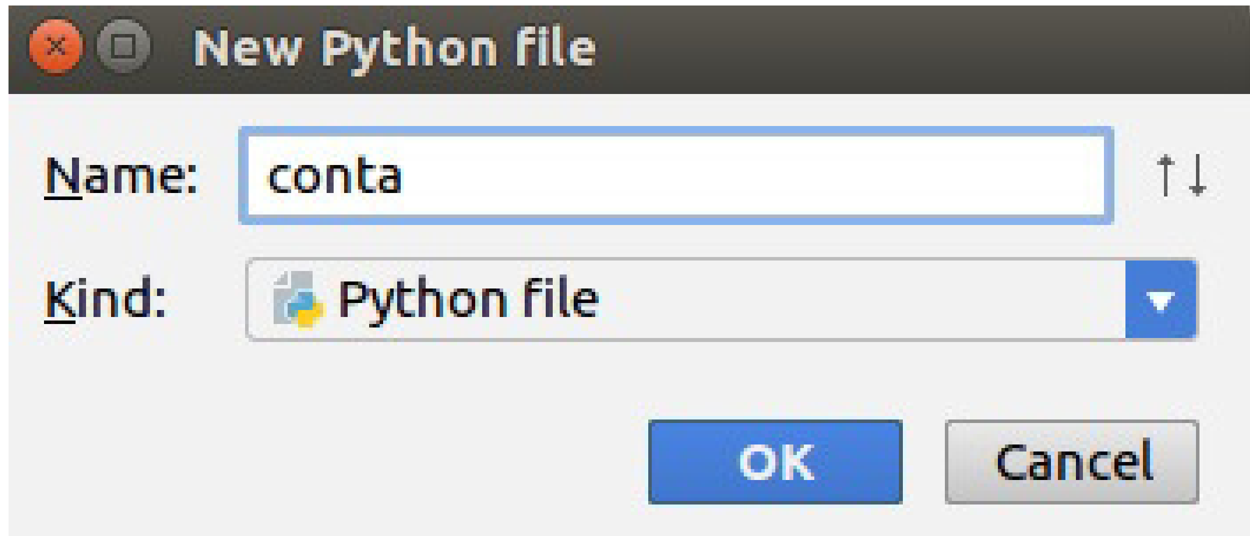
3. Após isso, clicamos com o botão direito na pasta **src** e selecionamos `Mark Directory as -> Sources Roots` para avisar o PyCharm que esta pasta será um diretório fonte de nosso projeto, onde ficarão nossos arquivos .py.



Repare que a pasta ficará da cor azul depois de executar desta ação:

4. Agora vamos criar nossa classe `Conta` que ficará no arquivo `conta.py` . Vamos fazer isso

utilizando um atalho do PyCharm. Coloque o cursor do mouse na pasta **src** e digite `ALT + Insert` . Escolha a opção *Python File*. Uma nova janela vai abrir, digite "conta" e clique em `OK` .



5. O arquivo será aberto a esquerda. Vamos começar a criar nossa classe. Ao começar escrever a função **init** a própria IDE vai mostrar opções em uma janela, basta clicar que ele auto-completa para você já com a argumento 'self'. Adicione os atributos de uma `Conta` como fizemos no exercício do capítulo anterior:

```
class Conta:

    def __init__(self, numero, titular, saldo, limite):
        self._numero = numero
        self._titular = titular
        self._saldo = saldo
        self._limite = limite
```

Aproveite a crie as properties de cada atributo. Abuse do `CTRL+ESPAÇO` para a IDE auto completar para você e do `ALT + ENTER` para sugestões.

6. Em seguida criamos a condicional para que o PyCharm rode algumas linhas de código caso o `__name__` seja igual a `__main__` , ou seja, o programa principal. O Pycharm também facilita esta criação, basta digitar a palavra 'main' e apertar `CTRL + ESPAÇO` que a estrutura do if é construída para você:

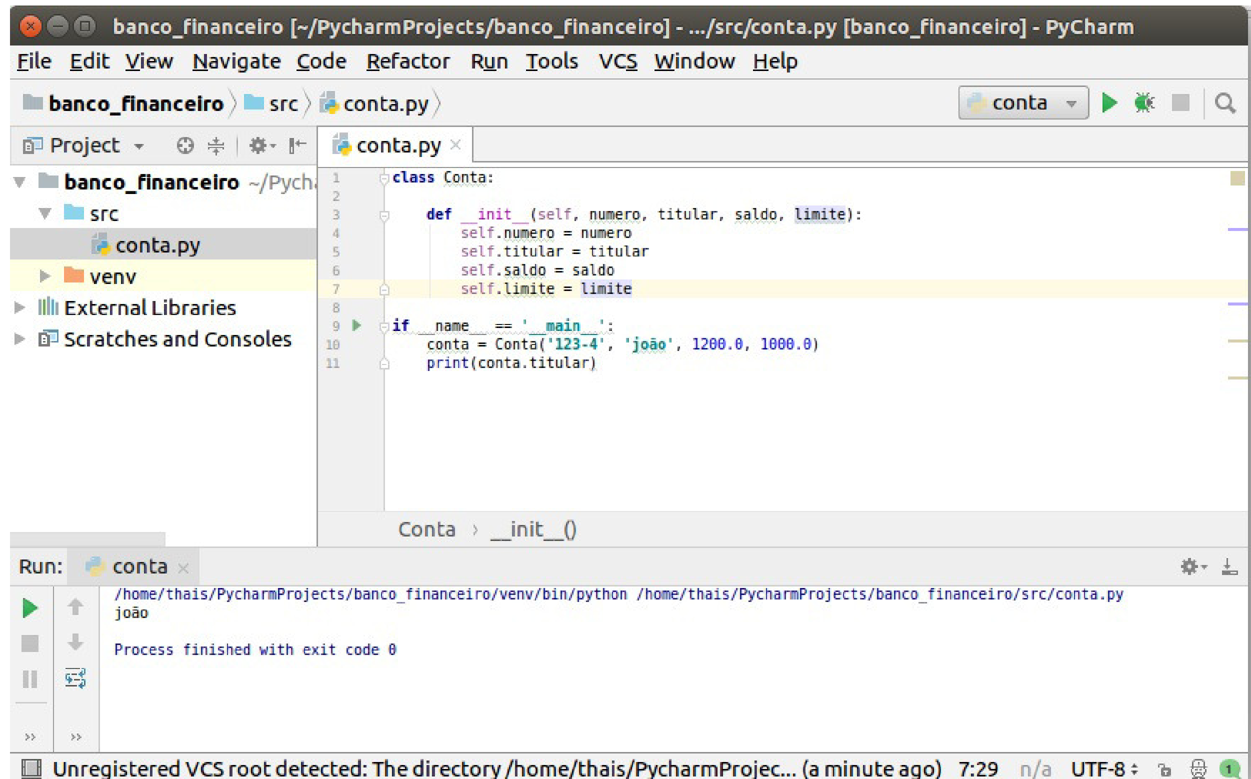
```
if __name__ == '__main__':
```

Vamos criar uma nova conta e imprimir o titular:

```
if __name__ == '__main__':
    conta = Conta('123-4','joão',1200.0, 1000.0)
    print(conta.titular)
```

7. Para rodar, basta clicar com o botão direito do mouse e escolher a opção `Run 'conta'` ou utilizar o

atalho `Ctrl+Shift_F10`. Ou ainda escolher a opção da barra de ferramentas com o símbolo de `play` da cor verde. O resultado vai aparecer no console, na janela inferior da IDE.



8. Crie os métodos `deposita()`, `saca()`, `extrato()` e `transfere_para()` como fizemos no último exercício. Aproveite os recursos da IDE que aprendemos para criar todos esses métodos. A propriedade *setter* do saldo é necessária?

9. Crie duas contas e teste os métodos que você criou no exercício anterior.

10. (Opcional) Crie um arquivo python chamado `cliente.py` e crie a classe `Cliente` com `nome`, `sobrenome` e `cpf`. Teste o código passando um cliente como titular de um `Conta`. Aproveite e adicione alguns métodos a ela.

Veja que a IDE facilita bastante na hora do desenvolvimento e ganhamos tempo também rodando o script diretamente no PyCharm.

CAPÍTULO 10

HERANÇA E POLIMORFISMO

10.1 REPETINDO CÓDIGO?

Como toda empresa, nosso banco possui funcionários. Um funcionário tem um nome, um cpf e um salário. Vamos modelar a classe `Funcionario` :

```
class Funcionario:

    def __init__(self, nome, cpf, salario):
        self._nome = nome
        self._cpf = cpf
        self._salario = salario

    # outros métodos e propriedades
```

Além de um funcionário comum, há também outros cargos, como os gerentes. Os gerentes guardam a mesma informação que um funcionário comum, mas possuem outras informações, além de ter funcionalidades um pouco diferentes. Um gerente no nosso banco possui também uma senha numérica que permite o acesso ao sistema interno do banco, além do número de funcionários que ele gerencia:

```
class Gerente:

    def __init__(self, nome, cpf, salario, senha, qtd_gerenciados):
        self._nome = nome
        self._cpf = cpf
        self._salario = salario
        self._senha = senha
        self._qtd_gerenciados = qtd_gerenciados

    def autentica(self, senha):
        if self._senha == senha
            print("acesso permitido")
            return True
        else:
           print("acesso negado")
           return False

    # outros métodos (comuns a um Funcionario)
```

Se tivéssemos um outro tipo de funcionário que tem características diferentes do funcionário comum, precisaríamos criar uma outra classe e copiar o código novamente.

Além disso, se um dia precisarmos adicionar uma nova informação para todos os funcionários, precisaremos passar por todas as classes de funcionário e adicionar esse atributo. O problema acontece novamente por não centralizarmos as informações principais do funcionário em um único lugar!

Existe um jeito de relacionarmos uma classe de tal maneira que uma delas **herda** tudo que o outra tem. Isto é uma relação de **herança**, uma relação entre classe 'mãe' e classe 'filha'. No nosso caso, gostaríamos de fazer com que `Gerente` tivesse tudo que um `Funcionario` tem, gostaríamos que ela fosse uma extensão de `Funcionario`. Fazemos isso acrescentando a classe mãe entre parenteses junto a classe filha:

```
class Gerente(Funcionario):

    def __init__(self, senha, qtd_funcionarios):
        self._senha = senha
        self._qtd_funcionarios = qtd_funcionarios

    def autentica(self, senha):
        if self._senha == senha:
            print("acesso permitido")
            return True
        else:
            print("acesso negado")
            return False
```

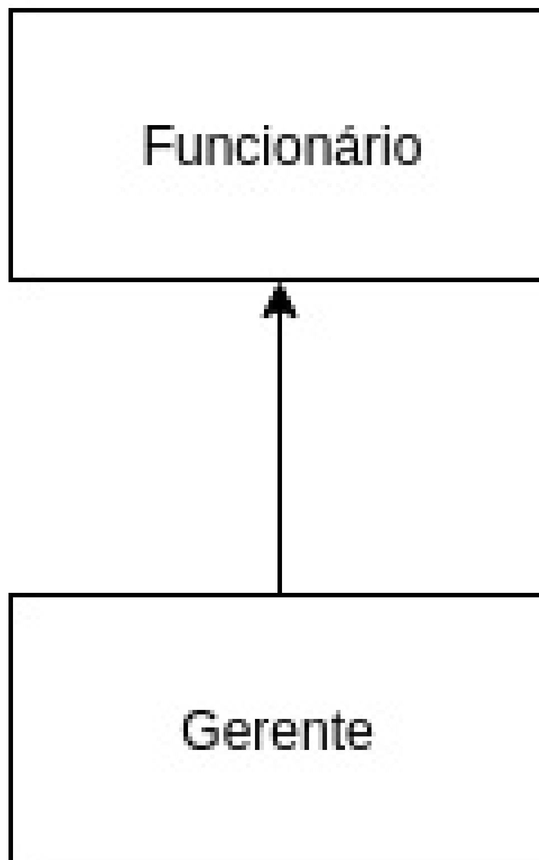
Todo momento que criarmos um objeto do tipo `Gerente` queremos que este objeto também herde os atributos definidos na classe `Funcionario`, pois um **Gerente é um Funcionário**.

Como a classe `Gerente` já possui um método `__init__()` com outros atributos, o método da classe `Funcionario` é **sobrescrito** pelo `Gerente`. Se queremos incluir os mesmos atributos de instância de `Funcionario` em um `Gerente` devemos chamar o método `__init__()` de `Funcionario` dentro do método `__init__()` de Gerente:

```
class Gerente(Funcionario):

    def __init__(self, senha, qtd_funcionarios):
        Funcionario.__init__(nome, cpf, salario)
        self._senha = senha
        self._qtd_funcionarios = qtd_funcionarios

    def autentica(self, senha):
        if self._senha == senha:
            print("acesso permitido")
            return True
        else:
            print("acesso negado")
            return False
```



Dizemos que a classe `Gerente` herda todos os atributos e métodos da classe mãe, no nosso caso, a `Funcionario`. Como Python tem tipagem dinâmica, precisamos garantir isso através do construtor da classe. Além de `senha` e `qtd_funcionarios` passamos também os atributos `nome`, `cpf` e `salario` que todo funcionário tem:

```
class Gerente(Funcionario):

    def __init__(self, nome, cpf, salario, senha, qtd_funcionarios):
        self._senha = senha
        self._qtd_funcionarios = qtd_funcionarios
```

Como estes são atributos de um `Funcionario` e não queremos repetir o código do método `__init__()` de `Funcionario` dentro da classe `Gerente`, podemos chamar este método da classe mãe como fizemos no exemplo acima ou podemos utilizar um método do Python chamado **super()**:

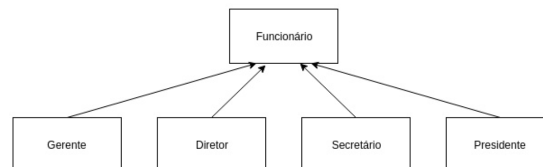
```
class Gerente(Funcionario):
```

```python
def __init__(self, nome, cpf, salario, senha, qtd_funcionarios):
    super().__init__(nome, cpf, salario)
    self._senha = senha
    self._qtd_funcionarios = qtd_funcionarios
```

Para ser mais preciso, ela também herda os atributos e métodos 'privados' de `Funcionario`. O `super()` é usado para fazer referência a superclasse, a classe mãe - no nosso exemplo a classe `Funcionario`.

> **PARA SABER MAIS: SUPER E SUB CLASSE**
>
> A nomenclatura mais encontrada é que Funcionario é a **superclasse** de Gerente, e Gerente é a **subclasse** de Funcionario. Dizemos também que todo Gerente **é um** Funcionario. Outra forma é dizer que Funcionario é a classe **mãe** de Gerente e Gerente é a classe **filha** de Funcionario.

Da mesma maneira, podemos ter uma classe `Diretor` que estenda `Gerente` e a classe `Presidente` pode estender diretamente de `Funcionario`. Fique claro que essa é uma relação de negócio. Se `Diretor` vai estender de `Gerente` ou não, vai depender, para você, `Diretor` é um `Gerente`?



Já conhece os cursos online Alura?

alura

A **Alura** oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex-aluno da Caelum tem 15% de desconto neste link!

Conheça os cursos online Alura.

10.2 REESCRITA DE MÉTODOS

Todo fim de ano, os funcionários do nosso banco recebem uma bonificação. Os funcionários comuns recebem 10% do valor do salário e os gerentes, 15%.

Vamos ver como fica a classe `Funcionario` :

```
class Funcionario:

    def __init__(self, nome, cpf, salario):
        self._nome = nome
        self._cpf = cpf
        self._salario = salario

    # outros métodos e properties

    def get_bonificacao(self):
        return self._salario * 0.10
```

Se deixarmos a classe `Gerente` como ela está, ela vai herdar o método `get_bonificacao()`

```
gerente = Gerente('José', '222222222-22', 5000.0, '1234', 0)
print(gerente.get_bonificacao())
```

O resultado aqui será 500. Não queremos essa resposta, pois o gerente deveria ter 750 de bônus nesse caso. Para consertar isso, uma das opções seria criar um novo método na classe `Gerente` , chamado, por exemplo, `get_bonificacao_do_gerente()` . O problema é que teríamos dois métodos em `Gerente` , confundindo bastante quem for usar essa classe, além de que cada um gerenciaria uma resposta diferente.

No Python, quando herdamos um método, podemos alterar seu comportamento. Podemos **reescrever** (sobrescrever, override) este método, assim como fizemos com o `__init__` :

```
class Gerente(Funcionario):

    def __init__(self, nome, cpf, salario, senha, qtd_gerenciaveis):
        super().__init__(nome, cpf, salario)
        self._senha = senha
        self._qtd_gerenciaveis = qtd_gerenciaveis

    def get_bonificacao(self):
        return self._salario * 0.15

    # metodos e properties
```

Agora o método está correto para o `Gerente` . Refaça o teste e veja que o valor impresso é o correto (750):

```
gerente = Gerente('José', '222222222-22', 5000.0, '1234', 0)
print(gerente.get_bonificacao())
```

Utilize o método `vars()` para acessar os atributos de `Gerente` e ver que a classe herda todos os atributos de `Funcionario` :

```
funcionario = Funcionario('João', '111111111-11', 2000.0)
```

```
print(vars(funcionario))

gerente = Gerente('José', '222222222-22', 5000.0, '1234', 0)
print(vars(gerente))
```

Saída:

```
{'_salario': 2000.0, '_nome': 'João', '_cpf': '111111111-11'}
{'_cpf': '222222222-22', '_salario': 5000.0, '_nome': 'José', '_qtd_funcionarios': 0, '_senha': '1234'
}
```

10.3 INVOCANDO O MÉTODO REESCRITO

Depois de reescrito, não podemos mais chamar o método antigo que fora herdado da classe mãe, realmente alteramos o seu comportamento. Mas podemos invocá-lo no caso de estarmos dentro da classe.

