

4Linux

também é consultoria, suporte e desenvolvimento.

- Você já deve nos conhecer pelos melhores cursos de linux do Brasil mas a 4Linux também é **consultoria, suporte e desenvolvimento** e executou alguns dos mais famosos projetos do Brasil utilizando tecnologias “open software”.
- Você sabia que quando um cidadão faz uma aposta nas loterias, saca dinheiro em um ATM (caixa eletrônico), recebe um SMS com o saldo de seu FGTS ou simula o valor de um financiamento imobiliário no “feirão” da casa própria, ele está usando uma infraestrutura baseada em softwares livres com consultoria e suporte da 4Linux?

→ Serviços

Consultoria:

Definição de Arquitetura, tuning em banco de dados , práticas DEVOPS, metodologias de ensino on-line, soluções open source (e-mail, monitoramento, servidor JEE), alocação de especialistas em momentos de crise e auditoria de segurança.

Suporte Linux e Open Source:

Contratos com regimes de atendimento – preventivo e/ou corretivo - em horário comercial (8x5) ou de permanente sobre-aviso (24x7) para ambientes de missão crítica. Suporte emergencial para ambientes construídos por terceiros.

Desenvolvimento de Software:

Customização visual e funcional de softwares Open Source, consultoria para a construção de ambiente ágeis de Integração Contínua (Java e PHP) e mentoria para uso de bibliotecas, plataformas e ambientes Open Source. Parceiro Oficial Zend.

→ Parceiros Estratégicos



Saiba mais

(11) 2125-4769
www.4linux.com.br/suporte
contato@4linux.com.br

Sumário

Agenda	2
História e Mercado	3
1.1 – História e Mercado	3
1.2 – Sintaxe Básica do Python	10
1.3 – Entrada e Saída de Dados	19
Tipos de Dados	25
2.1 – String Números e Booleanos	25
2.2 – Lista Tupla e Dicionário	33
2.3 – Estrutura de Dados e Conversão	43
Estrutura de Condição e Repetição	50
3.1 – Operadores e Estrutura de Condição	50
3.2 – Laços de Repetição	61
3.3 – Controle de Loop	68
Tratamento de Exceções	77
4.1 Erros e Exceções	77
4.2 – Exception Types	85
4.3 – Raise Exception	95
4.4 – Manipulação de Arquivos	101
Funções	109
5.1 – Escopo e Sintaxe	109
5.2 – Args e Kwargs	118
5.3 Funções Anônimas	125
Orientação a Objetos	132
6.1 – Orientação a Objeto	132
6.2 – Classes Propriedades e Método	141
6.3 – Herança e Polimorfismo	154
Módulos	161
7.1 – Módulos e Instaladores de Pacotes	161
7.2 – Módulos Nativos	171
7.3 – Ambientes Isolados	186

Sumário

Banco de Dados	193
8.1 – SQL	193
8.2 - PostgreSQL	199
8.3 - MySQL	207
8.4 – Estrutura de Dados no MongoDB	215
8.5 – Comandos no MongoDB	222
Manipulando Banco de Dados	230
9.1 – Python com PostgreSQL	230
9.2 – Python com MySQL	239
9.3 – Python com MongoDB	248
9.4 - SQLAlchemy	257
Testes Unitários	269
10.1 – Introdução a Teste Unitário	269
10.2 – Tes Driven Development	278
10.3 – Behavior Driven Development	286
Python para Sysadmin	313
11.1 – Compreendendo o Projeto	294
11.2 – Construindo o Projeto	300
Python para BigData	306
12.1 – Introdução ao SSH e Paramiko	306
12.2 – Jupyter	315



Python Fundamentals

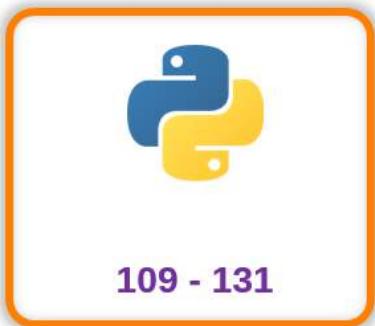
ic pact appears to be the first in which an entire nation has declared war on Taliban insurgents.

Indeed, in the past, Taliban gunmen have threatened tribal leaders who defied them. American military and the Afghan government have largely been unable to protect them.

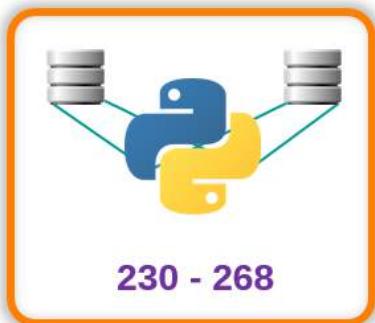
Anotações



História e Mercado



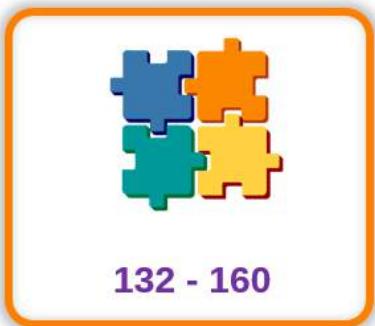
Funções



Manipulando BD



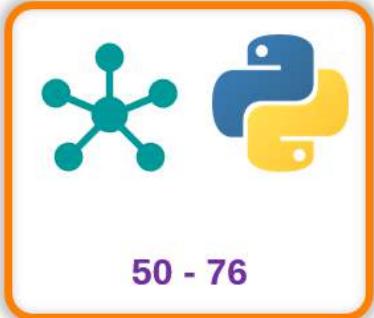
Tipos de dados



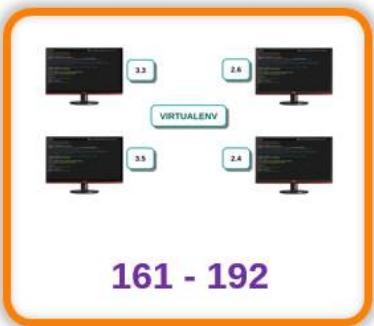
Orientação Objetos



Testes Unitários



Estr. Condição Repetição



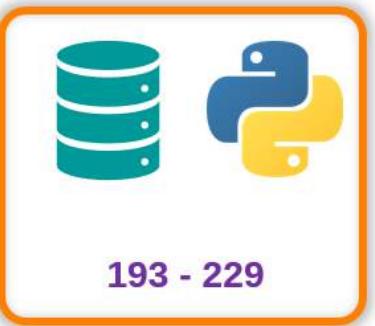
Módulos



Python Sysadmin



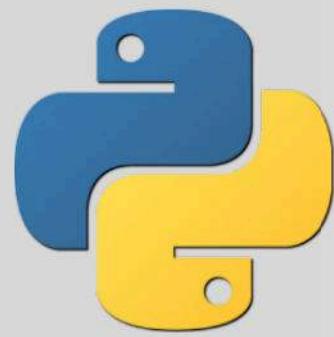
Tratamento Exceções



Banco de Dados



Python BigData



Python Fundamentals

História e Mercado

1.1

Anotações

Objetivos da Aula

- ✓ Conhecer sua história.
- ✓ Introdução ao Python.
- ✓ Quem usa Python.

História e Mercado

Anotações

Nesta aula apresentaremos a história do Python e, empresas que o utilizam.



1989	Criação do Python – Guido Van Rossum
1991	Classes e Herança
1994	Programação Funcional: lambda, map, filter, reduce
2001	Surgimento da Python Software Foundation
2008	Lançamento do Python 3
2012	Raspberry PI – Influência por Python

A linguagem Python surgiu em meados de 1989, criada por Guido Van Rossum, programador holandês. Com base na linguagem ABC cujo propósito era ensinar programação e prototipagem de programas, seguiu a mesma linha com o Python.

O nome Python, foi temporariamente dado pelo autor, por ser fã da série Monty Python, acabou ficando definitivo.

Hoje é uma linguagem onipresente em sistemas Linux, sendo usado pela RedHat, HP, Google, RackSpace, IBM, entre outras. Também possui variações para trabalhar com Java (Jython), DotNet (IronPython) e outras.

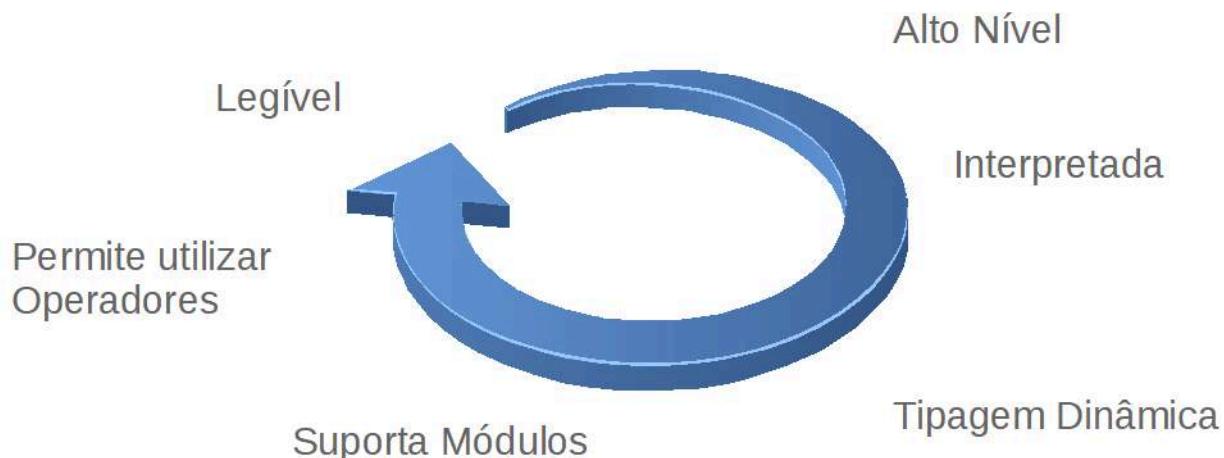
Na wiki da Python Software Foundation há uma lista dos usuários:
<https://wiki.python.org/moin/OrganizationsUsingPython>

Em 2015 foi considerada a 4ª linguagem mais popular pelo iee.org, perdendo somente para Java, C e C++.: <http://spectrum.ieee.org/computing/software/the-2015-top-ten-programming-languages>

Ficou também em 4º lugar no ranking da RedMonk:
<http://redmonk.com/sogrady/2015/07/01/language-rankings-6-15/>

Python é uma linguagem que serve a qualquer propósito e, se encaixa muito bem no contexto DevOps, que utilizaremos no decorrer deste curso.

Linguagem Orientada a Objetos



Conhecendo o Python

Alto Nível: podemos dizer que a linguagem é de alto nível por ser mais próxima à linguagem do ser humano, não diretamente relacionada ao computador, ou seja, não existe a necessidade do programador conhecer instruções de processador e afins.

Interpretada: significa uma linguagem de programação na qual não há necessidade de gerar um arquivo binário, você apenas terá o arquivo e, a partir disto é necessário um interpretador para conseguir executar o seu script ou aplicação.

Tipagem Dinâmica: não é necessário definir qual o tipo de valor em uma variável, para criar, basta atribuir um valor a esta. Ex: `idade = 30`, mesmo eu não tendo definido o tipo de valor, será assumido o tipo "int" (inteiro).

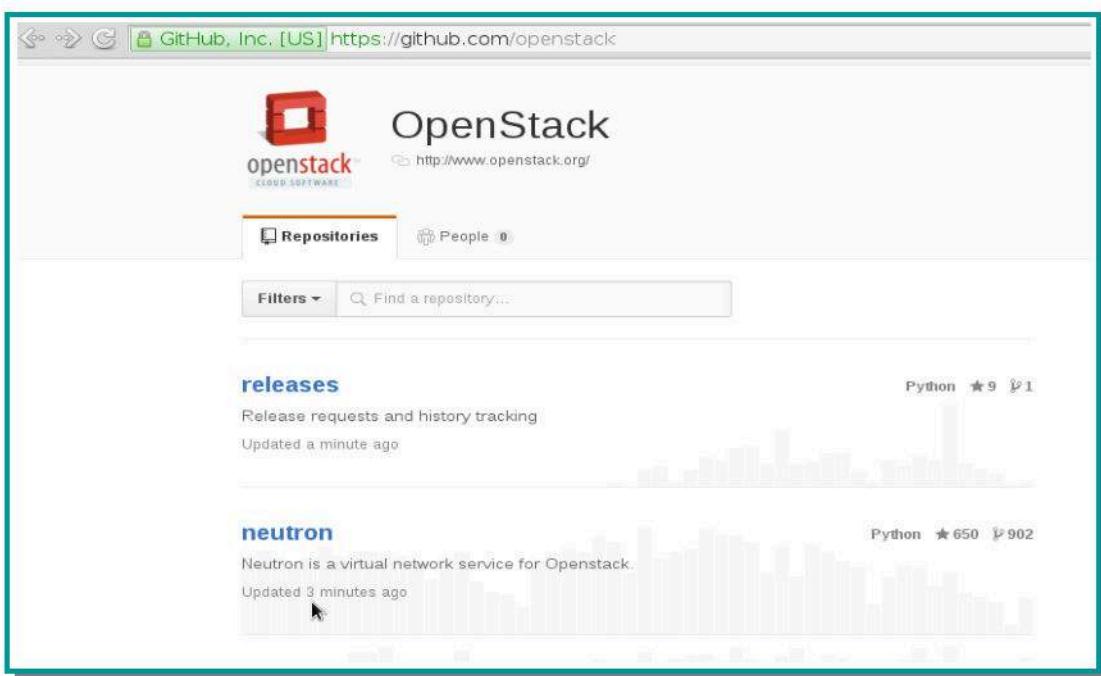
Suporta Módulos: podemos fazer a instalação de módulos externos, por exemplo: Paramiko para SSH ou o Flask para servidor web.

Permite utilizar Operadores: Conseguimos utilizá-los para adição, subtração, multiplicação, entre outros.

Legível: permite criar seu Code Style e sua identação, também possibilita definir blocos de código.

Linguagem Orientada a Objetos: Python tem recursos que dão suporte à Programação Orientada a Objetos (POO).

4LINUX / Introdução ao Python



OpenStack

OpenStack é a ferramenta de Virtualização OpenSource mais popular da atualidade, foi escrita utilizando a linguagem python.

OpenStack teve influência muito forte da RackSpace, empresa concorrente da Amazon WebServices e da Nasa (agência espacial americana), por disponibilizarem o código fonte no GitHub.

OpenStack: <https://github.com/openstack>

Anotações

4LINUX / Introdução ao Python

The screenshot shows the GitHub repository page for the Ansible project. At the top, there's a large 'A' logo and the word 'Ansible'. Below it, there are two main repository cards:

- ansible-modules-extras**: Python ★ 432 ⚡ 896. Description: Ansible extra modules - these modules ship with ansible. Updated 7 minutes ago.
- ansible**: Python ★ 14,620 ⚡ 4,301. Description: Ansible is a radically simple IT automation platform that makes your applications and systems easier to deploy. Avoid writing scripts or custom code to deploy and update your applications— automate in a language that approaches plain English, using SSH, with no agents to install on remote systems. Updated 9 minutes ago.

Ansible

O Ansible é uma ferramenta para automatização de infraestrutura de T.I., criada com o propósito de automatizar a instalação de pacotes no Linux, Deploy de Aplicações, Gerenciamento de Configurações, Automatização de Tarefas e Monitoração.

Também escrito em python seu código fonte está disponível no github no seguinte link:
<https://github.com/ansible/ansible>

Anotações



Globo

A Globo.com, também usa fortemente a linguagem de programação. É possível comprovar isso verificando a quantidade de repositórios na linguagem python que existentes no github:
<https://github.com/globocom>

Anotações



Python Fundamentals

Sintaxe Básica do Python

1.2

Anotações

Objetivos da Aula

- ✓ Conhecer o console interativo.
- ✓ Interpretador do Python.
- ✓ Identação.
- ✓ Variáveis.