Imagine que para calcular a bonificação de um `Gerente` devemos fazer igual ao cálculo de um `Funcionario` adicionando 1000.0 reais. Poderíamos fazer assim:

```
class Gerente(Funcionario):

    def __init__(self, senha, qtd_gerenciaveis):
        self._senha = senha
        self._qtd_gerenciaveis = qtd_gerenciaveis

    def get_bonificacao():
        return self._salario * 0.10 + 1000.0

    # métodos e properties
```

Aqui teríamos um problema: o dia que o `get_bonificacao()` do `Funcionario` mudar, precisaremos mudar o método do `Gerente` para acompanhar a nova bonificação. Para evitar isso, o `get_bonificacao()` do `Gerente` pode chamar o do `Funcionario` utilizando o método **super()**.

```
class Gerente(Funcionario):

    def __init__(self, senha, qtd_gerenciaveis):
        self._senha = senha
        self._qtd_gerenciaveis = qtd_gerenciaveis

    def get_bonificacao():
        return super().get_bonificacao() + 1000

    # métodos e properties
```

Essa invocação vai procurar o método com o nome `get_bonificacao()` de uma superclasse de `Gerente` . No caso, ele logo vai encontrar esse método em `Funcionario` .

Essa é uma prática comum, pois em muitos casos o método reescrito geralmente faz algo a mais que o método da classe mãe. Chamar ou não o método de cima é uma decisão e depende do seu problema. Algumas vezes não faz sentido invocar o método que reescrevemos.

Para escrever uma classe utilizando o Python 2 é preciso acrescentar a palavra 'object' quando

definimos uma classe:

```
class MinhaClasse(object):
    pass
```

Isso acontece porque toda classe é filha de **object** - que é chamada a mãe de todas as classes. No Pyhton, toda classe herda de *object*. No Python 3 não precisamos acrescentar o *object* mas não quer dizer que esta classe e a herança não existam, apenas que essa herança é implícita. Quando criamos uma classe vazia e utilizamos o método `dir()` para checar a lista de seus atributos, reparamos que ela não é vazia:

```
class MinhaClasse():
    pass

if __name__ == '__main__':
    mc = MinhaClasse()
    print(dir(mc))
```

Gera a saída:

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
'__init__', '__init_subclass__', '__le__', '__lt__', '__module__'
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__']
```

Todos estes atributos são herdados da classe *object* e podemos reescrever qualquer um deles na nossa subclasse. Todos eles são os conhecidos métodos 'mágicos' (começam e iniciam com dois *underscores*, e por este motivo, também chamados de *dunders*).

Vimos o comportamento do `__init__()` , `__new__()` e do `__dict__` . Outros métodos mágicos famosos são `__str__()` e `__repr__()` - métodos que retornam a representação do objeto como uma *string*. Quando chamamos `print(mc)` temos a saída

```
<__main__.MinhaClasse object at 0x7f11c1f59a58>
```

Esse é o modelo padrão de impressão de um objeto, implementado na classe `object` . A função `print()` na verdade usa a `string` definida pelo método `__str__()` de uma classe. Vamos reescrever este método:

```
class MinhaClasse:

    def __str__(self):
        return '< Instância de {}; endereço:{}>'.format(self.__class__.__name__, id(self))
```

Agora, quando executamos `print(mc)` , a saída é:

```
<Instância de MinhaClasse; endereço:0x7f11c1f59a58>
```

O Python sempre chama o método `__str__()` quando utiliza a função `print()` em um objeto. Novamente, estamos utilizando reescrita de métodos.

O método `__repr__()` também retorna uma `string` e podemos utilizar a função `repr()` para checar seu retorno:

```
class MinhaClasse():
    pass

if __name__ == '__main__':
    mc = MinhaClasse()
    print(repr(mc))
```

Que vai gerar a mesma saída padrão do `__str__()` :

```
<__main__.MinhaClasse object at 0x7f11c1f59a58>
```

Mas diferente do `__str__()` , não é comum sobrescrever este método. Ele é sobrescrito quando precisamos utilizá-lo junto com a função `eval()` do Python. A função `eval()` recebe uma `string` e tenta executar essa `string` como um comando do Python, veja um exemplo de uso:

```
>>> x = 1
>>> eval("x+1")
2
```

Vamos a um exemplo utilizando classes:

```
class Ponto:

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return "({}, {})".format(self.x, self.y)

    def __repr__(self):
        return "Ponto({}, {})".format(self.x + 1, self.y + 1)

if __name__ == '__main__':
    p1 = Ponto(1, 2)
    p2 = eval(repr(p1))

    print(p1)
    print(p2)
```

Se executarmos o código acima, temos:

```
(1, 2)
(2, 3)
```

Repare que utilizamos a função `repr()` passando uma instância de `Ponto` . O Python vai chamar então o método `__repr__()` da classe `Ponto` , que retorna a `string` "Ponto(2, 3)" já que `p1.x = 1` e `p1.y = 2` . Ao passá-la de argumento para a função `eval()` , teremos: `p2 = eval('Ponto(2, 3))` . Como a função `eval()` vai tentar executar essa `string` como um comando Python válido, ele vai ter sucesso e portanto `p2` será uma nova instância da classe `Ponto` com `p2.x = 2` e `p2.y = 3` .

Para concluir, é importante entender que tanto `__str__()` quanto `__repr__()` retornam uma `string` que representa o objeto mas com propósitos diferentes. O método `__str__()` é utilizado para apresentar mensagens para os usuários da classe, de maneira mais amigável. Já o método `__repr__()` é usado para representar o objeto de maneira técnica, inclusive podendo utilizá-lo como comando válido

do Python como vimos no exemplo da classe `Ponto` .

10.4 PARA SABER MAIS - MÉTODOS MÁGICOS

Os métodos mágicos são úteis pois permitem que os objetos de nossas classes possuam uma interface de acesso semelhante aos objetos embutidos do Python. O método `__add__()` , por exemplo, serve para executar a adição de dois objetos e é chamada sempre quando fazemos a operação de adição (obj + obj) utilizando o operador '+'. Por exemplo, quando fazemos `1 + 1` no Python, o que o interpretador faz é chamar o método `__add__()` da classe `int` . Vimos que uma `list` também implementa o método `__add__()` já que a operação de adição é definida para esta classe:

```
>>> lista = [1, 2, 3]
>>> lista + [4, 5]
[1, 2, 3, 4, 5]
```

O mesmo ocorre para as operações de multiplicação, divisão, módulo e potência que são definidas pelos métodos mágicos `__mul__()` , `__div__()` , `__mod__()` e `__pow__()` , respectivamente.

Podemos definir cada uma dessas operações em nossas classes sobrescrevendo tais métodos mágicos. Além desses o Python possui muitos outros que você pode acessar aqui: https://docs.python.org/3/reference/datamodel.html#Basic_customization

Você pode também fazer o curso PY-14 dessa apostila na Caelum

Querendo aprender ainda mais sobre? Esclarecer dúvidas dos exercícios? Ouvir explicações detalhadas com um instrutor?
A Caelum oferece o **curso PY-14** presencial nas cidades de São Paulo, Rio de Janeiro e Brasília, além de turmas incompany.

Consulte as vantagens do curso *Python e Orientação a Objetos*

10.5 POLIMORFISMO

O que guarda uma variável do tipo `Funcionario` ? Uma referência para um `Funcionario` , nunca o

objeto em si.

Na herança, vimos que todo `Gerente` é um `Funcionario`, pois é uma extensão deste. Podemos nos referir a um `Gerente` como sendo um `Funcionario`. Se alguém precisa falar com um `Funcionario` do banco, pode falar com um `Gerente`! Porque? Pois `Gerente` **é um** `Funcionario`. Essa é a semântica da herança.

Polimorfismo é a capacidade de um objeto poder ser referenciado de várias formas. (cuidado, polimorfismo não quer dizer que o objeto fica se transformando, muito pelo contrário, um objeto nasce de um tipo e morre daquele tipo, o que pode mudar é a maneira como nos referimos a ele).

A situação que costuma aparecer é a que temos um método que recebe um argumento do tipo `Funcionario`:

```
class ControleDeBonificacoes:

    def __init__(self, total_bonificacoes=0):
        self._total_bonificacoes = total_bonificacoes

    def registra(self, funcionario):
        self._total_bonificacoes += funcionario.get_bonificacao()

    @property
    def total_bonificacoes(self):
        return self._total_bonificacoes
```

E podemos fazer:

```
if __name__ == '__main__':
    funcionario = Funcionario('João', '111111111-11', 2000.0)
    print("bonificacao funcionario: {}".format(funcionario.get_bonificacao()))

    gerente = Gerente("José", "222222222-22", 5000.0 '1234', 0)
    print("bonificacao gerente: {}".format(gerente.get_bonificacao()))

    controle = ControleDeBonificacoes()
    controle.registra(funcionario)
    controle.registra(gerente)

    print("total: {}".format(controle.total_bonificacoes))
```

que gera a saída:

```
bonificacao funcionario: 200.0
bonificacao gerente: 1500.0
total: 1700.0
```

Repare que conseguimos passar um `Gerente` para um método que "recebe" um `Funcionario` como argumento. Pense como numa porta na agência bancária com o seguinte aviso: "Permitida a entrada apenas de Funcionários". Um gerente pode passar nessa porta? Sim, pois `Gerente` **é um** `Funcionario`.

Qual será o valor resultante? Não importa que dentro do método `registra()` do

`ControleDeBonificacoes` receba `Funcionario`. Quando ele receber um objeto que realmente é um `Gerente`, o seu método reescrito será invocado. Reafirmando: **não importa como nos referenciamos a um objeto, o método que será invocado é sempre o que é dele**.

No dia em que criarmos uma classe `Secretaria`, por exemplo, que é filha de `Funcionario`, precisaremos mudar a classe `ControleDeBonificacoes`? Não. Basta a classe `Secretaria` reescrever os métodos que lhe parecerem necessários. É exatamente esse o poder do polimorfismo, juntamente com a reescrita de método: diminuir o acoplamento entre as classes, para evitar que novos códigos resultem em modificações em inúmeros lugares.

Repare que quem criou `ControleDeBonificacoes` pode nunca ter imaginado a criação da classe `Secretaria` ou `Engenheiro`. Contudo, não será necessário reimplementar esse controle em cada nova classe: reaproveitamos aquele código.

Pensar desta maneira em linguagens com tipagem estática é o mais correto já que as variáveis são tipadas e garantem, através do compilador, que o método só funcionará se receber um **tipo** `Funcionario`. Mas não é o que acontece em linguagens de tipagem dinâmica como Python. Vamos supor que temos uma classe para representar os clientes do banco:

```
class Cliente:

    def __init__(self, nome, cpf, senha):
        self._nome = nome
        self._cpf = cpf
        self._senha = senha

    # métodos e properties
```

Nada impede de registrarmos um `Cliente` em `ControleDeBonificacoes`. Vamos ver o que acontece:

```
cliente = ('Maria', '333333333-33', '1234')
controle = ControleBonificacoes()
controle.registra(cliente)
```

Saída:

```
  File "<stdin>", line 99, in <module>
    controle.registra(cliente)
  File "<stdin">, line 67, in regista
    self._total_bonificacoes += funcionario.get_bonificacao()
AttributeError: 'Cliente' object has no attribute 'get_bonificacao'
```

Veja que lança um `AttibuteError` com a mensagem dizendo que `Cliente` não possui o atributo `get_bonificacao`. Portanto, aqui não importa se o objeto recebido no método `registra()` é um `Funcionario`, mas se ele possui o método `get_bonificacao()`.

O método `registra()` utiliza um método da classe `Funcionario` e, portanto, funcionará com qualquer instância de uma subclasse de `Funcionario` ou qualquer instância de uma classe que implemente o método `get_bonificacao()`.

Podemos evitar este erro verificando se o objeto passado possui ou não um atributo `get_bonificacao()` através da função `hasattr()` :

```
class ControleDeBonificacoes:

    def __init__(self, total_bonificacoes=0):
        self._total_bonificacoes = total_bonificacoes

    def registra(self, obj):
        if(hasattr(obj, 'get_bonificacao'))
            self._total_bonificacoes += obj.get_bonificacao()
        else
            print('instância de {} não implementa o método get_bonificacao()'.format(self.__class__._
_name__))

    # demais métodos
```

A função `hasattr()` recebe dois parâmetros, o objeto e o atributo (na forma de `string`) - e verifica se o objeto possui aquele atributo, ou seja, se o atributo está contido no `__dict__` do objeto. Então, fazemos a pergunta: `get_bonificacao()` é atributo de `Funcionario` ? Se sim, entra no bloco `if` e podemos chamar o método tranquilamente, evitando erros.

Agora, se tentarmos chamar o método `registra()` passando um `Cliente` recebemos a saída:

```
'Cliente' object has no attribute 'get_bonificacao
```

Portanto, o tipo passado para o método `registra()` não importa aqui e sim se o objeto passado implementa ou não o método `get_bonificacao()` . Ou seja, basta que o objeto atenda a um determinado **protocolo**.

Existe uma função no Python que funciona de forma semelhante mas considera o tipo da instância, é a função **isinstance()**. Ao invés de passar uma instância, passamos a classe no segundo parâmetro.

```
class ControleDeBonificacoes:

    def __init__(self, total_bonificacoes=0):
        self.__total_bonificacoes = total_bonificacoes

    def registra(self, obj):
        if(isinstance(obj, Funcionario)):
            self.__total_bonificacoes += obj.get_bonificacao()
        else:
            print('instância de {} não implementa o método get_bonificacao()'
                .format(self.__class__.__name__))
```

Mas essa não é a maneira Pythônica. Você deve escrever o código esperando somente uma interface do objeto, não o tipo dele. A interface é o conjunto de métodos públicos de uma classe. No caso da nossa classe `ControleDeBonificacoes` , o método `registra()` espera um objeto que possua o método `get_bonificacao()` e não um objeto do tipo `Funcionario` .

10.6 DUCK TYPING

Uma característica de linguagens dinâmicas como Python é a chamada **Duck Typing**, a tipagem de pato. É uma característica de um sistema de tipos em que a semântica de uma classe é determinada pela sua capacidade de responder a alguma mensagem, ou seja, responder a determinado atributo (ou método). O exemplo canônico (e a razão do nome) é o teste do pato: *se ele se parece com um pato, nada como um pato e grasna como um pato, então provavelmente é um pato*.

Veja o exemplo abaixo:

```
class Pato:
    def grasna(self):
        print('quack!')

class Ganso:
    def grasna(self):
        print('quack!')

if __name__ == '__main__':
    pato = Pato()
    print(p.grasna())

    ganso = Ganso()
    print(g.grasna())
```

Que gera a saída:

```
quack!
quack!
```

Você deve escrever o código esperando somente uma interface do objeto, não um tipo de objeto. No caso da nossa classe `ControleDeBonificacoes`, o método `registra()` espera um objeto que possua o método `get_bonificacao()` e não apenas um funcionário.

O *Duck Typing* é um estilo de programação que não procura o tipo do objeto para determinar se ele tem a interface correta. Ao invés disso, o método ou atributo é simplesmente chamado ou usado ('se parece como um pato e grasna como um pato, então deve ser um pato'). *Duck Typing* evita testes usando as funções `type()`, `isinstance()` e até mesmo a `hasattr()` - ao invés disso, deixa o erro estourar na frente do programador.

A maneira Pythônica para garantir a consistência do sistema não é verificar os tipos e atributos de um objeto, mas pressupor a existência do atributo no objeto e tratar uma exceção, caso ocorra, através do comando `try/except`:

```
    try:
        self._total_bonificacoes += obj.get_bonificacao()
    except AttributeError as e:
        print(e)
```

Estamos pedindo ao interpretador para tentar executar a linha de código dentro do comando `try` (tentar). Caso ocorra algum erro, ele vai tratar este erro com o comando `except` e executar algo, como imprimir o erro (similar ao exemplo). Não se preocupe de entender os detalhes sobre este código e o uso do `try/except` neste momento, teremos um capítulo só para falar deles.

O que é importante é que a maneira *pythônica* de se fazer é assumir a existência do atributo e capturar (tratar) um exceção quando o atributo não pertencer ao objeto e seguir o fluxo do programa. Por ora, faremos esta checagem utilizando a função `hasattr()` .

HERANÇA *VERSUS* ACOPLAMENTO

Note que o uso de herança **aumenta** o acoplamento entre as classes, isto é, o quanto uma classe depende de outra. A relação entre classe mãe e filha é muito forte e isso acaba fazendo com que o programador das classes filhas tenha que conhecer a implementação da classe mãe e vice-versa - fica difícil fazer uma mudança pontual no sistema.

Por exemplo, imagine se tivermos que mudar algo na nossa classe `Funcionario` , mas não quiséssemos que todos os funcionários sofressem a mesma mudança. Precisaríamos passar por cada uma das filhas de `Funcionario` verificando se ela se comporta como deveria ou se devemos sobrescrever o tal método modificado.

Esse é um problema da herança, e não do polimorfismo, que resolveremos mais tarde.

10.7 EXERCÍCIO: HERANÇA E POLIMORFISMO

Vamos ter mais de um tipo de conta no nosso sistema. Portanto, além das informações que já tínhamos na conta, temos agora o tipo: se queremos uma conta corrente ou uma conta poupança. Além disso, cada uma deve possuir uma taxa.

1. Adicione na classe `Conta` um novo método chamado `atualiza()*` que atualiza a conta de acordo com a taxa percentual:

```
class Conta:

    #outros métodos

    def atualiza(self, taxa):
        self._saldo += self._saldo * taxa
```

2. Crie duas subclasses da classe `Conta` : `ContaCorrente` e `ContaPoupanca` . Ambas terão o método `atualiza()` reescrito: a `ContaCorrente` deve atualizar-se com o dobro da taxa e a `ContaPoupanca` deve atualizar-se com o triplo da taxa. Além disso, a `ContaCorrente` deve reescrever o método `deposita()` afim de retirar uma taxa bancária de dez centavos de cada depósito.

   - Crie a classe **ContaCorrente** no arquivo `conta.py` e faça com que ela seja subclasse (filha) da classe `Conta` .

     ```
     class ContaCorrente(Conta):
     ```

```
    pass
```

- Crie a classe **ContaPoupanca** no arquivo `conta.py` e faça com que ela seja subclasse (filha) da classe `Conta` :

  ```
  class ContaPoupanca(Conta):
    pass
  ```

- Reescreva o método `atualiza()` na classe `ContaCorrente` , seguindo o enunciado:

  ```
  class ContaCorrente(Conta):

    def atualiza(self, taxa):
        self._saldo += self._saldo * taxa * 2
  ```

- Reescreva o método `atualiza()` na classe `ContaPopanca` , seguindo o enunciado:

  ```
  class ContaPoupanca(Conta):

    def atualiza(self, taxa):
        self._saldo += self._saldo * taxa * 3
  ```

- Na classe `ContaCorrente` , reescreva o método `deposita()` para descontar a taxa bancária de dez centavos:

  ```
  class Conta Corrente(Conta):

    def atualiza(self, taxa):
        self._saldo += self._saldo * taxa * 2

    def deposita(self, valor):
        self._saldo += valor - 0.10
  ```

3. Agora, teste suas classes no próprio módulo `conta.py` . Acrescente a condição quando o módulo for igual a `__main__` para executarmos no console no PyCharm. Instancie essas classes, atualize-as e veja o resultado:

   ```
   if __name__ == '__main__'
       c = Conta('123-4', 'Joao', 1000.0)
       cc = ContaCorrente('123-5', 'Jose', 1000.0)
       cp = ContaPoupanca('123-6', 'Maria', 1000.0)

       c.atualiza(0.01)
       cc.atualiza(0.01)
       cp.atualiza(0.01)

       print(c.saldo)
       print(cc.saldo)
       print(cp.saldo)
   ```

4. Implemente o método `__str__()` na classe `Conta` . Faça com que ele imprima uma representação mais amigável de um `Conta` contendo todos os seus atributos.

   ```
   def __str__(self):
       # sua implementação aqui
   ```

   Teste chamando o método `print()` passando algumas instâncias de `Conta` como argumento.

5. Vamos criar uma classe que seja responsável por fazer a atualização de todas as contas bancárias e gerar um relatório com o saldo anterior e saldo novo de cada uma das contas. Na pasta `src` crie a classe `AtualizadorDeContas`:

```
class AtualizadorDeContas:

    def __init__(self, selic, saldo_total=0):
        self._selic = selic
        self._saldo_total = saldo_total

    #propriedades

    def roda(self, conta):
        #imprime o saldo anterior, atualiza a conta e depois imprime o saldo final
        #soma o saldo final ao atributo saldo_total
```

   Não esqueça de fazer os *imports* necessário para o código funcionar.

6. No 'main', vamos criar algumas contas e rodá-las a partir do `AtualizadorDeContas`:

```
if __name__ == '__main__':
    c = Conta('123-4', 'Joao', 1000.0)
    cc = ContaCorrente('123-5', 'José', 1000.0)
    cp = ContaPoupanca('123-6', 'Maria', 1000.0)

    adc = AtualizadorDeContas(0.01)

    adc.roda(c)
    adc.roda(cc)
    adc.roda(cp)

    print('Saldo total: {}'.format(adc.saldo_total))
```

7. (opcional) Se você precisasse criar uma classe `ContaInvestimento`, e seu método `atualiza()` fosse complicadíssimo, você precisaria alterar a classe `AtualizadorDeContas`?

8. (opcional, Trabalhoso) Crie uma classe `Banco` que possui uma lista de contas. Repare que em uma lista de contas você pode colocar tanto `ContaCorrente` quanto `ContaPoupanca`. Crie um método `adiciona()` que adiciona uma conta na lista de contas; um método `pegaConta()` que devolve a conta em determinada posição da lista e outro `pegaTotalDeContas()` que retorna o total de contas na lista. Depois teste criando diversas contas, insira-as no `Banco` e depois, com um laço `for`, percorra todas as contas do `Banco` para passá-las como argumento para o método `roda()` do `AtualizadorDeContas`.

9. (opcional) Que maneira poderíamos implementar o método `atualiza()` nas classes `ContaCorrente` e `ContaPoupança` poupando reescrita de código?

10. (opcional) E se criarmos uma classe que não é filha de `Conta` e tentar passar uma instância no método roda de `AtualizadorDeContas`? Com o que aprendemos até aqui, como podemos evitar que erros aconteçam nestes casos?

Seus livros de tecnologia parecem do século passado?

Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.
Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

[Casa do Código, Livros de Tecnologia.]()

10.8 CLASSES ABSTRATAS

Vamos recordar nossa classe `Funcionario`:

```
class Funcionario:

    def __init__(self, nome, cpf, salario=0):
        #inicialização dos atributos

    #propriedades e outros métodos

    def get_bonificacao(self):
        return self._salario * 1.2
```

Considere agora nosso `ControleDeBonificacao`:

```
class ControleDeBonificacoes:

    def __init__(self, total_bonificacoes=0):
        self.__total_bonificacoes = total_bonificacoes

    def registra(self, obj):
        if(hasattr(obj, 'get_bonificacao')):
            self.__total_bonificacoes += obj.get_bonificacao()
        else:
            print('instância de {} não implementa o método get_bonificacao()'.format(self.__class__._
_name__))

    #propriedades
```

Nosso método `registra()` recebe um objeto de qualquer tipo mas estamos esperando que seja um `Funcionario` já que este implementa o método `get_bonificacao()`, isto é, podem ser objetos do tipo `Funcionario` e qualquer de seus subtipos: `Gerente`, `Diretor` e, eventualmente, alguma nova subclasse que venha ser escrita, sem prévio conhecimento do autor da `ControleDeBonificacao`.

Estamos utilizando aqui a classe `Funcionario` para o polimorfismo. Se não fosse ela, teríamos um

grande prejuízo: precisaríamos criar um método `registra()` para receber cada um dos tipos de `Funcionario`, um para `Gerente`, um para `Diretor`, etc. Repare que perder esse poder é muito pior do que a pequena vantagem que a herança traz em herdar código.

Porém, em alguns sistemas, como é o nosso caso, usamos uma classe com apenas esses intuitos: de economizar um pouco código e ganhar polimorfismo para criar métodos mais genéricos, que se encaixem a diversos objetos.

Faz sentido ter um objeto do tipo `Funcionario`? Essa pergunta é bastante relevante já que instanciar um `Funcionario` pode gerar um objeto que não faz sentido no nosso sistema. Nossa empresa tem apenas `Diretores`, `Gerentes`, `Secretárias`, etc... `Funcionario` é apenas uma classe que idealiza um tipo, define apenas um rascunho.

Vejamos um outro caso em que não faz sentido ter um objeto de determinado tipo, apesar da classe existir. Imagine a classe `Pessoa` e duas filhas: `PessoaFisica` e `PessoaJuridica`. Quando puxamos um relatório de nossos clientes (uma lista de objetos de tipo `Pessoa`, por exemplo), queremos que cada um deles seja ou uma `PessoaFisica` ou uma `PessoaJuridica`. A classe `Pessoa`, nesse caso, estaria sendo usada apenas para ganhar o polimorfismo e herdar algumas coisas - não faz sentido permitir instanciá-la.

Para o nosso sistema, é inadmissível que um objeto seja apenas do tipo `Funcionario` (pode existir um sistema em que faça sentido ter objetos do tipo `Funcionario` ou apenas `Pessoa`, mas, no nosso caso, não). Para resolver esses problemas, temos as **classes abstratas**.

Utilizaremos uma módulo do Python chamado **abc** que permite definirmos classes abstratas. Uma classe abstrata deve herdar de **ABC** (*Abstract Base Classes*). ABC é a superclasse para classes abstratas.

Uma classe abstrata não pode ser instanciada e deve conter pelo menos um método abstrato. Vamos ver isso na prática.

Vamos tornar nossa classe `Funcionario` abstrata:

```
import abc

class Funcionario(abc.ABC):

    # métodos e propriedades
```

Definimos nossa classe `Funcionario` como abstrata. Agora vamos tornar nosso método `get_bonificacao()` abstrato. Um método abstrato pode ter implementação, mas não faz sentido em nosso sistema, portanto vamos deixá-lo sem implementação. Para definir um método abstrato utilizamos o decorator `@abstractmethod`:

```
class Funcionario(abc.ABC):

    @abc.abstractmethod
    def get_bonificacao(self):
```

```
        pass
```

Agora, se tentarmos instanciar um objeto do tipo `Funcionario` :

```
if __name__ == '__main__':
    f = Funcionario()
```

Acusa um erro:

```
TypeError: Can't instantiate abstract class Funcionario with abstract methods get_bonificacao
```

Apesar de não conseguir instanciar a classe `Funcionario` , conseguimos instanciar suas filhas que são objetos que realmente existem em nosso sistema (objetos concretos):

```
class Gerente(Funcionario):
    # outros métodos e propriedades

    def get_bonificacao(self):
        return self._salario * 0.15

if __name__ == '__main__':
    gerente = Gerente('jose', '222222222-22', 5000.0, '1234', 0)
    print(gerente.get_bonificacao())
```

Vamos criar a classe `Diretor` que herda de `Fucionario` sem o método `get_bonificacao()` :

```
class Diretor(Funcionario):
    def __init__(self, nome, cpf, salario):
        super().__init__(nome, cpf, salario)

if __name__ == '__main__':
    diretor = Diretor('joao', '111111111-11', 4000.0)
```

Quando rodamos o código:

```
TypeError: Can't instantiate abstract class Diretor with abstract methods get_bonificacao
```

Não conseguimos instanciar uma subclasse de `Funcionario` sem implementar o método abstrato `get_bonificacao()` . Agora tornamos o método `get_bonificacao()` **obrigatório** para todo objeto que é subclasse de `Funcionario` . Caso venhamos a criar outras classes, como `Secretaria` e `Presidente` , que sejam filhas de `Funcionario` , seremos obrigados e criar o método `get_bonificacao()` , caso contrário, o código vai acursar erro quando executado.

10.9 EXERCÍCIOS - CLASSES ABSTRATAS

1. Torne a classe `Conta` abstrata.

```
import abc

class Conta(abc.ABC):

    def __init__(self, numero, titular, saldo=0, limite=1000.0):
        self._numero = numero
        self._titular = titular
        self._saldo = saldo
        self._limite = limite
```

```python
    # outros métodos e propriedades
```

2. Torne o método `atualiza()` abstrato:

```python
class Conta(abc.ABC):

    # código omitido

    @abc.abstractmethod
    def atualiza():
        pass
```

3. Tente instância uma `Conta`:

```python
if __name__ == '__main__':
    c = Conta()
```

   O que acontece?

4. Instancie uma `ContaCorrente` e uma `ContaPoupanca` e teste o código chamando o método `atualiza()`.

```python
if __name__ == '__main__':
    cc = ContaCorrente('123-4', 'João', 1000.0)
    cp = ContaPoupanca('123-5', 'José', 1000.0)

    cc.atualiza(0.01)
    cp.atualiza(0.01)

    print(cc.saldo)
    print(cp.saldo)
```

5. Crie uma classe chamada `ContaInvestimento`:

```python
class ContaInvestimeto(Conta):
    pass
```

6. Instancie uma `ContaInvestimeto`:

   ` python ci = ContaInvestimento('123-6', 'Maria', 1000.0) ```

7. Não conseguimos instanciar uma `ContaInvestimento` que herda `Conta` sem implementar o método abstrato `atualiza()`. Vamos criar uma implementação dentro da classe `ContaInvestimento`:

```python
def atualiza(self, taxa):
    self._saldo += self._saldo * taxa * 5
```

8. Agora teste instanciando uma `ContaInvestimento` e chame o método `atualiza()`:

```python
ci = ContaInvestimento('123-6', 'Maria', 1000)
ci.deposita(1000.0)
ci.atualiza(0.01)
print(ci.saldo)
```

9. (opcional) Crie um atributo `tipo` nas classes `ContaCorrente`, `ContaPoupanca` e

`ContaInvestimento` . Faça com que o `tipo` também seja impresso quando usamos a função `print()` .

CAPÍTULO 11

HERANÇA MÚLTIPLA E INTERFACES

Imagine que um Sistema de Controle do Banco pode ser acessado, além dos Gerentes, pelos Diretores do Banco. Teríamos uma classe `Diretor`.

```
class Diretor(Funcionario):

    def autentica(self, senha):
        # verifica se a senah confere
```

E a classe `Gerente`:

```
class Gerente(Funcionario):

    def autentica(self, senha):
        # verifica se a senha confere e também se o seu departamento tem acesso
```

Repare que o método de autenticação de cada tipo de `Funcionario` pode variar muito. Mas vamos aos problemas. Considere o `SistemaInterno` e seu controle: precisamos receber um `Diretor` ou `Gerente` como argumento, verificar se ele se autentica e colocá-lo dentro do sistema.

Vimos que podemos utilizar a função `hasattr()` para verificar se um objeto possui o método `autentica()`:

```
class SistemaInterno:

    def login(self, funcionario):
        if(hasattr(obj, 'autentica')):
            # chama método autentica
        else:
            # imprime mensagem de ação inválida
```

Mas podemos esquecer, no futuro, quando modelar a classe `Presidente` (que também é um funcionário e autenticável), de implementar o método `autentica()`. Não faz sentido colocarmos o método `autentica()` na classe `Funcionario` já que nem todo funcionário é autenticável.

Uma solução mais interessante seria criar uma classe no meio da árvore de herança, a `FuncionarioAutenticavel`:

```
class FuncionarioAutenticavel(Funcionario):

    def autentica(self, senha):
        # verifica se a senha confere
```

E as classes `Diretor`, `Gerente` e qualquer outro tipo de `FuncionarioAutenticavel` que vier a

existir em nosso sistema bancário passaria a estender de `FuncionarioAutenticavel`. Repare que `FuncionarioAutenticavel` é forte candidata a classe abstrata. Mais ainda, o método `autentica()` poderia ser um método abstrato.

O uso de herança simples resolve o caso, mas vamos a uma outra situação um pouco mais complexa: todos os clientes também devem possuir acesso ao `SistemaInterno`. O que fazer?

Uma opção é fazer uma herança sem sentido para resolver o problema, por exemplo, fazer `Cliente` estender de `FuncionarioAutenticavel`. Realmente resolve o problema, mas trará diversos outros. `Cliente` definitivamente não é um `FuncionarioAutenticavel`. Se você fizer isso, o `Cliente` terá, por exemplo, um método `get_bonificacao()`, um atributo `salario` e outros membros que não fazem o menor sentido para esta classe.

Precisamos, para resolver este problema, arranjar uma forma de referenciar `Diretor`, `Gerente` e `Cliente` de uma mesma maneira, isto é, achar um fator comum.

Se existisse uma forma na qual essas classes garantissem a existência de um determinado método, através de um contrato, resolveríamos o problema. Podemos criar um "contrato" que define tudo o que uma classe deve fazer se quiser ter um determinado status. Imagine:

```
contrato Autenticavel

    - quem quiser ser Autenticavel precisa saber fazer:
        autenticar dada uma senha, devolvendo um booleano
```

Quem quiser pode assinar este contrato, sendo assim obrigado a explicar como será feita essa autenticação. A vantagem é que, se um `Gerente` assinar esse contrato, podemos nos referenciar a um `Gerente` como um `Autenticavel`.

Como Python admite **herança múltipla** podemos criar a classe `Autenticavel`:

```
class Autenticavel:

    def autentica(self, senha):
        # verifica se a senha confere
```

E fazer `Gerente`, `Diretor` e `Cliente` herdarem essa classe:

```
class Gerente(Funcionario, Autenticavel):
    # código omitido

class Diretor(Funcionario, Autenticavel):
    # código omitido

class Cliente(Autenticavel):
    # código omitido
```

Ou seja, `Gerente` e `Diretor` além de funcionários são autenticáveis! Assim, podemos utilizar o `SistemaInterno` para funcionários autenticáveis e clientes:

```
class SistemaInterno:
```

```
    def login(self, obj):
        if(hasattr(obj, 'autentica')):
            obj.autentica()
            return True
        else:
            print('{} não é autenticável'.format(self.__class__.__name__))
            return False

if __name__ == '__main__':
    diretor = Diretor('João', '111111111-11', 3000.0, '1234')
    gerente = Gerente('José', '222222222-22', 5000.0, '1235')
    cliente = Cliente('Maria', '333333333-33', '1236')

    sistema = SistemaInterno()
    sistema.login(diretor)
    sistema.login(gerente)
    sistema.login(cliente)
```

Note que uma classe pode herdar de muitas outras classes. Mas vamos aos problemas que isso pode gerar. Por exemplo, várias classes podem possuir o mesmo método.

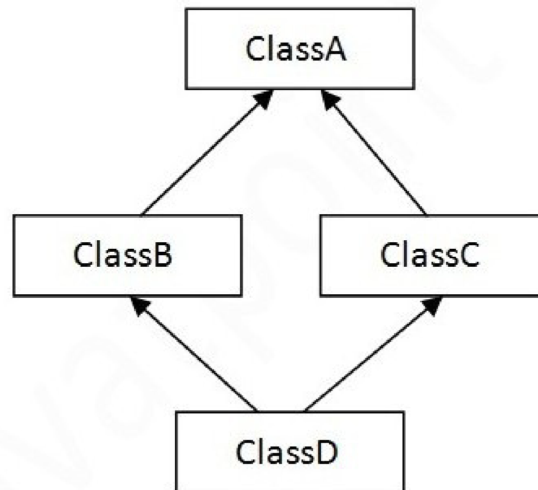
11.1 PROBLEMA DO DIAMANTE

O exemplo anterior pode parecer uma boa maneira de representar classes autenticáveis, mas se começássemos a estender esse sistema, logo encontraríamos algumas complicações. Em um banco de verdade, as divisões entre gerentes, diretores e clientes nem sempre são claras. Um `Cliente` , por exemplo, pode ser um `Funcionario` , um `Funcionario` pode ter outras subcategorias como fixos e temporários.

No Python, é possível que uma classe herde de várias outras classes. Poderíamos, por exemplo, criar uma classe `A` , que será superclasse das classes `B` e `C` . A herança múltipla não é muito difícil de entender se uma classe herda de várias classes que possuem propriedades completamente diferentes, mas as coisas ficam complicadas se duas superclasses implementam o mesmo método ou atributo.

Se as classes `B` e `C` herdarem a classe `A` e classe `D` herdar as classes `B` e `C` , e as classes `B` e `C` têm um método `m2()` , qual método a classe `D` herda?