Sintaxe básica do Python

Esta aula aborda o console interativo que a linguagem nos proporciona, entender sobre o interpretador do Python, como funciona a identação e como utilizar variáveis.

Anotações

Python possui um modo interativo que permite “conversar” diretamente com o interpretador para definir variáveis, funções e classes.

```
root@developer:/# python3
>>> print "Agora você está no modo
interativo :)"
Agora você está no modo interativo :)
>>> exit()
```

Console Interativo

Ao acessar o Console Interativo, o interpretador imprimirá uma mensagem de boas vindas, o número da versão, uma nota copyright e o prompt, que pode ser identificado por três sinais de maior (“>>>”), caso apareça “...”, Salveo interpretador está aguardando completar o comando.

O console é utilizado para a realização testes, antes de colocarmos o código em nossa aplicação. Desta forma, conseguimos verificar se determinada função ou classe, funciona tendo o retorno esperado.

O Console Interativo, também é interessante para aprender, por exemplo: ao executar o comando “help()” ele trará a documentação de como utilizar um módulo.

Para saber mais sobre o que é programar em Python, digite “import this”. Estes princípios também podem ser encontrados em: <https://www.python.org/dev/peps/pep-0020/>

Anotações

Ao criar scripts em Python, o caminho do interpretador deverá ser especificado para execução na versão correta.

Desta forma, adicionaremos no início do nosso script:

```
#!/usr/bin/python3
```

Para executar o script, utilize:

```
python3 script.py
```

Interpretador do Python

Para executar os scripts em Python, além da forma descrita, podemos utilizar `./script.py`. Neste Caso, é obrigatório ter o interpretador especificado para que não ocorram erros.

Quando executamos com “`python3 script.py`” não existe esta necessidade, pois ao chamar o script já, estamos falando quem irá interpretar, neste caso o `python3`.

Para utilizar um script com Python na versão 2, basta especificar o seguinte interpretador: `#!/usr/bin/python2` e pode ser executado com “`python2 script.sh`”.

Anotações

Python separa os blocos de código por identação:

```
#!/usr/bin/python3
print("Hello World!")
nome = input("Qual é a melhor linguagem de
    programação: ")
if nome == "python":
    print("Você acertou!")
else:
    print("Errou! =( ")
```

Identação

A sintaxe do Python é diferente de outras linguagens, não possui chaves para separar os blocos de código.

Os códigos são separados por identação, conforme o exemplo no slide, isso facilita a organização do código e a sua leitura.

Em Python facilmente sabemos quando a programação está errada. Caso o código esteja muito para a direita, identificamos a ocorrência de muitas funções aninhadas que, provavelmente podem ser separadas facilitando a manutenção e evitando a duplicação de código.

Anotações

Podemos adicionar comentários em nossos códigos. Serão ignorados pelo Python, mas, tornam o código organizado, facilitando sua compreensão.

#Comentário de uma linha



Desta forma podemos inserir comentários em quantas linhas forem necessárias!

Anotações

No Python existem os seguintes tipos de variáveis:

```
CONSTANTE = 3.14
numero = 1
decimal = 0.003
texto = "4Linux Devops"
dicionario = {"nome": "Guido", "idade": 59}
lista = ["item1", "item2", 3, "quatro", 3.14]
tupla = (1, 2, 3, "Python")
```

Variável e Constante

Programas de computador utilizam os recursos de hardware mais básicos para executar algoritmos. Enquanto o processador executa os cálculos, a memória é responsável por armazenar dados e servi-los ao processador. O recurso utilizado nos programas para escrever e ler dados da memória do computador é conhecido como variável. Consiste simplesmente um espaço na memória no qual guardamos uma informação que reservamos a qual atribuiremos um nome. Por exemplo: podemos criar uma variável chamada "idade" para armazenar a idade de uma pessoa.

Chamamos variável, este espaço alocado na memória, porque o valor armazenado pode ser alterado ao longo do tempo, ou seja, o valor ali alocado é "variável" ao longo do tempo. Diferente das constantes, cujo espaço reservado, na memória, para armazenar um valor, não muda com o tempo. Por exemplo: o valor PI (3.14159265359...), nunca vai mudar!

Anotações

1º

A Indentação deve ser de 4 espaços (Sem Tabs).

2º

Límite de 79 caracteres por linhas (84 no máximo).

3º

Linhas muito longas são quebradas por \

4º

Alinhar os parênteses em caso de quebra de linha.

5º

As funções devem estar sempre 2 linhas abaixo a de cima.

6º

Não usar espaço depois de abrir ou fechar um parênteses.

Exemplos

Python não apresenta muitas regras, quanto à sua forma de programar. Esse code style, foi baseado nas práticas dos desenvolvedores do Flask.

Exemplos de boas práticas

Nome de funções:

```
funcao_com_nome_muito_grande(param1,param2) \
    .cascateando_o_retorno()
```

Usando Parênteses:

```
quebrando_por_parenteses(param1,
                           Param2)
```

Em caso de listas:

```
alunos = [
    "se", "forem", "poucos", "items",
    "forem",
    "muitos"
]
```

Em caso de funções:

```
def funcao1(param):
    print "4linux"
```

```
def funcao2(param):
    print "devops"
```

No caso de métodos dentro de uma classe, separa-se somente por uma linha em branco, em caso de funções, 2 linhas em branco.

Em caso de expressões:

```
valor = (num1 / num2) * num3 / num4  
if var == "python":
```

No caso de comparadores, a recomendação é usar a variável, sempre, antes da string ou número a ser comparado.

Caso sejam valores booleanos, faça o if da seguinte maneira:

```
if valor is True:  
if valor is False
```

Em caso de valores nulos:

```
if var is None:
```

Essas comparações também podem ser feitas utilizando o sinal de igual ==.

Exemplo:

```
if var == True
```

Porém, o comparador is, foi criado com o propósito de comparar esses valores, logo, deve ser usado.

Anotações



Python Fundamentals

Entrada e Saída de Dados

1.3

Anotações

Objetivos da Aula

- ✓ Compreender como exibir os dados.
- ✓ Entender como interagir com os usuários.

Entrada e Saída de Dados

Nesta aula compreenderemos como interagir com o usuário usando entrada e saída de dados, deixando nossos programas dinâmicos.

Anotações

Saída de Dados

- ✓ Utilizamos a função “print ()” para retornar os dados do programa para a saída padrão (a tela).
- ✓ Também utiliza-se a função, caso o programa apresente algum comportamento inesperado, assim conseguimos verificar o retorno dos valores esperados.
- ✓ Além desta função, podemos enviar a saída de dados para um arquivo e armazená-la para uso futuro.

Saída de Dados

No Python 3, o **print** deve ser realizado com parênteses, da seguinte forma: `print (variavel)`.

Já no Python 2, não precisamos utilizar o parênteses, basta utilizar: `print variavel`.

Note também que em Python 3, conseguimos imprimir uma quantidade de valores, sendo possível definir o seu separador.

Veja todas as modificações na documentação oficial: <https://docs.python.org/3.0/whatsnew/3.0.html>

Anotações

Utilizando o Print

Para retornar um valor para o usuário, utilizaremos o “print”:

```
# /usr/bin/python3
animal = "Leão"
print(animal)
```

Podemos definir o separador quanto retornamos dois valores:

```
nome = "Guido"
sobrenome = "van Rossum"
print(nome, sobrenome, sep=". ", end="\n\n")
```

Parâmetros da função print.

A Função print possui parâmetros:

*args: permite passar diversos valores para impressão na tela.

sep: é o separador de cada valor, por padrão, recebe um espaço vazio como default sep=' '.

end: utilizado para quebrar linhas ou organizar a exibição dos valores após a execução da função.

Por padrão, recebe um '\n' como default, equivalente a uma quebra de linha end="\n".

Na ausência da especificação destes parâmetros, a função executará com os valores default.