```
class A:
    def m1(self):
        print('método de A')

class B(A):
    def m2(self):
        print('método de B')

class C(A):
    def m2(self):
        print('método de C')

class D(B, C):
    pass
```



Essa ambiguidade é conhecida como o problema do diamante, ou problema do losango, e diferentes linguagens resolvem esse problema de maneiras diferentes. O Python segue uma ordem específica para percorrer a hierarquia de classes e essa ordem é chamada de MRO: *Method Resolution Order* (Ordem de Resolução de Métodos).

Toda classe tem um atributo `__mro__` que retorna uma tupla de referências das superclasses na ordem MRO - da classe atual até a classe `object`. Vejamos o MRO da classe `D`:

```
print(D.mro())
```

Saída:

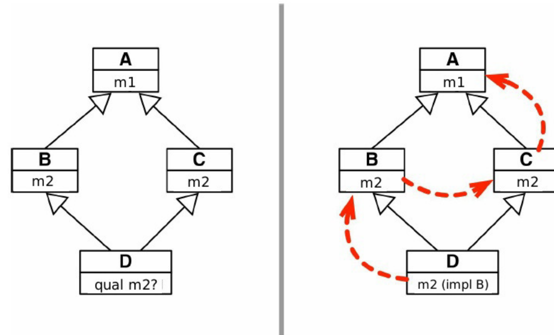
```
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class 'obje
ct'>)
```

A ordem é sempre da esquerda para direita. Repare que o Python vai procurar a chamada do método `m2()` primeiro na classe `D`, não encontrando vai procurar em `B` (a primeira classe herdada). Caso não encontre em `B`, vai procurar em `C` e só então procurar em `A` - e por último na classe `object`.

Também podemos acessar o atributo `__mro__` através do método `mro()` chamado pela classe que retorna uma lista ao invés de uma tupla:

```
print(D.mro())

saída:
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class 'obje
ct'>]
```

Portanto, seguindo o MRO, a classe `D` chama o método `m2()` da classe `B`:




```
d = D()
d.m1()
d.m2()
```

Saída:

```
método de A
método de B
```

Felizmente, a função `super()` sabe como lidar de forma inteligente com herança múltipla. Se usá-la dentro do método todos os métodos das superclasses devem ser chamados seguindo o MRO.

```
class A:
    def m1(self):
        print('método de A')

class B(A):

    def m1(self):
        super().m1()

    def m2(self):
        print('método de B')

class C(A):
    def m1(self):
        super().m1()

    def m2(self):
        print('método de C')

class D(B, C):
    def m1(self):
        super().m1()

    def m2(self):
        super().m2()

if __name__ == '__main__':
    d = D()
    d.m1()
    d.m2()
```

Gera a saída:

```
método de A
método de B
```

Você pode também fazer o curso PY-14 dessa apostila na Caelum

Querendo aprender ainda mais sobre? Esclarecer dúvidas dos exercícios? Ouvir explicações detalhadas com um instrutor?
A Caelum oferece o **curso PY-14** presencial nas cidades de São Paulo, Rio de Janeiro e Brasília, além de turmas incompany.

Consulte as vantagens do curso *Python e Orientação a Objetos*

11.2 MIX-INS

Se usarmos herança múltipla, geralmente é uma boa idéia projetarmos nossas classes de uma maneira que evite o tipo de ambiguidade descrita acima - apesar do Python possuir o MRO, em sistemas grandes a herança múltipla ainda pode causar muitos problemas.

Uma maneira de fazer isso é dividir a funcionalidade opcional em *mix-ins*. Um *mix-in* é uma classe que não se destina a ser independente - existe para adicionar funcionalidade extra a outra classe através de herança múltipla. A ideia é que classes herdem estes *mix-ins*, essas "misturas de funcionalidades".

Por exemplo, nossa classe `Autenticavel` pode ser um *mix-in* já que ela existe apenas para acrescentar a funcionalidade de ser *autenticável*, ou seja, para herdar seu método `autentica` .

Nossa classe `Autenticavel` já se comporta como um `Mix-In` . No Python não existe uma maneira específica de criar *mix-ins*. Os programadores, por convenção e para deixar explícito a classe como um *mix-in*, colocam o termo *'MixIn'* no nome da classe e utilizam através de herança múltipla:

```
class AutenticavelMixIn:
    def autentica(self, senha):
        # verifica senha
```

Cada *mix-in* é responsável por fornecer uma peça específica de funcionalidade opcional. Podemos ter outros *mix-ins* no nosso sistema:

```
class AtendimentoMixIn:
    def cadastra_atendimento(self):
        # faz cadastro atendimento

    def atende_cliente(self):
        # faz atendimento

class HoraExtraMixIn:

    def calcula_hora_extra(self, horas):
        # calcula horas extras
```

E podemos misturá-los nas classes de nosso sistema:

```
class Gerente(Funcionario, AutenticavelMixIn, HoraExtraMixIn):
    pass

class Diretor(Funcionario, AutenticavelMixIn):
    pass

class Cliente(AutentivavelMixIn):
    pass

class Escriturario(Funcionario, AtentimentoMixIn):
    pass
```

Repare que nossos *mix-ins* não tem um método `__init__()` . Muitos *mix-ins* apenas fornecem métodos adicionais mas não inicializam nada. Isso às vezes significa que eles dependem de outras propriedades que já existem em suas filhas. Cada *mix-in* é responsável por fornecer uma peça específica de funcionalidade opcional - é um jeito de compor classes.

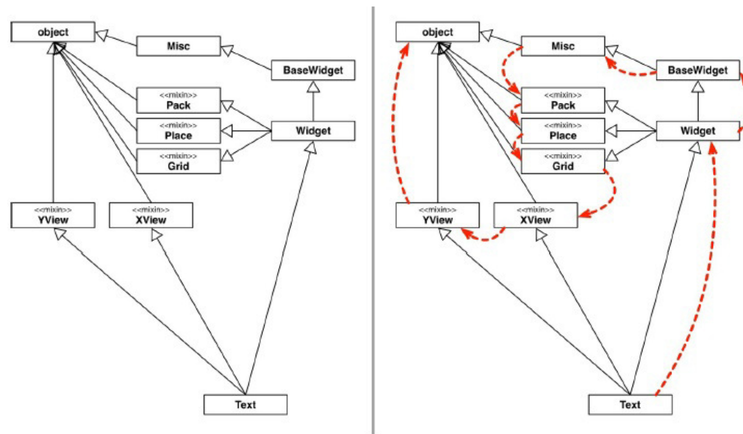
Poderíamos estender este exemplo com mais misturas que representam a capacidade de pagar taxas, a capacidade de ser pago por serviços, e assim por diante - poderíamos então criar uma hierarquia de classes relativamente plana para diferentes tipos de classes de funcionário que herdam `Funcionario` e alguns *mix-ins* .

Essa é uma das abordagens de se usar herança múltipla mas ela é bastante desencorajada. Caso você utilize, opte por *Mix Ins* sabendo de suas desvantagens. Usado em sistemas grandes podem ocorrer colisões com nomes de métodos, métodos substituídos acidentalmente, hierarquia de classe pouco clara e dificuldade de ler e entender classes compostas por muitos *mix-ins*, dentre outras desvantagens. O problema da herança múltipla permanece.

Outra abordagem possível é definir funções fora de classes, digamos em um módulo e fazer chamadas dessas funções passando nossos objetos. Mas isso é um afastamento radical do paradigma orientado a objetos que é baseada em métodos definidos dentro das classes.

11.3 PARA SABE MAIS - TKINTER

Tkinter é um *framework* que faz parte da biblioteca padrão do Python utilizado para criar interface gráfica. É um caso onde *mix-ins* trabalham bem já que se trata de um pequeno *framework*, mas também é suficientemente grande para que seja possível ver o problema. Veja um exemplo de parte de sua hierarquia de classe:



Essa figura mostra parte do complicado modelo de classes utilizando herança múltipla do pacote `Tkinter` . A setas representam o MRO que deve iniciar na classe `Text` . A classe `Text` implementa um campo de texto editável e tem muitas funcionalidades próprias, além de herdar muitos métodos de outras classes.

Uma outra classe do pacote que não aparece neste diagrama é a `Label` , utilizada para mostrar um texto ou *bitmap* na tela. Você pode testar no Pycharm, aproveitando a ferramenta de *autocomplete*, chamando `Tkinter.Label.` e a IDE vai te mostrar 181 sugestões de atributos em uma única classe! Ou você pode utilizar a função `help()` para checar a origem de cada um deles.

```
from tkinter import *

help(Label)
```

Esse pacote tem mais de 20 anos e é um exemplo de como a herança múltipla era utilizada quando os programadores não consideravam suas desvantagens. Apesar da maioria das classes se comportarem como *mix-ins*, o padrão de nomenclatura não era utilizado. Felizmente, o `Tkinter` é um *framework* estável.

11.4 (OPCIONAL) EXERCÍCIOS - MIX-INS

1. Nosso banco precisa tributar dinheiro de alguns bens que nossos clientes possuem. Para isso vamos criar uma classe `Tributavel` :

   ```
   class Tributavel:

       def get_valor_imposto(self):
           pass
   ```

   Lemos essa classe da seguinte maneira: "Todos que quiserem ser *tributável* precisam saber retornar o valor do imposto". Alguns bens são tributáveis e outros não, `ContaPoupanca` não é tributável, já para `ContaCorrente` você precisa pagar 1% da conta e o `SeguroDeVida` tem uma faixa fixa de 50 reais mais 5%. do valor do seguro.

2. Torne a classe `Tributavel` um *mix-in*:

```
class TributavelMixIn:

    def get_valor_imposto(self)
        pass
```

3. Faça a classe `ContaCorrente` herdar da classe `TributavelMixIn`. Crie a classe `SeguroDeVida`:

```
class ContaCorrente(Conta, TributavelMixIn):
    # código omitido

    def get_valor_imposto(self):
        return self._saldo * 0.01

class SeguroDeVida(TributavelMixIn):
    def __init__(self, valor, titular, numero_apolice):
        self._valor = valor
        self._titular = titular
        self._numero_apolice = numero_apolice

    def get_valor_imposto(self):
        return 42 + self._valor * 0.05
```

4. Vamos criar a classe `ManipuladorDeTributaveis` em um arquivo chamado *manipulador.py*. Essa classe deve ter um método chamado `calcula_imposto()` que recebe umas lista de tributáveis e retorna o total de impostos cobrados:

```
class ManipuladorDeTributaveis:

    def calcula_impostos(self, lista_tributaveis):
        total = 0
        for t in lista_tributaveis:
            total += t.get_valor_imposto()

        return total
```

5. Ainda no arquivo `manipulador.py`, vamos testar o código. Crie alguns objetos de `ContaCorrente` e de `SeguroDeVida`. Em seguida, crie uma lista de tributáveis e insira seus objetos nela. Instancie um `ManipuladorDeTributaveis` e chame o método `calcula_impostos()` passando a lista de tributáveis criada e imprima o valor total dos impostos:

```
if __name__ == '__main__':
    from conta import ContaCorrente, SeguroDeVida, TributavelMixIn

    cc1 = ContaCorrente('123-4', 'João', 1000.0)
    cc2 = ContaCorrente('123-4', 'José', 1000.0)
    seguro1 = SeguroDeVida(100.0, 'José', '345-77')
    seguro2 = SeguroDeVida(200.0, 'Maria', '237-98')

    lista_tributaveis = []
    lista_tributaveis.append(cc1)
    lista_tributaveis.append(cc2)
    lista_tributaveis.append(seguro1)
    lista_tributaveis.append(seguro2)

    manipulador = ManipuladorDeTributaveis()
    total = manipulador.calcula_impostos(lista_tributaveis)
```

```
    print(total)
```

Vimos que herança múltipla pode ser perigosa e se nosso sistema crescer pode gerar muita confusão e conflito de nomes de métodos. Uma maneira mais eficaz nestes casos é usar classes abstratas como interfaces que veremos a seguir.

Seus livros de tecnologia parecem do século passado?

Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.
Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

Casa do Código, Livros de Tecnologia.

11.5 INTEFACES

O Python não possui uma palavra reservada **interface**. Mesmo sem uma palavra reservada para interface toda classe tem uma interface. São os atributos públicos definidos (que em Python são tanto atributos quanto métodos) em uma classe - isso inclui os métodos especiais como `__str__()` e `__add__()` .

Uma interface vista como um conjunto de métodos para desempenhar um papel é o que os programadores da *SmallTalk* chamavam de **protocolo** e este termo foi disseminado em comunidades de programadores de linguagens dinâmicas. Esse protocolo funciona como um contrato.

Os protocolos são independentes de herança. Uma classe pode implementar vários protocolos, como os *mix-ins*. Protocolos são interfaces e são definidos apenas por documentação e convenções em linguagens dinâmicas, por isso são considerados informais. Os protocolos não podem ser verificados estaticamente pelo interpretador.

O método `__str__()` , por exemplo, é esperado que retorne uma representação do objeto em forma de `string` . Nada impede de fazermos outras coisas dentro do método como deletar algum conteúdo, fazer algum cálculo, etc... ao invés de retornarmos apenas a `string` . Mas há um entendimento prévio comum do que este método deve fazer e está presente na **documentação** do Python. Este é um exemplo onde o contrato semântico é descrito em um manual. Algumas linguagens de tipagem estática, como Java, possuem interfaces em sua biblioteca padrão e podem garantir este contrato em tempo de

compilação.

A partir do Python 2.6 a definição de interfaces utilizando o módulo ABC é uma solução mais elegante do que os *mix-ins*. Nossa classe `Autenticavel` pode ser uma classe abstrata com o método abstrato `autentica()`:

```
import abc

class Autenticavel(abc.ABC):

    @abc.abstractmethod
    def autentica(self, senha):
        pass
```

Como se trata de uma interface em uma linguagem de tipagem dinâmica como o Python, a boa prática é documentar esta classe garantindo o contrato semântico:

```
import abc

class Autenticavel(abc.ABC):
    """Classe abstrata que contém operações de um objeto autenticável.

    As subclasses concretas devem sobrescrever o método autentica
    """

    @abc.abstractmethod
    def autentica(self, senha):
        """ Método abstrato que faz verificação da senha
        return True se a senha confere, e False caso contrário.
        """
```

E nossas classes `Gerente`, `Diretor` e `Cliente` herdariam a classe `Autenticavel`. Mas qual a diferença de herdar muitos *mix-ins* e muitas ABCs? Realmente, aqui não há grande diferença e voltamos ao problema anterior dos *mix-ins* - muito acoplamento entre classes que gera a herança múltipla.

Mas a novidade das ABCs é seu método `register()`. As ABCs introduzem uma subclasse virtual, que são classes que não herdam de uma classe mas são reconhecidas pelos métodos `isinstance()` e `issubclass()`. Ou seja, nosso `Gerente` não precisa herdar a classe `Autenticavel`, basta registrarmos ele como uma implementação da classe `Autenticavel`.

```
Autenticavel.register(Gerente)
```

E testamos os métodos `isinstance()` e `issubclass()` com uma instância de `Gerente`:

```
gerente = Gerente('João', '111111111-11', 3000.0)
print(isinstance(Autenticavel))
print(issubclass(Autenticavel))
```

que vai gerar a saída:

```
True
True
```

O Python não vai verificar se existe uma implementação do método `autentica` em `Gerente`

quando registrarmos a classe. Ao registrarmos a classe `Gerente` como uma `Autenticavel` , prometemos ao Python que a classe implementa fielmente a nossa interface `Autenticavel` definida. O Python vai acreditar nisso e retornar **True** quando os métodos `isinstance()` e `issubclass()` forem chamados. Se mentirmos, ou seja, não implementarmos o método `autentica()` em `Gerente` , uma exceção será lançada quando tentarmos chamar este método.

Vejamos um exemplo com a classe `Diretor` . Não vamos implementar o método `autentica()` e registrar uma instância de `Diretor` como um `Autenticavel` :

```
class Diretor(Funcionario):
    # código omitido

if __name__ == '__main__':
    Autenticavel.register(Diretor)

    d = Diretor('José', '22222222-22', 3000.0)

    d.autentica('?')
```

E temos como saída:

```
Traceback (most recent call last):
  File <stdin>, line 47, in <module>
    d.autentica('?')
AttributeError: 'Diretor' object has no attribute 'autentica'
```

Novamente, podemos tratar a exceção ou utilizar os métodos `isinstance()` ou `issubclass()` para verificação. Apesar de considerada má práticas por muitos pythonistas, o módulo de classes abstratas justifica a utilização deste tipo de verificação. A verificação não é de tipagem, mas se um objeto está de acordo com a interface:

```
if __name__ == '__main__':
    Autenticavel.register(Diretor)

    d = Diretor('José', '22222222-22', 3000.0)

    if(isinstance(d, Autenticavel)):
        d.autentica('?')
    else:
        print("Diretor não implementa a interface Autenticavel")
```

e portanto, nossa classe `SistemaInterno` ficaria assim:

```
from autenticavel import Autenticavel

class SistemaInterno:

    def login(self, obj):
        if(isinstance(obj, Autenticavel)):
            obj.autentica(obj.senha)
            return True
        else:
            print("{} não é autenticável".format(self.__class__.__name__))
            return False
```

Dessa maneira fugimos da herança múltipla e garantimos um contrato, um protocolo. Classes

abstratas complementam o *duck typing* provendo uma maneira de definir interfaces quando técnicas como usar `hasattr()` são ruins ou sutilmente erradas. Você pode ler mais a respeito no documento da PEP que introduz classes abstratas - é a PEP 3119 e você pode acessar seu conteúdo neste link: https://www.python.org/dev/peps/pep-3119/

Algumas ABCs também podem prover métodos concretos, ou seja, não abstratos. Por exemplo, a classe `Interator` do módulo `collections` da biblioteca padrão do Python possui um método `__iter__()` retornando ele mesmo. Esta ABC pode ser considerada uma classe *mix-in*.

O Python já vem com algumas estruturas abstratas (ver módulos collections, numbers e io).

11.6 EXERCÍCIOS - INTERFACES E CLASSES ABSTRATAS

1. Nosso banco precisa tributar dinheiro de alguns bens que nossos clientes possuem. Para isso vamos criar uma classe `Tributavel` no módulo *tributavel.py*:

```
class Tributavel:

    def get_valor_imposto(self):
        pass
```

   Lemos essa classe da seguinte maneira: "Todos que quiserem ser *tributável* precisam saber retornar o valor do imposto". Alguns bens são tributáveis e outros não, `ContaPoupanca` não é tributável, já para `ContaCorrente` você precisa pagar 1% da conta e o `SeguroDeVida` tem uma faixa fixa de 50 reais mais 5% do valor do seguro.

2. Torne a classe `Tributavel` uma classe abstrata:

```
import abc

class Tributavel(abc.ABC):

    def get_valor_imposto(self)
        pass
```

3. O método `get_valor_imposto()` também deve ser abstrato:

```
import abc

class Tributavel(abc.ABC):

    @abc.abstractmethod
    def get_valor_imposto(self, valor):
        pass
```

4. Nada impede que os usuários de nossa classe tributavel implemente o método `get_valor_imposto` de maneira não esperada por nós. Então vamos acrescentar a documentação utilizando *docstring* que aprendemos no capítulo de módulos:

```
import abc
```

```
class Tributavel(abc.ABC):
    """ Classe que contém operações de um objeto autenticável

    As subclasses concretas devem sobrescrever o método get_valor_imposto.
    """
    @abc.abstractmethod
    def get_valor_imposto(self):
        """ aplica taxa de imposto sobre um determinado valor do objeto """
        pass
```

5. Utiliza a função `help()` passando a classe `Tributavel` para acessar a documentação.

6. Faça a classe `ContaCorrente` herdar da classe `Tributavel`. Crie a classe `SeguroDeVida` com os atributos `valor`, `titular` e `numero_apolice` que também deve ser um tributável. Implemente o método `get_valor_imposto()` de acordo com a regra de negócio definida pelo exercício:

```
class ContaCorrente(Conta, Tributavel):
    # código omitido

    def get_valor_imposto(self):
        return self._saldo * 0.01

class SeguroDeVida(Tributavel):
    def __init__(self, valor, titular, numero_apolice):
        self._valor = valor
        self._titular = titular
        self._numero_apolice = numero_apolice

    def get_valor_imposto(self):
        return 50 + self._valor * 0.05
```

7. Vamos criar a classe `ManipuladorDetributaveis` em um arquivo chamado *manipulador_tributaveis.py*. Essa classe deve ter um método chamado `calcula_imposto()` que recebe umas lista de tributáveis e retorna o total de impostos cobrados:

```
class ManipuladorDeTributaveis:

    def calcula_impostos(self, lista_tributaveis):
        total = 0
        for t in lista_tributaveis:
            total += t.get_valor_imposto()
        return total
```

8. Nossas classes `ContaCorrente` e `SeguraDeVida` já implementam o método `get_valor_imposto()`. Vamos instanciar cada umas delas e testar a chamada do método:

```
if __name__ == '__main__':
    cc = ContaCorrente('123-4', 'João', 1000.0)
    seguro = SeguroDeVida(100.0, 'José', '345-77')

    print(cc.get_valor_imposto())
    print(seguro.get_valor_imposto())
```

9. Crie uma lista com os objetos criados no exercício anterior, instancie um objeto do tipo `list` e passe a lista chamando o método `calcula_impostos()`.

```
if __name__ == '__main__':
```

```
    # código omitido

    lista_tributaveis = []
    lista_tributaveis.append(cc)
    lista_tributaveis.append(seguro)

    mt = ManipuladorDeTributaveis()
    total = mt.calcula_impostos(lista_tributaveis)
    print(total)
```

10. Nosso código funciona, mas ainda estamos utilizando herança múltipla! Vamos melhorar nosso código. Faça com que `ContaCorrente` e `SeguroDeVida` não mais herdem da classe `Tributavel`. Vamos registrar nossas classes `ContaCorrente` e `SeguroDeVida` como subclasses virtuais de `Tributavel`, de modo que funcione como uma **interface**.

```
class ContaCorrente(Conta):
    # código omitido

class SeguroDeVida:
    # código omitido

if __name__ == '__main__':
    from tributavel import Tributavel

    cc = ContaCorrente('João', '123-4')
    cc.deposita(1000.0)

    seguro = SeguroDeVida(100.0, 'José', '345-77')

    Tributavel.register(ContaCorrente)
    Tributavel.register(SeguroDeVida)

    lista_tributaveis = []
    lista_tributaveis.append(cc)
    lista_tributaveis.append(seguro)

    mt = ManipuladorDeTributaveis()
    total = mt.calcula_impostos(lista_tributaveis)
    print(total)
```

11. Modifique o método `calcula_impostos()` da classe `ManipuladorDeTributaveis` para checar se os elementos da listas são tributáveis através do método `isinstance()`. Caso um objeto da lista não seja um tributável, vamos imprimir uma mensagem de erro e apenas os tributáveis serão somados ao total:

```
class ManipuladorDeTributaveis:

    def calcula_impostos(self, lista_tributaveis):
        total = 0
        for t in lista_tributaveis:
            if(isinstance(t, Tributavel)):
                total += t.get_valor_imposto()
            else:
                print(t.__repr__(), "não é um tributável")
        return total
```

Teste novamente com a lista de tributáveis que fizemos no exercício anterior e veja se tudo continua

funcionando.

12. `ContaPoupanca` não é um tributável. Experimente instanciar uma `ContaPoupanca` , adicionar a lista de tributáveis e calcular o total de impostos através do `ManipuladorDetributaveis` :

```
if __name__ == '__main__':
    #código omitido do exercício anterior omitido

    cp = ContaPoupanca('123-6', 'Maria')
    lista_tributaveis.append(cp)

    total = mt.calcula_impostos(lista_tributaveis)
    print(total)
```

O que acontece?

13. (Opcional) Agora além de `ContaCorrente` e `SeguroDeVida` nossa `ContaInvestimento` também deve ser um tributável, cobrando 3% do saldo. Instancie uma `ContaInvestimento` e registre a classe `ContaInvestimento` como tributável. Adicione a `ContaInvestimento` criada na lista de tributáveis do exercício anterior e calcule o total de impostos através do `ManipuladorDeTributaveis` .

Neste capítulo aprendemos sobre herança múltipla e suas desvantagens mesmo utilizando *mix-ins*. Aprendemos utilizar classes abstratas como interfaces registrando as classes e evitando os problemas com a herança múltipla. Agora nossa classe abstrata `Tributavel` funciona como um protocolo. No capítulo sobre o módulo collections veremos na prática alguns conceitos vistos nestes capítulo.

CAPÍTULO 12

EXCEÇÕES E ERROS

Voltando as contas que criamos no capítulo 6, o que aconteceria ao tentar chamar o método `saca()` com um valor fora do limite? O sistema mostraria uma mensagem de erro, mas quem chamou o método `saca()` não saberá que isso aconteceu.

Como avisar aquele que chamou o método de que ele não conseguiu fazer aquilo que deveria?

Os métodos dizem qual o contrato que eles devem seguir. Se, ao tentar `sacar()`, ele não consegue fazer o que deveria, ele precisa, ao menos, avisar ao usuário que o saque não foi feito.

Veja no exemplo abaixo: estamos forçando uma `Conta` a ter um valor negativo, isto é, estar em um estado inconsistente de acordo com a nossa modelagem.

```
conta = Conta('123-4', 'João')
conta.deposita(100.0)
conta.saca(3000.0)

#o método saca funcionou?
```

Em sistemas de verdade, é muito comum que quem saiba tratar o erro é aquele que chamou o método e não a própria classe! Portanto, nada mais natural sinalizar que um erro ocorreu.

A solução mais simples utilizada antigamente é a de marcar o retorno de um método como boolean e retornar `True`, se tudo ocorreu da maneira planejada, ou `False`, caso contrário:

```
if (valor > self.saldo + self.limite):
    print("nao posso sacar fora do limite")
    return False
else:
    self.saldo -= valor
    return True
```

Um novo exemplo de chamada do método acima:

```
conta = Conta('123-4', 'João')
conta.deposita(100.0)
conta.limite = 100.0

if(not conta.saca(3000.0)):
    print("nao saquei")
```

Repare que tivemos de lembrar de testar o retorno do método, mas não somos obrigados a fazer isso. Esquecer de testar o retorno desse método teria consequências drásticas: a máquina de autoatendimento poderia vir a liberar a quantia desejada de dinheiro, mesmo se o sistema não tivesse conseguido efetuar o

método `saca()` com sucesso, como no exemplo a seguir:

```
conta = Conta("123-4", "João")
conta.deposita(100.0)

# ...

valor = 5000.0
conta.saca(valor) # vai retornar False, mas ninguém verifica

caixa_eletronico.emite(valor)
```

Mesmo invocando o método e tratando o retorno de maneira correta, o que faríamos se fosse necessário sinalizar quando o usuário passou um valor negativo como *valor*. Uma solução seria alterar o retorno de `boolean` para `int` e retornar o código do erro que ocorreu. Isso é considerado uma má prática (conhecida também como uso de "magic numbers").

Além de você perder o retorno do método, o valor devolvido é "mágico" e só legível perante extensa documentação, além de não obrigar o programador a tratar esse retorno e, no caso de esquecer isso, seu programa continuará rodando já num estado inconsistente.

Por esses e outro motivos, utilizamos um código diferente para tratar aquilo que chamamos de exceções: os casos onde acontece algo que, normalmente, não iria acontecer. O exemplo do argumento do saque inválido ou do id inválido de um cliente é uma exceção à regra.

Uma exceção representa uma situação que normalmente não ocorre e representa algo de estranho ou inesperado no sistema.

Antes de resolvermos o nosso problema, vamos ver como o interpretador age ao se deparar com situações inesperadas, como divisão por zero ou acesso a um índice de uma lista que não existe.

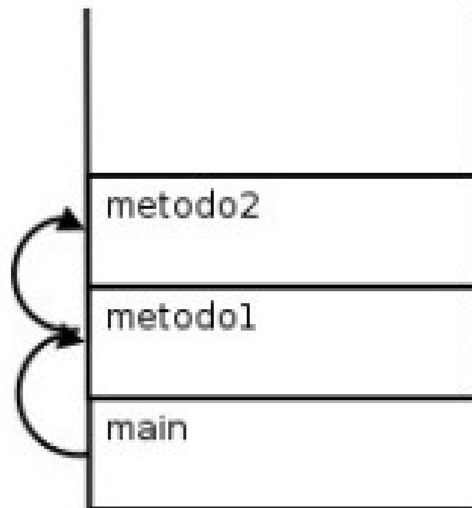
Para aprendermos os conceitos básicos das *exceptions* do Python, crie um arquivo *teste_erro.py* e teste o seguinte código você mesmo:

```
from conta import ContaCorrente

def metodo1():
    print('início do metodo1')
    metodo2()
    print('fim do metodo1')

def metodo2():
    print('início do metodo2')
    cc = ContaCorrente('José', '123')
    for i in range(1,15):
        cc.deposita(i + 1000)
        print(cc.saldo)
        if(i == 5):
            cc = None

    print('fim do metodo2')

if __name__ == '__main__':
    print('início do main')
```

```
metodo1()
print('fim do main')
```

Repare que durante a execução do programa chamamos o `metodo1()` e esse, por sua vez, chama o `metodo2()`. Cada um desses métodos pode ter suas próprias variáveis locais, isto é: o `metodo1()` não enxerga as variáveis declaradas dentro do executável e por aí em diante.

Como o Python (e muitas outras linguagens) faz isso? Toda invocação de método é empilhado em uma estrutura de dados que isola a área e memória de cada um. Quando um método termina (retorna), ele volta para o método que o invocou. Ele descobre isso através da pilha de execução (*stack*): basta remover o marcador que está no topo da pilha:



Porém, o nosso `metodo2()` propositalmente possui um enorme problema: está acessando uma referência para `None` quando o índice for igual a 6!

Rode o código. Qual a saída? O que isso representa? O que ela indica?

```
inicio do metodo1
inicio do metodo2
1001.0
2003.0
3006.0
4010.0
5015.0
Traceback (most recent call last):
  File "/home/thaisandre/PycharmProjects/teste-bot/erro.py", line 23, in <module>
    metodo1()
  File "/home/thaisandre/PycharmProjects/teste-bot/erro.py", line 5, in metodo1
    metodo2()
  File "/home/thaisandre/PycharmProjects/teste-bot/erro.py", line 13, in metodo2
    cc.deposita(i + 1000)
AttributeError: 'NoneType' object has no attribute 'deposita'

Process finished with exit code 1
```

Essa saída é o rastro de pilha, o *Traceback*. É uma saída importantíssima para o programador - tanto que, em qualquer fórum ou lista de discussão, é comum os programadores enviarem, juntamente com a descrição do problema, essa *Traceback*. Mas por que isso aconteceu?

O sistema de exceções do Python funciona da seguinte maneira: quando uma exceção é lançada

(*raise*), o interpretador entra em estado de alerta e vai ver se o método atual toma alguma precaução ao tentar executar esse trecho de código. Como podemos ver, o `metodo2()` não toma nenhuma medida diferente do que vimos até agora.

Como o `metodo2()` não está tratando esse problema, o interpretador para a execução dele anormalmente, sem esperar ele terminar, e volta um *stackframe* para baixo, onde será feita nova verificação: "o `método1()` está se precavendo de um problema chamado `AttributeError`? "Não..." Volta para o executável, onde também não há proteção, então o interpretador morre.

Obviamente, aqui estamos forçando esse caso e não faria sentido tomarmos cuidado com ele. É fácil arrumar um problema desses: basta verificar antes de chamar os métodos se a variável está com referência para `None`.

Porém, apenas para entender o controle de fluxo de uma *Exception*, vamos colocar o código que vai tentar (*try*) executar um bloco perigoso e, caso o problema seja do tipo `AttributeError`, ele será excluído(*except*). Repare que é interessante que cada exceção no Python tenha um tipo... ela pode ter atributos e métodos.

Adicione um `try/except` em volta do `for`, 'pegando' um `AttributeError`. O que o código imprime?

```
from conta import ContaCorrente

def metodo1():
    print('início do metodo1')
    metodo2()
    print('fim do metodo1')

def metodo2():
    print('início do metodo2')
    cc = ContaCorrente('José', '123')
    try:
        for i in range(1,15):
            cc.deposita(i + 1000)
            print(cc.saldo)
            if(i == 5):
            cc = None
    except:
        print('erro')

    print('fim do metodo2')

if __name__ == '__main__':
    print('início do main')
    metodo1()
    print('fim do main')
```

```
Run  erro
inicio do main
inicio do metodo1
inicio do metodo2
1001.0
2003.0
3006.0
4010.0
5015.0
erro
fim do metodo2
fim do metodo1
fim do main

Process finished with exit code 0
```

Em vez de fazer o `try` em torno do `for` inteiro, tente apenas com o bloco dentro do `for` :

```
def metodo2():
    print('início do metodo2')
    cc = ContaCorrente('José', '123')

    for i in range(1,15):
        try:
            cc.deposita(i + 1000)
            print(cc.saldo)
            if(i == 5):
                cc = None
        except:
            print('erro')

    print('fim do metodo2')
```

Qual a diferença?

```
Run  erro
inicio do metodo1
inicio do metodo2
1001.0
2003.0
3006.0
4010.0
5015.0
erro
erro
erro
erro
erro
erro
erro
erro
erro
fim do metodo2
fim do metodo1
fim do main

Process finished with exit code 0
```

Retire o `try/except` e coloque ele em volta da chamada do `metodo2()` :

```
def metodo1():
    print('início do metodo1')
    try:
        metodo2()
    except AttributeError:
        print('erro')
    print('fim do metodo1')
```

```
Run  erro
inicio do main
inicio do metodo1
inicio do metodo2
1001.0
2003.0
3006.0
4010.0
5015.0
erro
fim do metodo1
fim do main

Process finished with exit code 0
```

Faça o mesmo, retirando o `try/except` novamente e colocando em volta da chamada do `metodo1()`. Rode os códigos, o que acontece?

```
if __name__ == '__main__':
    print('início do main')
    try:
        metodo1()
    except AttributeError:
        print('erro')
    print('fim do main')
```

```
Run  erro
inicio do main
inicio do metodo1
inicio do metodo2
1001.0
2003.0
3006.0
4010.0
5015.0
erro
fim do main

Process finished with exit code 0
```

Repare que, a partir do momento que uma *exception* foi *catched* (pega, tratada, *handled*), a exceção volta ao normal a partir daquele ponto.

12.1 EXCEÇÕES E TIPOS DE ERROS

Runtime

Este tipo de erro ocorre quando algo de errado acontece durante a execução do programa. A maior parte das mensagens deste tipo de erro inclui informações do que o programa estava fazendo e o local que o erro aconteceu.

O interpretador mostra a famosa *Traceback* - ele mostra a sequência de chamadas de função que fez com que você chegasse onde está, incluindo o número da linha de seu arquivo onde cada chamada ocorreu.

Os erros mais comuns de tempo de execução são:

- **NameError**

Quando tentamos acessar uma variável que não existe.

```
print(x)

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

No exemplo acima tentamos imprimir `x` sem defini-lo antes. Este erro também é muito comum de ocorrer quando tentamos acessar um variável local em um contexto local.

- **TypeError**

Quando tentamos usar um valor de forma inadequada, como por exemplo tentar indexar um sequência com algo diferente de um número inteiro ou de um fatiamento:

```
lista = [1, 2, 3]
print(lista['a'])

Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: list indices must be integers or slices, not str
```

- **KeyError**

Quando tentamos acessar um elemento de um dicionário usando uma chave que não existe.

```
dicionario = {'nome': 'João', 'idade': 25}
print(dicionario['cidade'])

Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyError: 'cidade'
```

- **AttributeError**

Quando tentamos acessar um atributo ou método que não existe em um objeto.

```
lista = [1, 2, 3]
print(lista.nome)

Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
AttributeError: 'list' object has no attribute 'nome'
```

- **IndexError**

Quando tentamos acessar um elemento de uma sequência com um índice maior que seu comprimento menos um.

```
tupla = (1, 2, 3)
print(tupla[3])

Traceback (most recent call last):
```

```
  File "<stdin>", line 2, in <module>
IndexError: tuple index out of range
```

Agora é a melhor hora de aprender algo novo

alura

Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

Conheça a Alura Cursos Online.

12.2 TRATANDO EXCEÇÕES

Há muitos outros erros de tempo de execução. Que tal dividir um número por zero? Será que o interpretador consegue fazer aquilo que nós definimos que não existe?

```
n = 2
n = n / 0

Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: division by zero
```

Repare que um `ZeroDivisionError` poderia ser facilmente evitado com um `if` que checaria se o denominador é diferente de zero mas a forma correta de se tratar um erro no Python é através do comando **try/except**:

```
try:
    n = n/0
except ZeroDivisionError:
    print('divisão por zero')
```

Que gera a saída:

```
divisão por zero
```

O conjunto de instruções dentro do bloco `try` é executado (o interpretador tentará executar), se nenhuma exceção ocorrer, o comando `except` é ignorado e a execução é finalizada. Mas se ocorrer alguma exceção durante a execução do bloco `try`, as instruções remanescentes são ignoradas e se a exceção lançada prever um `except`, então as instruções dentro do bloco `except` são executadas.