Anotações

Entrada de Dados

- ✓ Até este momento trabalhamos de forma estática, ou seja, sem interagir com o usuário. Python permite que o usuário faça inserção de dados.
- ✓ Os dados inseridos pelo usuário, sempre serão interpretados como “string” e, deverão ser tratados em nosso programa.
- ✓ O que for inserido pelo usuário, é armazenado em uma variável e utilizada no programa.

Entrada de Dados

No Python 3, a entrada de dados é realizada apenas com o comando “**input**”, que tratará todas as inserções da mesma forma (como string).

No Python 2, há dois comandos para a entrada de dados, “**raw_input**” para guardar dados do tipo texto e “**input**” para guardar qualquer outro tipo de dado. Por exemplo:

```
>>> nome = raw_input("Digite o seu nome: ")  
Digite o seu nome: Mariana  
>>> idade = input("Digite a sua idade: ")  
Digite a sua idade: 50  
>>> print type(nome)  
<type 'str'>  
>>> print type(idade)  
<type 'int'>
```

Anotações

Utilizando o Input

Para interagir com usuário, utilizaremos a função “input”:

```
#!/usr/bin/python3
input('Digite o nome de um animal: ')
```

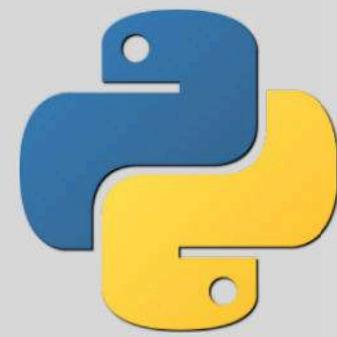
Podemos gravar esta interação em uma variável:

```
#!/usr/bin/python3
animal = input('Digite o nome de um animal: ')
print(animal)
```

Entrada e Saída de Dados

Com a função input recebemos dados do usuário, serão interpretados como string, armazenados em uma variável e, posteriormente exibido o valor da variável com a função print.

Anotações



Python Fundamentals

String, Números e Booleanos

2.1

Anotações

Objetivos da Aula

- ✓ Como o Python utiliza os dados.
- ✓ Tipos de Dados: strings, números, booleanos.

Tipos de Dados Básicos

Nesta aula apresentaremos como o Python utiliza os tipos de dados, strings, números e booleanos.

Anotações

- ✓ O Tipo de dado constitui a maneira como classificamos a informação. Por exemplo: dentro do meu código, números inteiros serão processados diferente de números decimais.
- ✓ Toda a informação é obrigatoriamente de um “tipo”.
- ✓ Em Python, temos os **built-ins**, tipos de dados, embutidos no núcleo da linguagem.

Tipagem de Dados

Muitas linguagens de programação, exigem que as aplicações contenham especificações acerca de qual tipo, constituem determinadas variáveis. Por exemplo: em JAVA para definir um valor inteiro, será necessário utilizar: “`int var = 1;`” outras linguagens que utilizam são: C, C++ e Haskell.

Como vimos na Aula 01, em Python a tipagem de dados é dinâmica, ou seja, não precisamos especificar qual é o tipo de uma informação.

Anotações

Os tipos numéricos no Python são: números inteiros, números de ponto flutuante e números complexos.

Inteiros

```
>>> x = 10  
>>> type(x)  
<class'int'>
```

Flutuantes

```
>>> x = 3.12  
>>> type(x)  
<class'float'>
```

Complexos

```
>>> x= 1 - 2j  
>>> type(x)  
<class'complex'
```

Tipos de Dados - Numéricos

Na linguagem Python, há três tipos de dados numéricos: números inteiros, números de ponto flutuante e números complexos.

Números inteiros, são identificados por ‘**int**’, são descritos como números de precisão fixa.
Números de ponto flutuante, são reconhecidos como ‘**float**’, são números de precisão variável.
Números complexos, diferenciam-se por ‘**complex**’, possuem uma parte real e uma parte imaginária.

Anotações

- ✓ Booleanos em Python são: True (Verdadeiro) e False (Falso).
- ✓ Apesar dessa representação em caracteres, os booleanos False podem representar 0 e True o inteiro 1.
- ✓ Em controle de decisão, Python também considera 0 como False e tudo que for diferente como True.

Anotações

```
In [1]: vdd = True
In [2]: fal = False
In [3]: vdd.numerator
Out[3]: 1
In [4]: fal.numerator
Out[4]: 0
In [5]: not vdd
Out[5]: False
In [6]: not fal
Out[6]: True
```

Tipos de Dados - Booleanos

- In[1]: Definimos uma variável que recebeu um booleano verdadeiro.
- In[2]: Definimos uma variável que recebeu um booleano falso.
- In[3]: Utilizamos o atributo 'numerator' do objeto booleano para ver seu valor numérico.
- In[4]: Utilizamos o atributo 'numerator' do objeto booleano para ver seu valor numérico.
- In[5]: Utilizamos o operador de negação e objetivamos a saída inversa.
- In[6]: Utilizamos o operador de negação e objetivamos a saída inversa.

Anotações

Dado do tipo texto em Python, é identificado por “string”.

```
String  
>>> x = "Python Fundamentals - 4linux"  
>>> print(x)  
Python Fundamentals - 4linux  
>>> type(x)  
<class 'str'>
```

Tipos de Dados – Texto

Os dados de texto em Python, são identificados por ‘str’, podem ser definidos de duas formas:

Citações simples: 'Python Fundamentals – 4linux'

Aspas: “Python Fundamentals – 4linux”

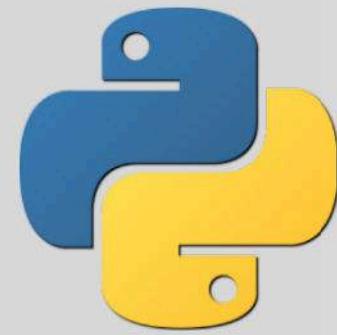
Anotações

```
In [1]: var = ' curso python fundamentos '
In [2]: var.title()
Out[2]: ' Curso Python Fundamentos '
In [3]: var.upper()
Out[3]: ' CURSO PYTHON FUNDAMENTOS '
In [4]: var.lower()
Out[4]: ' curso python fundamentos '
In [5]: var.replace('o', 'ø')
Out[5]: ' curso þþthon fundaméntos '
In [6]: var.strip()
Out[6]: 'curso python fundamentos'
In [7]: var.split()
Out[7]: ['curso', 'python', 'fundamentos']
In [8]: var.islower()
Out[8]: True
In [9]: var.startswith(' c')
Out[9]: True
In [10]: ' . '.join(var)
Out[10]: ' .c.u.r.s.o. .p.y.t.h.o.n. .f.u.n.d.a.m.e.n.t.o.s. '
```

Tipos de Dados – Texto

- In [1] : Definimos a variável var que recebe a string.
- In [2] : O método ‘title’ deixa a string no formato título.
- In [3] : O método ‘upper’ deixa a string no formato uppercase.
- In [4] : O método ‘lower’ deixa a string no formato lowercase.
- In [5] : O método ‘replace’ recebe dois parâmetros, primeiro o ‘string’ que deseja trocar, segundo ‘string’ a qual deseja substituir a anterior.
- In [6] : O método ‘strip’ elimina espaços vazios antes e depois da ‘string’, temos também o ‘rstrip’, elimina espaços vazios na direita e, ‘lstrip’ elimina espaços vazios na esquerda.
- In [7] : o método ‘split’, quebra a ‘string’ em uma lista, podendo ser utilizado passando um parâmetro para definir por qual carácter, separar cada elemento da lista, por default este parâmetro recebe um espaço vazio.
- In [8] : O método ‘isxxx’ verifica o tipo da ‘string’ e retorna um booleano se verdadeiro ou falso.
- In [9] : O método ‘startswith’ verifica os caracteres iniciais da ‘string’ e devolve um booleano.
- In [10] : O método ‘join’ interage uma ‘string’ com a outra , mesclando seus caracteres.

Anotações



Python Fundamentals

Listas, Tupla e Dicionário

2.2

Anotações

Objetivos da Aula

- ✓ Conhecer como o Python utiliza os dados.
- ✓ Conhecer os Tipos de Dados Básicos: listas, tuplas e dicionários.

Tipos de Dados Básicos

Nesta aula compreenderemos como o Python utiliza dados em listas, tuplas e dicionários.

Anotações

Estrutura de dados composta por itens organizados de forma linear, podem ser acessados a partir de um índice que representa sua posição na coleção (iniciando em zero).

```
lista = ['a', 'b', 'c', 'd']
print(lista)
['a', 'b', 'c', 'd']

print(lista[0])
'a'
```

```
print(type(lista))
<class 'list'>

print(type(lista[0]))
<class 'str'>
```

Anotações

Índice e fatiamento

```
In [1]: lista = ['a', 'b', 'c', 'd']

In [2]: lista[2]
Out[2]: 'c'

In [3]: lista[-1]
Out[3]: 'd'

In [4]: lista[0] = 'A'
Out[4]: ['A', 'b', 'c', 'd']

In [5]: lista
Out[5]: ['A', 'b', 'c', 'd']

In [6]: lista[:]
Out[6]: ['A', 'b', 'c', 'd']

In [7]: lista[1:]
Out[7]: ['b', 'c', 'd']

In [8]: lista[1:-2]
Out[8]: ['b']
```

Tipos de dados: Índice e fatiamento

In[1]: Definimos uma lista de strings em uma variável.

In[2]: Acessamos um elemento pelo índice.

In[3]: Podemos acessar os índices de trás pra frente, passando números negativos.

In[4]: Podemos alterar o valor do elemento de uma lista atribuindo com sinal de recebe.

In[5]: Visualizamos a lista com alteração realizada no passo anterior.

In[6]: Podemos clonar uma lista, ou uma parte dela, passando o fatiamento, na sintaxe acima o que vem antes do sinal ‘:’ é início, posteriormente é o fim. Caso não especifique, ocorrerá uma cópia do começo ao fim.

In[7]: Especifica o índice de início do fatiamento, ocultando o fim, observe que foi feio um recorte na lista.

In[8]: Fatia a lista especificando início e fim, o início é inclusivo e o fim é exclusivo.

Anotações

```
In [1]: letras = ['a', 'b', 'c', 'd']
In [2]: letras.append('e')
In [3]: letras
Out[3]: ['a', 'b', 'c', 'd', 'e']
In [4]: letras.insert(0,'A')
In [5]: letras
Out[5]: ['A', 'a', 'b', 'c', 'd', 'e']
In [6]: letras.pop()
Out[6]: 'e'
In [7]: letras
Out[7]: ['A', 'a', 'b', 'c', 'd']
In [8]: letras.pop(2)
Out[8]: 'b'
In [9]: letras
Out[9]: ['A', 'a', 'c', 'd']
```

Tipos de dados: Métodos e lista

In[1]: Definimos uma lista de strings em uma variável.

In[2]: Utilizamos o método 'append', recebe parâmetro do que será inserido incluindo no último índice da lista.

In[3]: Visualizamos a lista com a alteração realizada no passo anterior.

In[4]: Utilizamos o método 'insert' para receber dois parâmetros, primeiro, a posição que deverá ser inserida. Segundo, o que será inserido na posição informada.

In[5]: Visualizamos a lista com a alteração realizada no passo anterior.

In[6]: Utilizamos o método 'pop', remove o último índice da lista.

In[7]: Visualizamos a lista com a alteração realizada no passo anterior.

In[8]: Utilizamos o método 'pop' especificando a posição do item que deve ser removido.

In[9]: Visualizamos a lista com a alteração realizada no passo anterior.

Anotações

```
In [1]: letras = ['z', 'a', 'j', 'f']
In [2]: letras.sort()
In [3]: letras
Out[3]: ['a', 'f', 'j', 'z']
In [4]: letras.reverse()
In [5]: letras
Out[5]: ['z', 'j', 'f', 'a']
In [6]: letras.index('j')
Out[6]: 1
In [7]: letras.count('f')
Out[7]: 1
In [8]: letras.remove('a')
In [9]: letras
Out[9]: ['z', 'j', 'f']
```

Tipos de dados: Métodos e lista

- In[1]: Definimos uma lista de strings em uma variável.
- In[2]: Utilizamos o método 'sort' que ordena a lista.
- In[3]: Visualizamos a lista com a alteração realizada no passo anterior.
- In[4]: Utilizamos o método 'reverse' que inverte a ordem da lista.
- In[5]: Visualizamos a lista com a alteração realizada no passo anterior.
- In[6]: Utilizamos o método 'index' que recebe um parâmetro de busca e retorna a posição da primeira ocorrência do elemento procurado.
- In[7]: Utilizamos o método 'count' que recebe um parâmetro de busca e retorna o número de elementos encontrado.
- In[8]: Utilizamos o método 'remove' que recebe um parâmetro de busca e remove a primeira ocorrência encontrada.
- In[9]: Visualizamos a lista com a alteração realizada no passo anterior.

Anotações

- ✓ Como uma lista, trata-se de uma sequência de itens de qualquer tipo.
- ✓ Entretanto, tuplas são imutáveis, isso as diferencia de listas. Sintaticamente, uma tupla consiste uma sequência de valores separados por vírgula.
- ✓ Apesar de não ser necessário, há o entendimento de envolver uma tupla entre parêntese.

Anotações

```
In [1]: linguagens = ('python', 'java', 'golang')
In [2]: linguagens
Out[2]: ('python', 'java', 'golang')

In [3]: linguagens[0]
Out[3]: 'python'

In [4]: linguagens[0].title()
Out[4]: 'Python'

In [5]: linguagens[1] = 'javascript'
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-5-a0572093153a> in <module>()
----> 1 linguagens[1] = 'javascript'

TypeError: 'tuple' object does not support item assignment
```

Tipos de dados: Tuplas

In[1]: Atribuímos uma tupla de strings em uma variável.

In[2]: Listamos seus valores.

In[3]: Acessamos seus valores por índice.

In[4]: Utilizamos um método de string, no valor acessado da tupla.

In[5]: Tentamos alterar um elemento da tupla, recebemos uma mensagem de erro, pois as tuplas são imutáveis.

Anotações

- ✓ Constituem um coleção desordenada de objetos.
- ✓ Representados na forma de chave. Esta, é usada para referenciar um determinado valor.
- ✓ As chaves de um dicionário são de tipo imutável como: inteiros, floats e strings.
- ✓ Não possuem uma noção de índice, não podem ser fatiados.
- ✓ São mutáveis, a qualquer momento é possível inserir ou remover itens.

Anotações

```
In [1]: ling_favorita = {'joao':'java', 'daniel':'python', 'hector':'php'}
```

```
In [2]: ling_favorita['daniel']
```

```
Out[2]: 'python'
```

```
In [3]: ling_favorita['hector'] = 'python'
```

```
In [4]: ling_favorita
```

```
Out[4]: {'joao': 'java', 'daniel': 'python', 'hector': 'python'}
```

```
In [5]: ling_favorita.get('joao')
```

```
Out[5]: 'java'
```

```
In [6]: ling_favorita.keys()
```

```
Out[6]: dict_keys(['joao', 'daniel', 'hector'])
```

```
In [7]: ling_favorita.values()
```

```
Out[7]: dict_values(['java', 'python', 'python'])
```

```
In [8]: ling_favorita.items()
```

```
Out[8]: dict_items([('joao', 'java'), ('daniel', 'python'), ('hector', 'python')])
```

Tipos de dados: Dicionários

In[1]: Atribuímos um dicionário em uma variável

In[2]: Acessamos um valor específico, utilizando a chave.

In[3]: Atribuímos um novo valor, especificando a chave que receberá a alteração.

In[4]: Listamos o dicionário para visualizar alterações efetuadas no passo anterior.

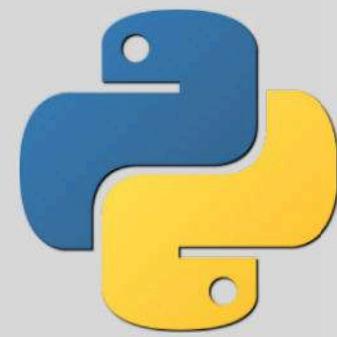
In[5]: Utilizamos o método 'get' para efetuar uma busca no dicionário.