O comando `try` pode ter mais de um comando `except` para especificar múltiplos tratadores para diferentes exceções. No máximo um único tratador será ativado. Tratadores só são sensíveis às exceções

levantadas no interior da cláusula `try` , e não as que tenham ocorrido no interior de outro tratador numa mesma instrução `try` . Um tratador pode ser sensível a múltiplas exceções, desde que as especifique em uma tupla:

```
except(RuntimeError, TypeError, NameError):
    pass
```

A última cláusula `except` pode omitir o nome da exceção, funcionando como um curinga. Não é aconselhável abusar deste recurso já que isso pode esconder erros do programador e do usuário.

O bloco `try/except` possui um comando opcional `else` que, quando usado, deve ser colocado depois de todos os comandos `except` . É útil para código que precisa ser executado se nenhuma exceção foi lançada, por exemplo:

```
try:
    arquivo = open('palavras.txt', 'r')
except IOError:
    print('não foi possível abrir o arquivo')
else:
    print('o arquivo tem {} palavras'.format(len(arquivo.readlines())))
    arquivo.close()
```

12.3 LEVANTANDO EXCEÇÕES

O comando `raise` nos permite forçar a ocorrência de um determinado tipo de exceção. Por exemplo:

```
raise NameError('oi')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: oi
```

O argumento de `raise` indica a exceção a ser lançada. Esse argumento deve ser uma instância de `Exception` ou uma classe de alguma exceção - uma classe que deriva de `Exception` .

Caso você precise determinar se uma exceção foi lançada ou não, mas não quer manipular o erro, uma forma é lançá-la novamente através da instrução `raise` :

```
try:
    raise NameError('oi')
except NameError:
    print('lançou uma exceção')
    raise
```

Saída:

```
lançou uma exceção
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: oi
```

12.4 DEFINIR UMA EXCEÇÃO

Programas podem definir novos tipos de exceções, através da criação de uma nova classe. Exceções devem ser derivadas da classe `Exception` , direta ou indiretamente. Por exemplo:

```
class MeuErro(Exception):
    def __init__(self, valor):
        self.valor = valor
    def __str__(self):
        return repr(self.valor)

if __name__ == '__main__':
    try:
        raise MeuErro(2*2)
    except MeuErro as e:
        print('Minha exceção ocorreu, valor: {}'.format(e.valor))

    raise MeuErro('oops!')
```

Que quando executado gera a saída:

```
Minha exceção ocorreu, valor: 4
Traceback (most recent call last):
  File "<stdin>", line 13, in <module>
    raise MeuErro('oops!')
    __main__.MeuErro: 'oops!'
```

Neste exemplo, o método `__init__` da classe `Exception` foi reescrito. O novo comportamento simplesmente cria o atributo valor. Classes de exceções podem ser definidas para fazer qualquer coisa que qualquer outra classe faz, mas em geral são bem simples, frequentemente oferecendo apenas alguns atributos que fornecem informações sobre o erro que ocorreu.

Ao criar um módulo que pode gerar diversos erros, uma prática comum é criar uma classe base para as exceções definidas por aquele módulo, e as classes específicas para cada condição de erro como subclasses dela:

```
class MeuError(Exception):
    """Classe base para outras exceções"""
    pass

class ValorMuitoPequenoError(Error):
    """É lançada quando o valor passado é muito pequeno"""
    pass

class ValorMuitoGrandeError(Error):
    """É lançada quando o valor passado é muito grande"""
    pass
```

Essa é a maneira padrão de definir exceções no Python mas o programador não precisa ficar preso a ela. É comum que novas exceções sejam definidas com nomes terminando em “Error”, semelhante a muitas exceções embutidas.

Editora Casa do Código com livros de uma forma diferente

Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.
Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

[Casa do Código, ebook com preço de ebook.]

12.5 PARA SABER MAIS: FINALLY

O comando *try* pode ter outro comando opcional chamado *finally*. Sua finalidade é permitir a implementação de ações de limpeza, que sempre devem ser executadas independentemente da ocorrência de exceções. Como no exemplo:

```
def divisao(x, y):
    try:
        resultado = x / y
    except ZeroDivisionError:
        print("Divisão por zero")
    else:
        print("o resultado é {}".format(resultado))
    finally:
        print("executando o finally")


if __name__ == '__main__':
    divide(2, 1)
    divide(2, 0)
    divide('2', '1')
```

Executando:

```
resultado é 2
executando o finally
divisão por zero
executando o finally
executando o finally
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

Repare que o bloco *finally* é executado em todos os casos. A exceção `TypeError` levantada pela divisão de duas *strings* e não é tratada no *except* e portanto é relançada depois que o *finally* é executado.

Em aplicações reais, o *finally* é útil para liberar recursos externos (como arquivos ou conexões de rede), independentemente do uso do recurso ter sido bem sucedido ou não.

12.6 ÁRVORE DE EXCEÇÕES

No Python todas as exceções são instâncias de uma classe derivada de `BaseException` . Ela não serve para ser diretamente herdada por exceções criadas por programadores, para isso utilizamos `Exception` que também é filha de `BaseException` .

Abaixo está a hierarquia de classes de exceções do Python. Para mais informações sobre cada uma delas consulte a documentação: https://docs.python.org/3/library/exceptions.html

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      |    +-- ModuleNotFoundError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- OSError
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError
      |    |    +-- ConnectionAbortedError
      |    |    +-- ConnectionRefusedError
      |    |    +-- ConnectionResetError
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- InterruptedError
      |    +-- IsADirectoryError
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- ProcessLookupError
      |    +-- TimeoutError
      +-- ReferenceError
      +-- RuntimeError
      |    +-- NotImplementedError
      |    +-- RecursionError
      +-- SyntaxError
      |    +-- IndentationError
```

```
 |         +-- TabError
 +-- SystemError
 +-- TypeError
 +-- ValueError
 |    +-- UnicodeError
 |         +-- UnicodeDecodeError
 |         +-- UnicodeEncodeError
 |         +-- UnicodeTranslateError
 +-- Warning
      +-- DeprecationWarning
      +-- PendingDeprecationWarning
      +-- RuntimeWarning
      +-- SyntaxWarning
      +-- UserWarning
      +-- FutureWarning
      +-- ImportWarning
      +-- UnicodeWarning
      +-- BytesWarning
      +-- ResourceWarning
```

12.7 EXERCÍCIOS: EXCEÇÕES

1. Na classe `Conta`, modifique o método `deposita()`. Ele deve lançar uma exceção chamada `ValueError`, que já faz parte da biblioteca padrão do Python, sempre que o valor passado como argumento for inválido (por exemplo, quando for negativo):

```
def deposita(self, valor):
    if(valor < 0):
        raise ValueError
    else:
        self._saldo += valor
```

2. Da maneira com está, apenas saberemos que ocorreu um `ValueError` mas não saberemos o motivo. Vamos acrescentar uma mensagem para deixar o erro mais claro:

```
def deposita(self, valor):
    if(valor < 0):
        raise ValueError('Você tentou depositar um valor negativo')
    else:
        self._saldo += valor
```

3. Faça o mesmo para o método `saca()` da classe `ContaCorrente`, afinal o cliente também não pode sacar um valor negativo.

4. Vamos validar também que o cliente não pode sacar um valor maior do que o saldo disponível em conta. Crie sua própria exceção chamada `SaldoInsuficienteError`. Para isso, você precisa criar uma classe com esse nome que seja filha de RuntimeError.

```
class SaldoInsuficienteError(RuntimeError):
    pass
```

No método saca da classe `ContaCorrente` vamos utilizar esta nova exceção:

```
class ContaCorrente(Conta):
    # código omitido
```

```
    def saca(self, valor):
        if(valor < 0):
            raise ValueError('Você tentou sacar um valor negativo')
        if(self._saldo < valor):
            raise SaldoInsuficienteError()
        self._saldo -= (valor + 0.10)
```


Já conhece os cursos online Alura?

A **Alura** oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex-aluno da Caelum tem 15% de desconto neste link!

Conheça os cursos online Alura.

12.8 OUTROS ERROS

- **Erros de sintaxe** Um dos erros mais comuns é o **SyntaxError**. Geralmente suas mensagens não dizem muito, a mais comum é a `SyntaxError: invalid syntax` . Por outro lado, a mensagem diz o local onde o problema ocorreu. - onde o Python encontrou o problema. São descobertos quando o interpretador está traduzindo o código fonte para o bytecode. Indicam que há algo de errado com a estrutura do programa. Por exemplo: esquecer de fechar aspas, simples ou duplas, na hora de imprimir um mensagem; esquecer de colocar dois pontos (":") ao final de uma instrução `if` , `while` ou `for` , etc...

- **Erro semântico** Este erro é quando o programa não se comporta como esperado. Aqui não é lançada uma exceção, o programa apenas não faz a coisa certa. São mais difíceis de encontrar porque o interpretador não fornece nenhuma informação já que não sabe o que o programa deveria fazer. São erros na regra de negócio. Utilizar a função `print()` em alguns lugares do código onde você suspeita que está gerando o erro pode ajudar.

12.9 PARA SABER MAIS - DEPURADOR DO PYTHON

O depurador do Python, o pdb, é um módulo embutido que funciona como um console interativo onde é possível realizar debug de códigos python. Você pode ler mais a respeito na documentação: https://docs.python.org/3/library/pdb.html

CAPÍTULO 13

COLLECTIONS

Objetivos:

- conhecer o módulo collections
- conhecer o módulo collections.abc

No capítulo 4 vimos uma introdução das principais estruturas de dados do Python como listas, tuplas, conjuntos e dicionários. Também aprendemos em orientação a objetos que tudo em Python é um objeto, inclusive essas estruturas.

O Python possui uma biblioteca chamada `collections` que reúne outros tipos de dados alternativos ao já apresentados no capítulo 4. Esses tipos trazem novas funcionalidades.

O módulo collections também provê um módulo de classes abstratas, o módulo `abc.collections` , que podem ser usadas para testar se determinada classe provê uma interface particular e aprenderemos um pouco sobre elas e seu uso.

13.1 USERLIST, USERDICT E USERSTRING

As estruturas de dados padrão do Python são de grande valia e muito utilizadas na linguagem, mas existem momentos que precisamos de funcionalidades extras que são comuns de projeto para projeto. Nesse sentido surge o módulo `collections` , pra acrescentar essas funcionalidades.

Por exemplo, no Raspberry Pi ou Arduino, uma placa programada com pinos GPIO é representada por um objeto *board* com um atributo *pins*. Esse atributo contém um mapeamento das localizações físicas dos pinos para objetos que representam os pinos. A localização física pode ser um número ou uma *string* como "A0" ou "B1". Por consistência, é desejável que todas as chaves sejam *strings* assim como é conveniente que funcione para *pin[13]* quando o programador desejar fazer piscar o LED do pino 13.

Precisamos usar índices que são *strings*, portanto um dicionário. Além disso, nosso dicionário poderia **apenas** aceitar *strings* como chaves para este objetivo específico. Para não tratar isso durante a execução de nosso programa podemos criar uma classe que tenha o comportamento de um dicionário com essa característica específica.

Para isso, criamos uma classe que herda de uma classe chamada `UserDict` do pacote `collections` :

```
class MeuDicionario(UserDict):
    pass
```

A classe `UserDict` não herda de `dict` mas simula um dicionário. A `UserDict` possui uma instância de `dict` interna chamada `data`, que armazena os itens propriamente ditos.

Criar subclasses de tipos embutidos como `dict` ou `list` diretamente é propenso a erros porque seus métodos geralmente ignoram as versões sobrescritas. Além de que cada implementação pode se comportar de maneira diferente. O fato de herdarmos de `UserDict` e não diretamente de `dict` é para evitar esses problemas.

Criando a classe desta maneira, temos uma classe nossa que funciona como um dicionário. Mas não faz sentido criá-la sem acrescentar funcionalidades, já que o Python já possui essa estrutura pronta que é o `dict`.

Vamos criar nosso dicionário de modo que só aceite chaves como `strings` e vai representar os pinos da placa do Rasbperry Pi, por exemplo:

```
class Pins(UserDict):

    def __contains__(self, key):
        return str(key) in self.keys()

    def __setitem__(self, key, value):
        self.data[str(key)] = value
```

Note que a sobrescrita de `__setitem__` garante que a chave sempre será uma `string`. Podemos testar essa classe:

```
if __name__ == '__main__':
    pins = Pins(one=1)
    print(pins)
    pins[3] = 1
    lista = [1, 2, 3]
    pins(lista) = 2
    print(pins)
```

Perceba que quando imprimimos o dicionário, todas suas chaves são `strings`.

Você pode também fazer o curso PY-14 dessa apostila na Caelum

Querendo aprender ainda mais sobre? Esclarecer dúvidas dos exercícios? Ouvir explicações detalhadas com um instrutor?
A Caelum oferece o **curso PY-14** presencial nas cidades de São Paulo, Rio de Janeiro e Brasília, além de turmas incompany.

Consulte as vantagens do curso *Python e Orientação a Objetos*

13.2 PARA SABER MAIS

Outros tipos que existem no módulo collections são: `defaultdict` , `counter` , `deque` e `namedtuple` .

Ao contrário do `dict` , no `defaultdict` não é necessário verificar se uma chave está presente ou não.

```
cores = [('1', 'azul'), ('2', 'amarelo'), ('3', 'vermelho'), ('1', 'branco'), ('3', 'verde')]

cores_favoritas = defaultdict(list)


for chave, valor in cores:
    cores_favoritas[chave].append(valor)

print(cores_favoritas)
```

O código vai gerar a saída:

```
[('1', ['azul', 'branco']), ('2', ['amarelo']), ('3', ['vermelho', 'verde'])]
```

Sem acusar `KeyError` .

- **Counter**

O `Counter` é um contador e permite contar as ocorrências de um determinado item em uma estrutura de dados:

```
from collections import Counter

cores = ['amarelo', 'azul', 'azul', 'vermelho', 'azul', 'verde', 'vermelho']

contador = Counter(cores)

print(contador)
```

Vai imprimir:

```
Counter({'azul': 3, 'vermelho': 2, 'amarelo': 1, 'verde': 1})
```

Um `Counter` é um `dict` e pode receber um objeto iterável ou um mapa como argumento para realizar a contagem de seus elementos.

- **deque**

O `deque` é uma estrutura de dados que fornece uma `fila` com duas extremidades e é possível adicionar e remover elementos de ambos os lados:

```
from collections import deque

fila = deque()

fila.append('1')
fila.append('2')
fila.append('3')

print(len(fila))        #saída: 3

fila.pop()              #exclui elemento da direita

fila.append('3')       #adiciona elemento na direita

fila.popleft()          #exclui elemento da esquerda

fila.appendleft('1')    #adiciona elemento na esquerda
```

- **namedtuple**

A `namedtuple` , como o nome sugere, são tuplas nomeadas. Não é necessário usar índices inteiros para acessar seus elementos e podemos utilizar *strings* - similar aos dicionários. Mas ao contrários dos dicionários, `namedtuple` é imutável:

```
from collections import namedtuple

Conta = namedtuple('Conta', 'numero titular saldo limite')
conta = Conta('123-4', 'João', 1000.0, 1000.0)

print(conta)            # saída: Conta(numero='123-4', titular='João', saldo=1000.0, limite=1000.0)

print(conta.titular)    #saída: João
```

Note que para acessar o elemento nomeado utilizamos o operador '.' (ponto). Uma `namedtuple` posui dois argumentos obrigatórios que são: o nome da tupla e seus campos (separados por vírgula ou espaço). No exemplo, a tupla se chama `Conta` e possui 4 campos: `numero` , `titular` , `saldo` e `limite` . Como são imutáveis, não podemos modificar os valores de seus campos:

```
conta.titular = "José"
```

Isso vai gerar o seguinte erro:

```
Traceback (most recent call last):
```

```
  File <stdin>, line 5, in <module>
    conta.titular = "José"
AttributeError: can't set attribute
```

A `namedtuple` também é compatível com uma tupla normal. Isso quer dizer que você também pode usar índices inteiros para acessar seus elementos.

```
print(conta[0])     #saída: '123-4'
```

Mais detalhes de cada uma dessas estruturas estão na documentação e pode ser acessada por este link: https://docs.python.org/3/library/collections.html . Outra alternativa é usar a função `help(){ } no objeto para acessar a documentação.

13.3 COLLECTIONS ABC

O módulo `collections.abc` fornece classes abstratas que podem ser usadas para testar se uma classe fornece uma interface específica. Por exemplo, se ela é iterável ou não.

Imagine que o banco nos entregou um arquivo com vários funcionários e pediu que calculássemos a bonificação de cada um deles. Precisamos acrescentar este arquivo em nossa aplicação para iniciar a leitura.

Conteúdo do arquivo funcionarios.txt:

```
João,111111111-11,2500.0
Jose,222222222-22,3500.0
Maria,333333333-33,4000.0
Pedro,444444444-44,2500.0
Mauro,555555555-55,1700.0
Denise,666666666-66,3000.0
Tomas,777777777-77,4200.0
```

Cada linha do arquivo representa um `Funcionario` com seus atributos separados por vírgula. Este arquivo está no padrão *Comma-separated-values*, também conhecido como *csv* e são comumente usados. O Python dá suporte de leitura para este tipo de arquivo. Então vamos acrescentar o módulo `csv` que vai ajudar na tarefa de ler o arquivo:

```
import csv

arquivo = open('funcionario.txt', 'r')
leitor = csv.reader(arquivo)

for linha in leitor:
    print(linha)

arquivo.open()
```

O programa acima abre um arquivo e um leitor do módulo `csv` , o `reader` - recebe o arquivo como parâmetro e devolve um leitor que vai ler linha a linha e guardar seu conteúdo. Podemos iterar sobre este leitor e pedir para imprimir o conteúdo de cada linha - que é exatamente o que é feito no laço *for*. Por último fechamos o arquivo.