In[6]: Utilizamos o método 'keys' para listar as chaves do dicionário.

In[7]: Utilizamos o método 'values' para listar os valores do dicionário.

In[8]: Utilizamos o método 'items' para listar chaves e valores do dicionário.

Anotações



Python Fundamentals

Estrutura de Dados e Conversão

2.3

Anotações

Objetivos da Aula

- ✓ Entender a Estrutura de dados.
- ✓ Aprender as conversões de tipos.

Estrutura de dados e Conversão de tipos

Nesta aula apresentaremos a Estrutura dos dados, seus métodos e a conversão de tipos.

Anotações

- ✓ Em Python podemos definir dois tipos de estruturas: sequências e dicionários.
- ✓ Sequências são dados ordenados, finitos e acessados através de um índice.
- ✓ Dicionários são objetos mapeados através de chaves.

Anotações

Podemos concatenar strings utilizando o método “format”.

```
In [1]: nome, idade = 'Guido', 62
In [2]: mensagem = ' O nome do criador do Python é {0} e sua idade é {1}'.format(nome, idade)
In [3]: mensagem
Out[3]: ' O nome do criador do Python é Guido e sua idade é 62'
```

Concatenando Strings

In [1]: Atribuímos dois valores em duas variáveis, observe que podemos atribuir inúmeras variáveis de uma vez em uma só linha. Neste caso, nome = ‘guido’ e idade = 62

In [2]: Utilizamos o método ‘format’ para concatenar string, dentro da string utilize a sintaxe de {} que serão substituídas por variáveis, que serão passadas dentro do método, neste exemplo, passamos parâmetros posicionais, nas próximas aulas aprenderemos a passar parâmetros nomeados.

In [3]: Visualizamos a variável para confirmar as alterações do passo anterior.

Anotações

Outra forma para concatenar uma string, utiliza o operador **(+)**, porém, em python não é possível concatenar dados de tipos diferentes com este operador.

```
In [1]: nome, idade = 'Guido', 62
In [2]: msg1 = 'O nome do criador do Python é ' + nome + ' e sua idade é ' + str(idade)
In [3]: msg1
Out[3]: 'O nome do criador do Python é Guido e sua idade é 62'
```

Concatenando Strings

In [1]: Atribuímos dois valores em variáveis.

In [2]: Utilizamos o operador **'+'**, para concatenar as strings, porém, a variável **idade** do tipo inteiro requer a conversão para **'str'** de maneira a evitar erro na concatenação, atribuímos o resultado final em uma variável.

In [3]: Exibimos a variável para visualizar as alterações do passo anterior.

Anotações

Conversão de tipos

```
In [1]: num = '2018'

In [2]: num.isnumeric()
Out[2]: True

In [3]: float(num)
Out[3]: 2018.0

In [4]: str(num)
Out[4]: '2018'

In [5]: num = int(num)

In [6]: num
Out[6]: 2018

In [7]: bin(num)
Out[7]: '0b11111100010'

In [8]: hex(num)
Out[8]: '0x7e2'
```

Conversões de tipos

In [1]: Atribuímos uma string em uma variável.

In [2]: Utilizamos o método de string 'isnumeric', para verificar se a string é um valor numérico com possibilidade de conversão.

In [3]: Convertemos a string para float.

In [4]: Embora a variável seja string, podemos converter também números e outros tipos para string, utilizando a função str.

In [5]: Atribuímos uma conversão permanente para inteiro.

In [6]: Verificamos a alteração do passo anterior.

In [7]: Convertemos o número para binário

In [8]: Convertemos o número para hexadecimal.

Anotações

Conversão de tipos

```
In [1]: letras = ['a', 'b', 'c', 'd']

In [2]: tuple(letras)
Out[2]: ('a', 'b', 'c', 'd')

In [3]: ling_favorita = {'joao':'javascript', 'hector':'php', 'daniel':'python'}

In [4]: list(ling_favorita.keys())
Out[4]: ['joao', 'hector', 'daniel']

In [5]: list(ling_favorita.values())
Out[5]: ['javascript', 'php', 'python']

In [6]: ling_favorita = list(ling_favorita.items())

In [7]: ling_favorita
Out[7]: [('joao', 'javascript'), ('hector', 'php'), ('daniel', 'python')]

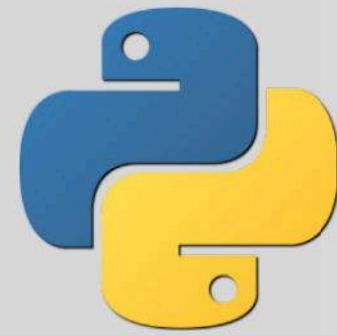
In [8]: type(ling_favorita[0])
Out[8]: tuple

In [9]: dict(ling_favorita)
Out[9]: {'joao': 'javascript', 'hector': 'php', 'daniel': 'python'}
```

Conversões de tipos

- In [1]: Atribuímos uma lista de strings em uma variável.
- In [2]: Convertemos a lista para o formato tupla.
- In [3]: Atribuímos um dicionário em uma variável.
- In [4]: Convertemos as chaves do dicionário em lista.
- In [5]: Convertemos os valores do dicionário em lista.
- In [6]: Convertemos chaves e valores do dicionário, em lista e atribuímos em uma variável.
- In [7]: Verificamos a variável para visualizar as alterações do passo anterior.
- In [8]: Verificamos o tipo de dado do índice, 0 da lista, que retorna uma tupla. Logo, temos uma lista de tuplas.
- In [9]: Convertemos nossa lista de tuplas para dicionário.

Anotações



Python Fundamentals

Operadores e Estrutura de Condição

3.1

Anotações

Objetivos da Aula

- ✓ Conhecer os operadores.
- ✓ Conhecer estrutura de condição.

Anotações

Tipos de Operadores

No Python temos três tipos de operadores:

Aritméticos: utilizados para realizar algum tipo de cálculo.

Relacionais: empregados para comparações, retornando true ou false em seus resultados.

Lógicos: usados para testes condicionais.

Anotações

Operadores aritméticos

```
#!/usr/bin/python3
num1 = 5;
num2 = 4;
mult = num1 * num2; # Para a Multiplicação(*)
adic = num1 + num2; # Para a Adição(+)
subt = num1 - num2; # Para a Subtração(-)
divi = num1 / num2; # Para a Divisão(/)
modu = num1 % num2; # Para o Módulo(%)
```

Anotações

Operadores aritméticos — Forma abreviada

```
#!/usr/bin/python3
number = 1; # A variável number recebe 1
number += 2; # Somamos 2 a variável
number -= 2; # Subtraímos 2 a variável
number *= 2; # Multiplicamos a 2 a variável
number /= 2; # Dividimos por 2 a variável
number %= 2; # Resto da divisão por 2
```

Anotações

Operadores relacionais

<code>==</code>	= Igual
<code>!=</code>	= Não igual ou Diferente
<code><</code>	= Menor
<code><=</code>	= Menor ou igual
<code>></code>	= Maior
<code>>=</code>	= Maior ou igual
<code>is</code>	= Compara Booleano

Anotações

Operadores e Estrutura de condição

Os operadores lógicos são:

```
var1 and var2: # retorna True se var1 e var2  
                  forem TRUE;  
var1 or var2:  # retorna True se num1 ou num2  
                  forem TRUE;  
not var:        # retorna TRUE se var for False;  
num1 == num2:   # retorna True se num1 e num2  
                  forem iguais;  
num1 != num2:   # True se num1 for diferente de  
                  num2 forem TRUE;
```

Operadores

Constituem meios para manipular dois valores (operandos), como por exemplo: $5 + 10 = 15$. Neste caso, 5 e 10 são chamados de operandos, o sinal + é chamado de operador. Os operadores nos ajudam principalmente quando trabalhamos com estruturas de decisão.