A saída será:

```
['João', '111111111-11', '2500.0']
['Jose', '222222222-22', '3500.0']
['Maria', '333333333-33', '4000.0']
['Pedro', '444444444-44', '2500.0']
['Mauro', '555555555-55', '1700.0']
['Denise', '666666666-66', '3000.0']
['Tomas', '777777777-77', '4200.0']
```

Repare que o `reader` guarda cada linha de um arquivo em uma lista e cada valor delimitado por vírgula se torna um elemento desta lista o que facilita o acesso aos dados.

Agora, com estes dados em mãos, podemos construir nossos objetos de tipo `Funcionario` :

```
for linha in reader:
    funcionario = Funcionario(linha[0], linha[1], linha[2])
```

Mas ainda precisamos de uma estrutura para guardá-los. Vamos utilizar uma lista:

```
funcionarios = []

for linha in reader:
    funcionario = Funcionario(linha[0], linha[1], linha[2])
    funcionarios.append(funcionario)
```

E por fim imprimimos os saldos da lista:

```
for f in funcionarios:
    print(f.saldo)
```

Acontece que nada impede, posteriormente, de inserirmos nesta lista qualquer outro objeto que não um funcionário:

```
funcionarios.append('Python')
funcionarios.append(1234)
funcionarios.append(True)
```

A `list` da biblioteca padrão aceita qualquer tipo de objeto como elemento. Não queremos este comportamento já que iremos calcular a bonificação de cada um deles e dependendo do tipo de objetos inserido na lista, gerará erros.

O ideal é que tivéssemos uma estrutura de dados que aceitasse apenas objetos de tipo Funcionario. O módulo `collections.abc` fornece classes abstratas que nos ajudam a construir estruturas específicas, com características da regra de negócio da aplicação.

13.4 CONSTRUINDO UM CONTAINER

O módulo `collections.abc` possui uma classe absrata chamada `Container` > Um `container` é qualquer objeto que contém um número arbitrário de outros objetos. Listas, tuplas, conjuntos e dicionários são tipos de *containers*. A classe `Container` suporta o operador `in` com o método `__contains__` .

Precisamos construir um *container* de objetos de tipo `Funcionario`. Podemos construir uma classe que representará essa estrutura que deve ser subclasse de `Container`:

```
from collections.abc import Container

class Funcionarios(Container):
    pass

if __name__ == '__main__':
    funcionarios = Funcionarios()
```

O código acima acusa um `TypeError`:

```
TypeError: Can't instantiate abstract class Funcionarios with abstract methods __contains__
```

Precisamos implementar o método `__contains__` já que `Funcionarios` deve implementar a classe abstrata `Container`. A ideia é que nosso *container* se comporte como uma lista, então teremos um atributo do tipo lista em nossa classe para guardar os objetos e implementar o método *contains*:

```
from collections.abc import Container

class Funcionarios(Container):

    _dados = []

    def __contains__(self, posicao):
        return self._dados.__contains__(self, posicao)

if __name__ == '__main__':
    funcionarios = Funcionarios()
```


Seus livros de tecnologia parecem do século passado?

Conheça a **Casa do Código**, uma **nova** editora, com autores de destaque no mercado, foco em **ebooks** (PDF, epub, mobi), preços **imbatíveis** e assuntos **atuais**.
Com a curadoria da **Caelum** e excelentes autores, é uma abordagem **diferente** para livros de tecnologia no Brasil.

Casa do Código, Livros de Tecnologia.

13.5 SIZED

O tamanho do nosso *container* também é uma informação importante. Nossa classe `Funcionarios` deve saber retornar esse valor. Utilizamos a classe abstrata `Sized` para garantir essa funcionalidade. A classe `Sized` provê o método `len()` através do método especial `__len__()`:

```python
from collections.abc import Container

class Funcionarios(Container, Sized):

    _dados = []

    def __contains__(self, posicao):
        return self._dados.__contains__(self, posicao)

    def __len__(self):
        return len(self._dados)

if __name__ == '__main__':
    funcionarios = Funcionarios()
```

13.6 ITERABLE

Além de conter objetos e saber retornar a quantidade de seus elementos, queremos que nosso *container* seja iterável, ou seja, que consigamos iterar sobre seus elementos em um laço *for*, por exemplo. O módulo `collection.abc` também provê uma classe abstrata para este comportamento, é a classe `Iterable`. `Iterable` suporta iteração com o método `__iter__`:

```python
from collections.abc import Container

class Funcionarios(Container, Sized, Iterable):

    _dados = []

    def __contains__(self, posicao):
        return self._dados.__contains__(self, posicao)

    def __len__(self):
        return len(self._dados)

    def __iter__(self):
        return self._dados.__iter__(self)

if __name__ == '__main__':
    funcionarios = Funcionarios()
```

Toda coleção deve herdar dessas classes ABCs: `Container`, `Iterable` e `Sized`. Ou implementar seus protocolos: `__contains__`, `__iter__` e `__len__`.

Além dessas classes existem outras que facilitam esse trabalho e implementam outros protocolos. Veja a hierarquia de classe do módulo collections.abc:

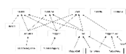


Figura 13.1: legenda da imagem

Além do que já foi implementado, a ideia é que nossa classe `Funcionario` funcione como uma lista contando apenas objetos do tipo `Funcionario`. Como aprendemos no capítulo 4, uma lista é uma sequência. Além de uma sequência, é uma sequência **mutável** - podemos adicionar elementos em uma

lista. Nossa classe `Funcionario` também deve possuir essa funcionalidade.

Segundo o diagrama de classes do módulo `collections.abc`, a classe que representa essa estrutura é a `MutableSequence`. Note que `MutableSequece` herda de `Sequence` que representa uma sequência; que por sua vez herda de `Container`, `Iterable` e `Sized`.

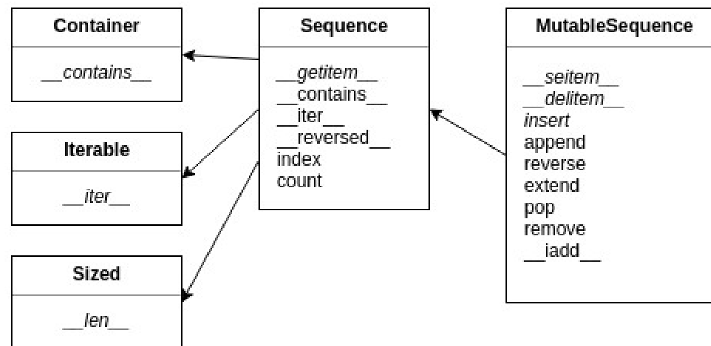


Figura 13.2: legenda da imagem

Portanto, devemos implementar 5 métodos abstratos (em itálico na imagem) segundo a documentação de `MutableSequence`: `__len__`, `__getitem__`, `__setitem__`, `__delitem__` e `insert`. O método `__getitem__` garante que a classe é um `Container` e `Iterable`. Segundo a PEP 234 (https://www.python.org/dev/peps/pep-0234/) um objeto pode ser iterável com um laço *for* se implementa `__iter__` ou `__getitem__`.

Então, basta nossa classe `Funcionario` herdar de `MutableSequence` e implementar seus métodos abstratos:

```
class Funcionarios(MutableSequence):

    _dados = []

    def __len__(self):
        return len(self._dados)

    def __getitem__(self, posicao):
        return self._dados[posicao]

    def __setitem__(self, posicao, valor):
        self._dados[posicao] = valor

    def __delitem__(self, posicao):
        del self._dados[posicao]

    def insert(self, posicao, valor):
        return self._dados.insert(posicao, valor)
```

E podemos voltar ao nosso código para acrescentar os dados de um arquivo em nosso *container* `Funcionarios`:

```
import csv

arquivo = open('funcionario.txt', 'r')
leitor = csv.reader(arquivo)

funcionarios = Funcionarios()

for linha in leitor:
    funcionario = Funcionario(linha[0], linha[1], linha[2])
    funcionarios.append(funcionario)

arquivo.open()
```

O método `insert()` garante o funcionamento do método `append()`. E podemos imprimir os valores dos salários de cada funcionário:

```
for f in funcionarios:
    print(f.salario)
```

Mas até aqui não há nada de diferente de uma lista comum. Ainda não há nada que impeça de inserir qualquer outro objeto em nossa lista. Nossa classe `Funcionarios` se comporta como uma lista comum. A ideia de implementarmos as interfaces de `collections.abc` era exatamente modificar alguns comportamentos.

Queremos que nossa lista de funcionários apenas aceite objetos `Funcionario`. Vamos sobrescrever os métodos `__setitem__()` que atribuiu um valor em determinada posição na lista. Este método pode apenas atribuir a uma determinada posição um objeto `Funcionario`.

Para isso, vamos usar o método `isinstance()` que vai verificar se o objeto a ser atribuído é uma instância de `Funcionario`. Caso contrário, vamos lançar uma exceção `TypeError` com uma mensagem de erro:

```
def __setitem__(self, posicao, valor):
    if (isinstance(valor, Funcionario)):
        self._dados[posicao] = valor
    else:
        raise ValueError('Valor atribuído não é um Funcionario')
```

Agora, ao tentar atribuir uma valor a determinada posição de nossa lista, recebemos um `TypeError`:

```
funcionarios[0] = 'Python'
```

Saída:

```
Traceback (most recent call last):
  File <stdin>, line 18, in __setitem__
    raise ValueError('Valor atribuído não é um Funcionario')
ValueError: valor atribuído não é um Funcionario
```

Faremos o mesmo com o método `insert()`:

```
def insert(self, posicao, valor):
    if(isinstance(valor, Funcionario)):
        return self._dados.insert(posicao, valor)
```

```
    else:
        raise ValueError('Valor inserido não é um Funcionario')
```

E podemos testar nossa classe imprimindo não apenas o salário mas o valor da bonificação de cada `Funcionario` através do método `get_bonificacao()` que definimos nos capítulos passados:

```
if __name__ == '__main__':
    import csv

    arquivo = open('funcionario.txt', 'r')
    leitor = csv.reader(arquivo)

    funcionarios = Funcionarios()

    for linha in leitor:
        funcionario = Funcionario(linha[0], linha[1], linha[2])
        funcionarios.append(funcionario)

    print('salário - bonificação')
    for c in contas:
        print('{} - {}'.format(f.salario, f.get_bonificacao()))

    arquivo.open()
```

As classes ABCs foram criadas para encapsular conceitos genéricos e abstrações como aprendemos no capítulo de classes abstratas. São comumente utilizadas em grandes aplicações e frameworks para garantir a consistência do sistema através dos métodos `isinstance()` e `issubclass()`. No dia a dia é raramente usado e basta o uso correto das estruturas já fornecidas pela biblioteca padrão do Python para a maior parte das tarefas.

Conhecer o módulo collections.abc é

13.7 EXERCÍCIO: CRIANDO NOSSA SEQUÊNCIA

1. Vá na pasta no curso e copie o arquivo `contas.txt` na pasta `src` do projeto `banco` que contém vários dados de contas correntes de clientes do banco.

2. Crie um arquivo chamado *contas.py* na pasta `src` do projeto `banco`. Crie uma classe chamada `Contas` que herde da classe abstrata `MutableSequence`:

```
from collections.abc import Sequence

class Contas(MutableSequence):
    pass
```

3. Vamos criar um atributo da classe do tipo `list` para armazenar nossas contas:

```
from collections.abc import MutableSequence

class Contas(MutableSequence):

    _dados = []
```

4. Tente instanciar um objeto de tipo `Contas`:

```
if __name__=='__main__':
    contas = Contas()
```

Note que não podemos instanciar este objeto. A interface `MutableSequence` nos obriga a implementar alguns métodos:

```
Traceback (most recent call last):
  File <stdin>, line 44, in <module>
    contas = Contas()
TypeError: Can't instantiate abstract class Contas with abstract methods __delitem__, __getitem__
, __len__, __setitem__, insert
```

5. Implemente os métodos exigidos pela interface `MutableSequence` na classe `Contas` :

```
from collections.abc import MutableSequence

class Contas(MutableSequence):

    _dados = []

    def __len__(self):
        return len(self._dados)

    def __getitem__(self, posicao):
        return self._dados[posicao]

    def __setitem__(self, posicao, valor):
        self._dados[posicao] = valor

    def __delitem__(self, posicao):
        del self._dados[posicao]

    def insert(self, posicao, valor):
        return self._dados.insert(posicao, valor)
```

Agora conseguimos instanciar nossa classe sem nenhum erro:

```
if __name__=='__main__':
    contas = Contas()
```

6. Nossa sequência só deve permitir adicionar elementos que sejam do tipo `Conta` . Vamos acrescentar essa validação nos métodos `__setitem__` e `insert` . Caso o valor não seja uma `Conta` , vamos lançar um `ValueError` com as devidas mensagens de erro:

```
def __setitem__(self, posicao, valor):
    if (isinstance(valor, Conta)):
        self._dados[posicao] = valor
    else:
        raise ValueError("valor atribuído não é uma conta")

def insert(self, posicao, valor):
    if(isinstance(valor, Conta)):
        return self._dados.insert(posicao, valor)
    else:
        raise ValueError('valor inserido não é uma conta')
```

7. Vamos iniciar a leitura dos dados do arquivo para armazenar em nosso objeto `contas` :

```
if __name__=='__main__':
```

```python
import csv

contas = Contas()

arquivo = open('contas.txt', 'r')
leitor = csv.reader(arquivo)

arquivo.close()
```

8. Vamos criar uma laço *for* para ler cada linha do arquivo e construir um objeto do tipo `ContaCorrente`.

```python
if __name__=='__main__':
    import csv
    from conta import ContaConrrete

    contas = Contas()

    arquivo = open('contas.txt', 'r')
    leitor = csv.reader(arquivo)

    for linha in leitor:
        conta = ContaCorrente(linha[0], linha[1], linha[2], linha[3])

    arquivo.close()
```

9. Queremos inserir cada conta criada em nossa sequência mutável `contas`. Vamos pedir para que o programa acrescente cada conta criada em contas:

```python
for linha in leitor:
    conta = ContaCorrente(linha[0], linha[1], float(linha[2]))
    contas.append(conta)

arquivo.close()
```

10. Nossa classe `Contas` implementa `MutableSequence`. Isso quer dizer que ela é iterável já que `MutableSequence` implementa o protocolo `__iter__` através do método `__getitem__`. Vamos iterar através de uma laço *for* nosso objeto `contas` e pedir para imprimir o saldo e o valor do imposto de cada uma delas:

```python
if __name__ == '__main__':
    #código omitido

    arquivo.close()

    print('saldo -  imposto')

    for c in contas:
        print('{} - {}'.format(c.saldo, c.get_valor_imposto()))
```

Que vai gerar a saída:

```
saldo  - imposto
1200.0 - 12.0
2200.0 - 22.0
1500.0 - 15.0
5300.0 - 53.0
7800.0 - 78.0
```

```
1700.0 - 17.0
2300.0 - 23.0
8000.0 - 80.0
4600.0 - 46.0
9400.0 - 94.0
```

11. (Opcional) Modifique o código do exercício anterior de modo que imprima o valor do saldo atualizado das contas.

12. (Opcional) Faça o mesmo com as contas poupanças. Crie um arquivo com extensão `.csv` com algumas contas poupanças, faça a leitura, construa os objetos e acrescente em uma estrutura de dados do tipo `MutableSequence` .

13. (Opcional) Refaça o exercício utilizando `MutableMapping` ao invés de `MutableSequence` .

Agora é a melhor hora de aprender algo novo

alura

Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

Conheça a Alura Cursos Online.

CAPÍTULO 14

APÊNDICE - PYTHON2 OU PYTHON3?

Caso você esteja iniciando seus estudos na linguagem Python ou começando um projeto novo, aconselhamos fortemente que você utilize o Python3.

O Python2 vem sendo chamado de Python legado ou Python antigo por boa parte da comunidade que está em constante atividade para fazer a migração da base de código existente (bem grande, por sinal) para Python3.

Aconselhamos a leitura deste artigo para maiores detalhes: https://wiki.python.org/moin/Python2orPython3.

A pergunta correta aqui é: Quando devo usar o Python antigo? E a resposta mais comum que você vai encontrar é: use Python antigo quando você não tiver escolha.

Por exemplo, quando você trabalhar em um projeto antigo e migrar para a nova versão não for uma alternativa no momento. Ou quando você precisa utilizar uma biblioteca que ainda não funciona no Python3 ou não está em processo de mudança. Outro caso é quando seu servidor de hospedagem só permite usar Python2 - aqui o aconselhável é procurar por outro serviço que atenda sua demanda.

No mais, você encontrará muito material sobre o Python2 na internet e aos poucos vai conhecendo melhor as diferenças entre uma versão e outra.

14.1 QUAIS AS DIFERENÇAS?

Neste artigo você vai encontrar a resposta https://docs.python.org/3/whatsnew/3.0.html. Mas neste capítulo mostramos as diferenças mais básicas e importantes para você iniciar seus estudos.

Agora é a melhor hora de aprender algo novo

alura

Se você está gostando dessa apostila, certamente vai aproveitar os **cursos online** que lançamos na plataforma **Alura**. Você estuda a qualquer momento com a **qualidade** Caelum. Programação, Mobile, Design, Infra, Front-End e Business! Ex-aluno da Caelum tem 15% de desconto, siga o link!

Conheça a Alura Cursos Online.

14.2 A FUNÇÃO PRINT()

No Python2 o comando `print` funciona de maneira diferente já que não é uma função. Para imprimir algo fazemos:

```
# no python2
>>> print "Hello World!"
Hello World!
```

No Python3 `print` é uma função e utilizamos os parênteses como delimitadores:

```
# no python3
>>> print("Hello World!")
Hello World!
```

14.3 A FUNÇÃO INPUT()

A função `raw_input` do Python2 foi renomeada para `input()` no Python3:

```
# no python2
>>> nome = raw_input("Digite seu nome: ")
```

No Python3:

```
# no python3
>>> nome = input("Digite seu nome: ")
```

14.4 DIVISÃO DECIMAL

No Python2 a divisão entre números decimais é diferente entre um número decimal e um inteiro:

```
# no python2
>>> 5 / 2
2
>>> 5 / 2.0
2.5
```

No Python3 a divisão tem o mesmo comportamento da matemática. E se quisermos o resultado inteiro da divisão utilizamos `//` :

```
# no python3
>>> 5 / 2
2.5
>>> 5 // 2
2
```


Editora Casa do Código com livros de uma forma diferente

Editoras tradicionais pouco ligam para ebooks e novas tecnologias. Não dominam tecnicamente o assunto para revisar os livros a fundo. Não têm anos de experiência em didáticas com cursos.
Conheça a **Casa do Código**, uma editora diferente, com curadoria da **Caelum** e obsessão por livros de qualidade a preços justos.