Anotações

Estruturas de Decisão

Estruturas de decisão, permitem manipular o fluxo de execução de código em uma aplicação, tendo como base um teste lógico, que espera um valor verdadeiro ou falso.

```
se condição faça  
    comandos;  
caso contrário faça  
    comandos;
```

```
if condição:  
    print("verdadeiro")  
else:  
    print("falso")
```

Estruturas de decisão

As estruturas de controle, provarão uma determinada condição, caso esta seja verdadeira, executará o bloco do “if”, se for falsa executará o bloco do “else”. Os comandos a serem executados, caso a condição seja verdadeira ou falsa, devem estar identados “dentro da condição”.

Maiores informações sobre o if/else podem ser consultadas em:

https://docs.python.org/3/reference/compound_stmts.html#if

Anotações

Estruturas de Decisão

A estrutura de decisão **IF** testará uma condição, caso seja verdadeira, o que estiver dentro do seu bloco de código será executado. Caso seja Falso, o bloco **ELSE** é executado.

```
#!/usr/bin/python3
idade = 18
if idade == 18:
    print('Você é maior de idade')
else:
    print('Você é menor de idade')
```

Anotações

Utilizando operadores

Podemos atender mais de um requisito em uma estrutura de decisão utilizando operadores **and** e **or**.

```
#!/usr/bin/python3
idade = 18
habilitacao = True
if idade >= 18 and habilitacao == True:
    print('Você pode dirigir')
else:
    print('Você não pode dirigir')
```

Estruturas de decisão com operadores

Podemos criar outras estruturas de decisão combinando mais de um tipo. Por exemplo: pense, para dirigir você precisa ter no mínimo 18 anos precisa ter carteira de habilitação ou, estar acompanhado dos pais. O código para esta situação é semelhante ao que segue:

```
#!/usr/bin/python3
idade = 22
carteira = False
pais = True
if idade >= 18 and carteira == True or pais == True:
    print('Você pode dirigir')
else:
    print('Você não pode dirigir')
```

Encadeamento de condição

Em programação, existe o encadeamento de condições, ou seja, podemos ter diversos **ifs** dentro de uma estrutura.

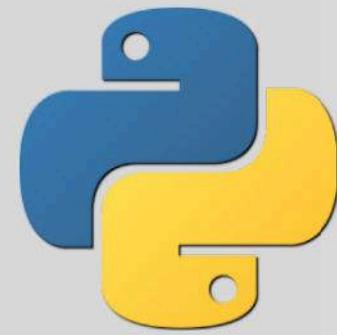
```
se(condição) faça  
    comandos;  
Senão  
    se(condição)  
        comandos;  
    senão  
        comandos  
fim
```

```
if condição:  
    print('verdadeiro')  
elif condicao:  
    print('falso')  
else:  
    print('Faça isso')
```

Encadeamento de condição

Usamos elif para verificar mais de uma condição, precisamos deste comando quando é necessário entrar somente em um dos ifs encadeados. Caso precise mais de um, é preciso fazer um if dentro do bloco de código do outro.

```
#!/usr/bin/python3  
  
idade = 17  
carteira = False  
pais = True  
  
if idade >= 18 and carteira == True:  
    print ('Você pode dirigir')  
elif idade == 17 and pais == True:  
    print ('Você pode dirigir em emergencias')  
else:  
    print ('Você não pode dirigir')
```



Python Fundamentals

Laços de Repetição For e While

3.2

Anotações

Objetivos da Aula

- ✓ Conhecer os laços de repetição.
- ✓ Entender a diferença entre **for** e **while**.

Conhecer os laços de repetição for e while

Nesta aula apresentaremos os laços de repetição for e while, entenderemos as diferenças entre eles e criaremos alguns exemplos práticos.

Anotações

while

While permite a execução de um bloco de código, desde que a expressão que foi passada como parâmetro seja verdadeira.

```
enquanto (condição) faça  
    comandos;  
fim enquanto
```

```
while condição:  
    comandos
```



Atenção: é muito comum não atentarmos acerca de como o laço irá terminar. Para que termine, precisamos fazer com que a condição se torne falsa em algum momento.

Laço de repetição while

Nos laços de repetição while, devemos evitar a situação de loops infinitos, estes ocorrem quando em algum momento uma condição torna-se falsa. Em alguns casos, estes loops fazem parte da lógica do nosso script, aprenderemos a utilização de outras formas de condições que interrompem o loop.

Anotações

Utilizando o while

Podemos utilizar o while da seguinte forma:

```
#!/usr/bin/python3
x = 1
while x < 10:
    print("Número: {}".format(x))
    x += 1
print("O while acabou !")
```

Operadores

Ainda conseguimos definir o while e passar um True para que execute, porém, não é o mais indicado. Exemplo:

```
#!/usr/bin/python3
x = 1
while True:
    print ("Número: {}!".format(x))
    x += 1
```

Em Python não existe o laço de repetição, DO WHILE, como em outras linguagens, na PEP 0315 o autor explica o porquê da não implementação: <https://www.python.org/dev/peps/pep-0315/>

Anotações

for

A instrução **for** é perfeita ao trabalhar-se com listas no python, pode ser utilizada de maneira tradicional, partindo de um valor inicial para outro valor final.

```
para(valor/condicao)
    comandos;
fimpara
```

```
for item in lista:
    print(item)
```



Atenção: **for** é utilizado para executar um montante fixo. Por exemplo: percorrer uma lista ou executar até certa quantidade de vezes.

Anotações

Utilizando o for

Podemos utilizar **for** da seguinte maneira:

```
#!/usr/bin/python3
fruta = [ "Laranja", "Melancia", "Uva" ]

for f in fruta:
    print(f)
```

Anotações

Com **for** podemos utilizar uma função chamada “**range**”, retornará uma sequencia:

```
#!/usr/bin/python3
for i in range(10, 0, -1):
    print (i)
```

Sua sintaxe será a seguinte:

```
for i in range(inicio, fim, incremento)
```

Operadores

A função range() consiste uma maneira, muito mais fácil, para gerar uma sequência de números sem utilizar o while.

Assim como no while, os valores são gerados um por um.

Ainda com o for, podemos utilizar a função enumerate(), para trazer as informações como um indice:

```
#!/usr/bin/python3
fruta = ["Laranja", "Abacaxi", "Uva" ]
for num,item in enumerate(fruta):
    print ("{} esta na posicao {}".format(item,num))
```

Anotações



Python Fundamentals

Controle de Loop

3.3

Anotações

Objetivos da Aula

- ✓ Conhecer Break e Continue.
- ✓ Compreender a utilização do Else do loop.

Controle de Loop

Nesta aula conheceremos o controle de loop com break e continue, compreendendo também o else do loop.

Anotações

Break e Continue

- ✓ Você pode enfrentar situação na qual precise sair de um loop quando uma condição externa é acionada.
- ✓ Também pode ocorrer circunstância, em que deseja ignorar uma parte do loop e, iniciar a próxima execução.
- ✓ Python fornece as instruções break e continue para lidar com tais situações, mantendo bom controle em seu loop.

Anotações

Break

- ✓ No Python esta instrução encerra o loop atual e, retoma a execução na próxima instrução, semelhante a quebra tradicional encontrada em C.
- ✓ O uso mais comum de quebra, ocorre quando alguma condição externa é acionada, exigindo a saída rápida de um loop.
- ✓ A instrução break pode ser usada em loops while e for.

Interrompendo um loop

In [1]: Definimos uma variável que será utilizada como contador, para o bloco de repetição.

In [2]: Definimos um bloco de repetição, com uma condição de contador menor que 10 e, o contador com incremento de + 1 por execução do loop, exibindo uma mensagem para visualizar quantidade de execuções.

Definimos um bloco de condição no interior do bloco de repetição, assim, quando o valor do contador for igual a 2, o loop será interrompido, deste modo, o loop será executado 3 vezes ao invés de 10, primeira execução para contador igual a 0 , em seguida 1, 2 e interrompe.