Casa do Código, ebook com preço de ebook.

14.5 HERANÇA

No Python2 suas classes devem herdar de `object` :

```
# no python2
>>> class MinhaClasse(object):
    def metodo(self, attr1, attr2):
        return attr1 + attr2
```

No Python3 essa herança é implícita, não precisando herdar explicitamente de `object` :

```
# no python3
>>> class MinhaClasse():
    def metodo(self, attr1, attr2):
        return attr1 + attr2
```

CAPÍTULO 15

APÊNDICE - INSTALAÇÃO

O Python já vem instalado nos sistemas Linux e Mac OS mas será necessário fazer o download da última versão (Python 3.6) para acompanhar a apostila. O Python não vem instalado por padrão no Windows e o download deverá ser feito no site https://www.python.org/ além de algumas configurações extras.

15.1 INSTALANDO O PYTHON NO WINDOWS

O primeiro passo é acessar o site do Python: https://www.python.org/. Na sessão de `Downloads` já será disponibilizado o instalador específico do Windows automaticamente, portanto é só baixar o Python3, na sua versão mais atual.

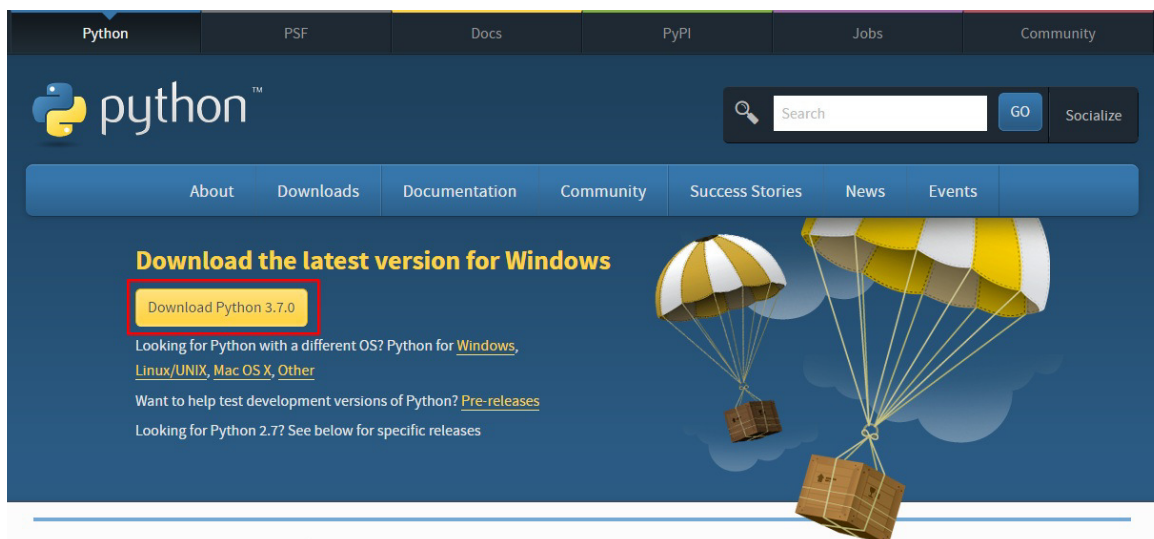


Figura 15.1: Tela de download do Python para o Windows

Após o download ser finalizado, abra-o e na primeira tela marque a opção `Add Python 3.X to PATH`. Essa opção é importante para conseguirmos executar o Python dentro do Prompt de Comando do Windows. Caso você não tenha marcado esta opção, terá que configurar a variável de ambiente no Windows de forma manual.

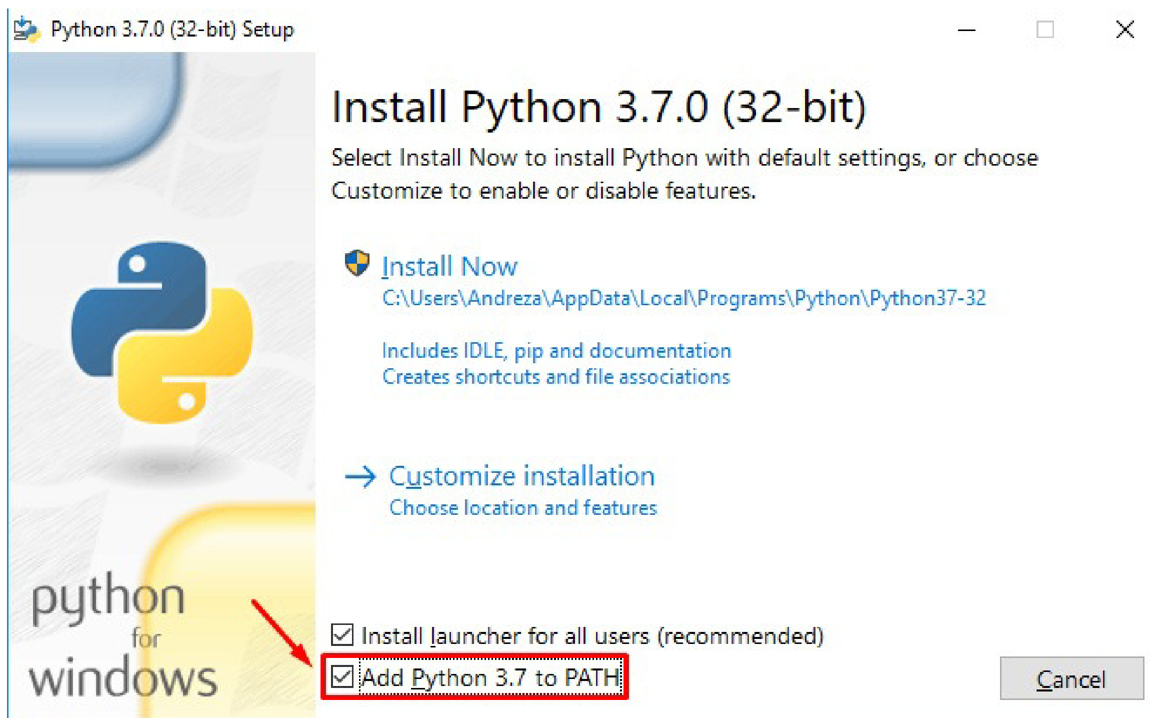


Figura 15.2: Checkbox selecionado

Selecione a instalação customizada somente para ver a instalação com mais detalhes.

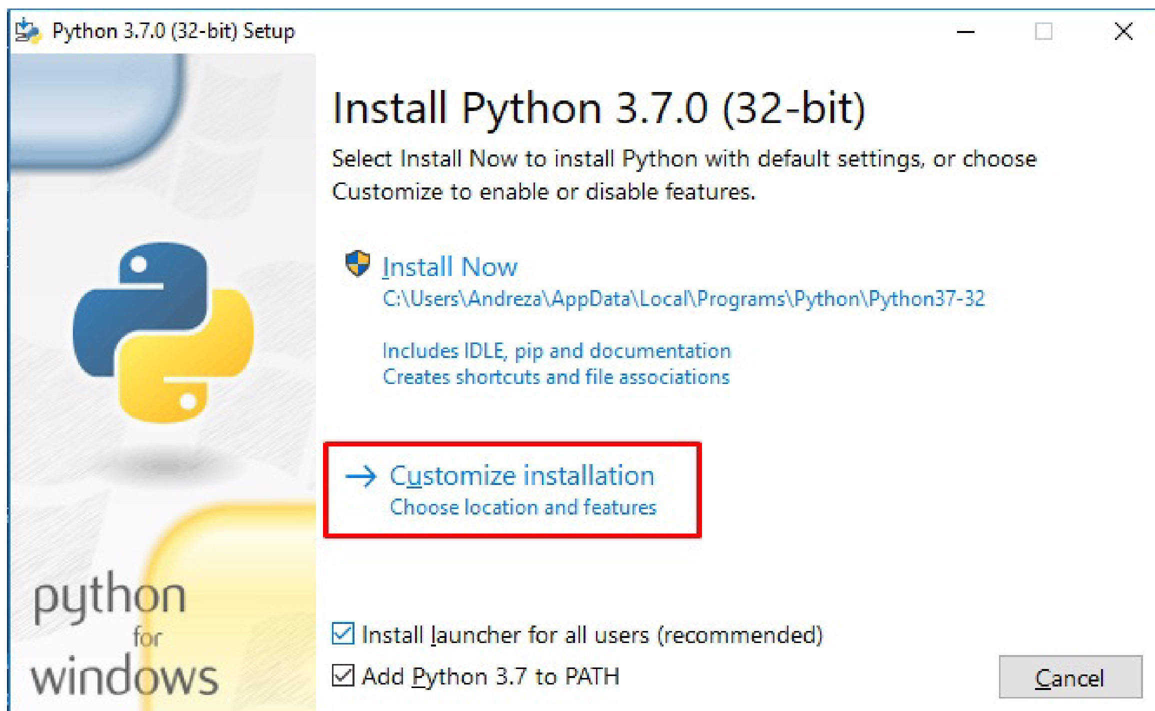


Figura 15.3: Instalação customizada

Na tela seguinte são as features opcionais, se certifique que o gerenciador de pacotes `pip` esteja selecionado, ele que permite instalar pacotes e bibliotecas no Python. Clique em `Next` para dar

seguimento na instalação.

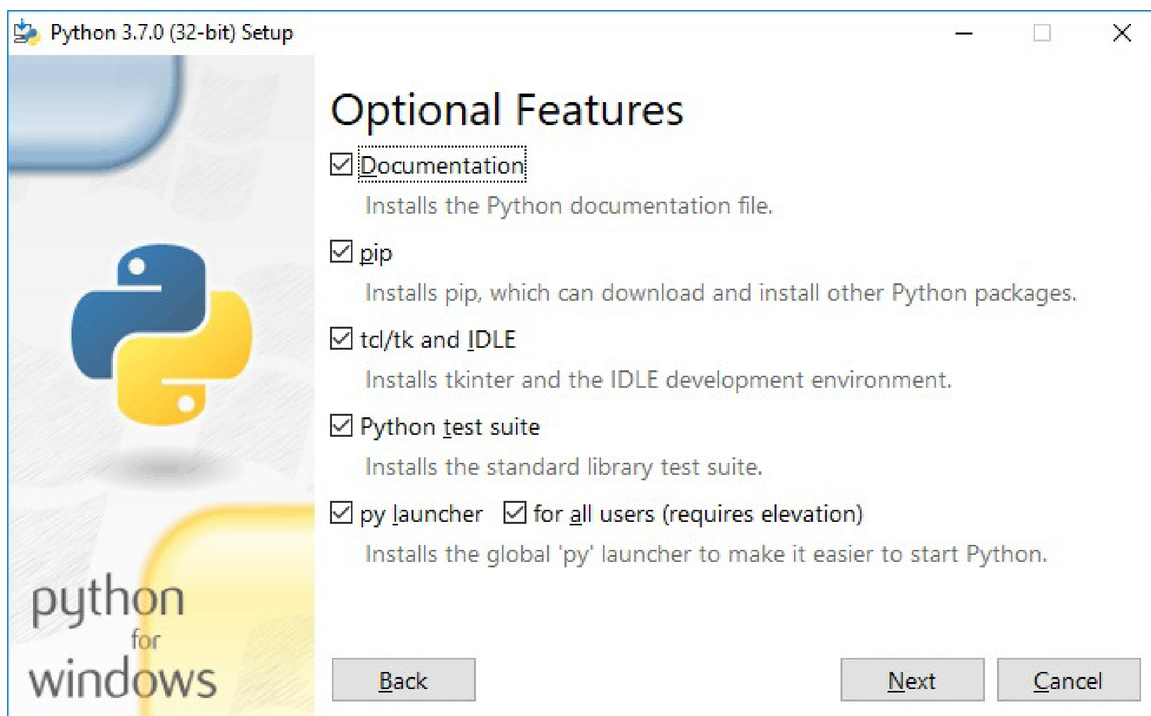


Figura 15.4: Optional Features

Já na terceira tela, deixe tudo como está, mas se atente ao diretório de instalação do Python, para caso queira procurar o executável ou algo que envolva o seu diretório.

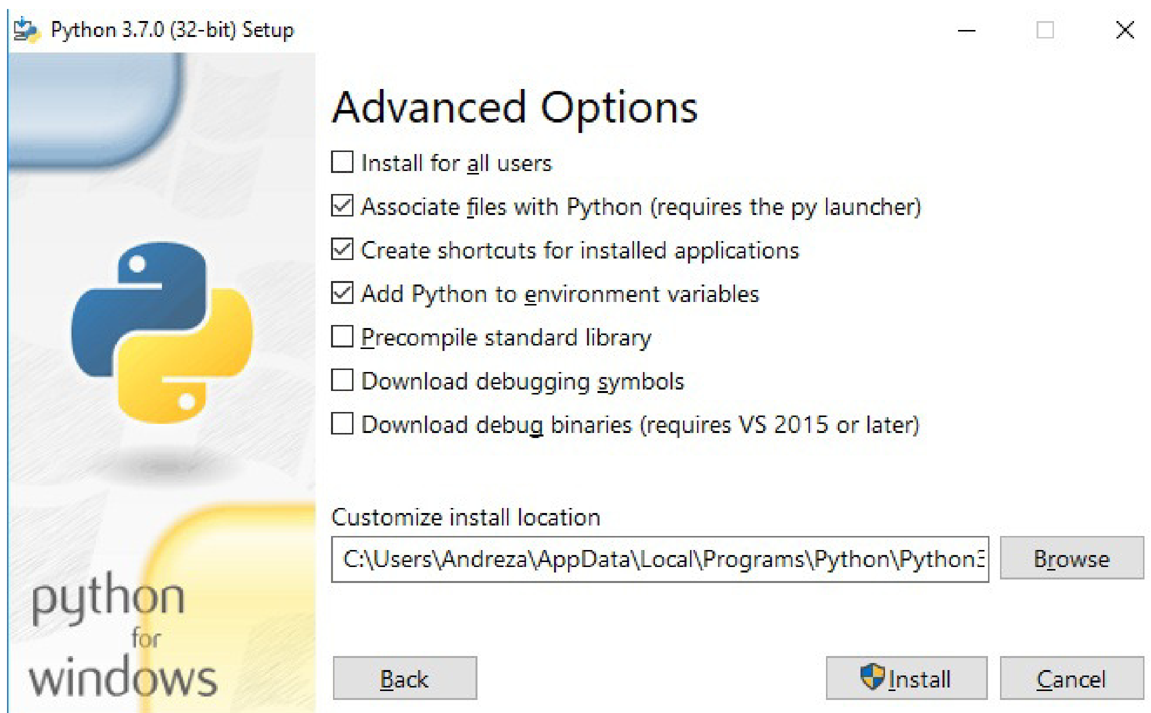


Figura 15.5: Advanced Options

Por fim, basta clicar em `Install` e aguardar o término da instalação.

Figura 15.6: Instalação Concluída com Sucesso

Terminada a instalação, teste se o Python foi instalado corretamente. Abra o Prompt de Comando e execute:

```
python -V
```

OBS: Para que funcione corretamente é necessário que seja no Prompt de Comando e não em algum programa Git Bash instalado em sua máquina. E o comando `python -V` é importante que esteja com o `V` com letra maiúscula.

Esse comando imprime a versão do Python instalada no Windows. Se a versão for impressa, significa que o Python foi instalado corretamente. Agora, rode o comando `python` :

```
python
```

Assim você terá acesso ao console do próprio Python, conseguindo assim utilizá-lo.

Já conhece os cursos online Alura?

A **Alura** oferece centenas de **cursos online** em sua plataforma exclusiva de ensino que favorece o aprendizado com a **qualidade** reconhecida da Caelum. Você pode escolher um curso nas áreas de Programação, Front-end, Mobile, Design & UX, Infra e Business, com um plano que dá acesso a todos os cursos. Ex-aluno da Caelum tem 15% de desconto neste link!

Conheça os cursos online Alura.

15.2 INSTALANDO O PYTHON NO LINUX

Os sistemas operacionais baseados no Debian já possuem o Python3 pré-instalado. Verifique se o seu sistema já possui o Python3 instalado executando o seguinte comando no terminal:

```
python3 -V
```

OBS: O comando `python3 -V` é importante que esteja com o `V` com letra maiúscula.

Este comando retorna a versão do Python3 instalada. Se você ainda não tiver ele instalado, digite os seguintes comandos no terminal:

```
sudo apt-get update
sudo apt3-get install python
```

Para que você consiga instalar os pacotes do Python é necessário ter o gerenciador de pacotes `pip` instalado no sistema. Para instalar esse gerenciador, digite no terminal:

```
sudo apt-get install python-pip
```

15.3 INSTALANDO O PYTHON NO MACOS

A maneira mais fácil de instalar o Python3 no MacOS é utilizando o `Homebrew` . Com o `Homebrew` instalado, abra o terminal e digite os seguintes comandos:

```
brew update
brew install python3
```

Para que você consiga instalar os pacotes do Python é necessário ter o gerenciador de pacotes `pip` instalado no sistema. Para instalar esse gerenciador, digite no terminal:

```
sudo apt-get install python-pip
```

15.4 OUTRAS FORMAS DE UTILIZAR O PYTHON

Podemos rodar o Python diretamente do seu próprio Prompt.

Podemos procurar pelo Python na caixa de pesquisa do Windows e abri-lo, assim o seu console próprio será aberto. Uma outra forma é abrir a IDLE do Python, que se parece muito com o console mas vem com um menu que possui algumas opções extras.

Você pode também fazer o curso PY-14 dessa apostila na Caelum

Querendo aprender ainda mais sobre? Esclarecer dúvidas dos exercícios? Ouvir explicações detalhadas com um instrutor?
A Caelum oferece o **curso PY-14** presencial nas cidades de São Paulo, Rio de Janeiro e Brasília, além de turmas incompany.

Consulte as vantagens do curso *Python e Orientação a Objetos*