Anotações

Break

```
In [1]: cont = 0
In [2]: while cont < 10:
...:     print('vezes de execução {}'.format(cont + 1))
...:     if cont == 2:
...:         print('bloco de condição que interrompe o loop')
...:         break
...:     cont += 1
...:
vezes de execução 1
vezes de execução 2
vezes de execução 3
bloco de condição que interrompe o loop
```

Interrompendo um loop

In [1]: Definimos uma variável que será utilizada como contador, para o bloco de repetição.

In [2]: Definimos um bloco de repetição, com uma condição de contador menor que 10 e, o contador com incremento de + 1 por execução do loop, exibindo uma mensagem para visualizar quantidade de execuções.

Definimos um bloco de condição no interior do bloco de repetição, assim, quando o valor do contador for igual a 2, o loop será interrompido, neste modo, o loop será executado 3 vezes ao invés de 10, primeira execução para contador igual a 0 , em seguida 1, 2 e interrompe.

Anotações

Continue

- ✓ No Python, continue, retorna o controle para o início do loop while.
- ✓ Esta instrução, rejeita todas as demais restantes na iteração atual do loop e, move o controle de volta para o início do loop.
- ✓ Continue pode ser usada em loops while e for.

Ignorando instruções do loop

In [1]: Atribuímos uma lista vazia na variável.

In [2]: Definimos um loop numérico de 0 a 20, colocamos uma estrutura de condição na qual, o resto da divisão do número por 2, igual a zero, retorna ao início do loop, ignorando a instrução abaixo que insere o número na lista vazia, deste modo, serão inclusos os números ímpares.

In [3]: Arrolamos a lista com a alteração realizada no loop do passo anterior.

Anotações

Continue

```
In [1]: impar = []

In [2]: for num in range(20):
...:     if num % 2 == 0:
...:         continue
...:     impar.append(num)
...:

In [3]: impar
Out[3]: [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

Ignorando instruções do loop

In [1]: Atribuímos uma lista vazia na variável.

In [2]: Definimos um loop numérico de 0 a 20, colocamos uma estrutura de condição na qual, o resto da divisão do número por 2, igual a zero, retorna ao início do loop, ignorando a instrução abaixo que insere o número na lista vazia, deste modo, serão inclusos os números ímpares.

In [3]: Arrolamos a lista com a alteração realizada no loop do passo anterior.

Anotações

Else do loop

- ✓ Python suporta uso de uma instrução else associada a uma instrução de loop.
- ✓ Se for usada com um loop for, else será executada quando o loop tiver esgotado a iteração da lista.
- ✓ Se for usada com um loop while, else será executada quando a condição se tornar-se falsa.

Anotações

```
In [1]: nomes = ['maria', 'joao', 'pedro']

In [2]: busca = input('qual nome deseja procurar? ')
qual nome deseja procurar? maria

In [3]: for nome in nomes:
...:     if busca == nome:
...:         print('Está na lista')
...:         break
...:     else:
...:         print('Não está na Lista')
...:
Está na lista
```

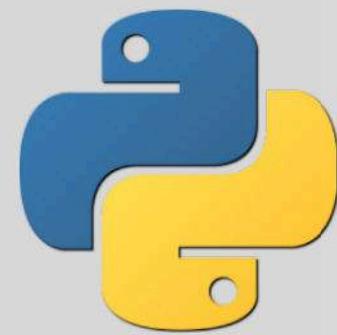
Else do loop

In [1]: Definimos uma lista de nomes (strings).

In[2]: Interagimos uma entrada com a função input, atribuindo a entrada em uma variável.

In[3]: Percorremos a lista de nomes, efetuando uma condição, se a busca for igual ao nome, exibir uma mensagem que encontrou, caso o loop não seja interrompido, executar o bloco ELSE com a mensagem não encontrou.

Anotações



Python Fundamentals

Erros e Exceções

4.1

Anotações

Objetivos da Aula

- ✓ Compreender o que são Erros e Exceções.
- ✓ Trabalhar com Try / Except / Finally.

Erros e Exceções

Nesta aula apresentaremos o conceito sobre erros e exceções, veremos quais as suas diferenças, como o Python trabalha com cada tipo e implementaremos o tratamento de erro em nossa aplicação.

Anotações

Em Python os erros são divididos em dois grupos: erros de sintaxe e exceções.

Os erros de sintaxe ou, erros de análise, consistem o tipo mais comum quando estamos aprendendo. Acontecem ao executar o script, o interpretador exibe uma linha indicando “SyntaxError” com uma “seta” apontando onde o erro foi encontrado.

```
if 'Mariana' == nome
```

^

```
SyntaxError: invalid syntax
```

Anotações

Exceções

- ✓ São geradas pela aplicação, mesmo que a sintaxe esteja correta, indicam que está ocorrendo algum erro ao executá-la.
- ✓ Exceções podem ser fatais, ou seja, sua aplicação continuará funcionando, mesmo que ocorra alguma, porém sempre devemos tratá-las.

Anotações

- ✓ Exceções não são tratadas automaticamente, porém exibem mensagens de erro que podem nos ajudar. Exemplo:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

Exceções

Um exemplo: tentar fazer uma comparação com uma variável, porém sem definir o que será a variável, veja:

```
#!/usr/bin/python3
if 'mariana' == nome:
    print('nome correto')
else:
    print('nome errado')
```

Ao executar o script, a seguinte exceção será gerada:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'nome' is not defined
```

A última mensagem indica o que ocorreu. Mostra o tipo da exceção, seguida de maiores detalhes sobre a ocorrência. O tipo da exceção é “NameError”, sugere que a variável “nome” não foi definida.

Tratamento de exceções

- ✓ Existem diversos tipos de exceções que conseguimos manipular, evitando, caso ocorram em um trecho de código, que o nosso script seja finalizado.
- ✓ Para isto, usaremos o try / except:

```
try:  
    comandos  
except Except_Type:  
    comandos
```

```
tente:  
    comandos  
exceção tipo_da_exceção:  
    comandos
```

Como funciona o Try/Except

- 1 - Os comandos que fazem parte do try são executados.
- 2 - Caso nenhuma exceção ocorra, o except é ignorado e a execução é concluída.
- 3 - Ocorrendo alguma exceção durante a execução do try, o resto do código é ignorado, se o Exception Type corresponder ao ocorrido, o except é executado.
- 4 - Caso ocorra erro e o Exception Type não corresponder, a execução do script será parada.

O principal objetivo ao colocar código dentro de um try / except, é garantir que continue sendo executado caso ocorra algum erro.

Anotações

Utilizando o try/except

```
#!/usr/bin/python3

while True:

    try:

        x = int(input("digite o primeiro numero: "))

        y = int(input("digite o segundo numero: "))

        print (x + y)

    except Exception as e:

        print ("Digite apenas números")
```

Anotações

Finally

- ✓ O try/except possui uma opção destinada a executar independente da ocorrência ou não de exception.
- ✓ Para implementar o finally, adicionamos abaixo da Exception:

```
finally:
```

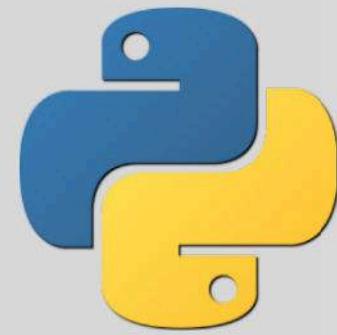
```
    comandos
```

O “**finally**” será executado independente da ocorrência ou não de exceção no seu código. Significa, seja qual for o resultado do código, o que está abaixo dele será executado. Podemos implementar o finally no exemplo anterior da seguinte forma:

```
try:  
    x = int(input("digite o primeiro numero: "))  
    y = int(input("digite o segundo numero: "))  
    print (x + y )  
except Exception as e:  
    print("Digite apenas números")  
Finally:  
    print("Saindo do script")
```

Quando implementamos log em nossa aplicação, as linhas de código podem ficar no finally. Independentemente de ocorrer erro ou execução com sucesso, o log será gravado e assim, teremos um controle dos horários da execução ou do erro acontecido.

Anotações



Python Fundamentals

Exception Types

4.2

Anotações

Objetivos da Aula

- ✓ Conhecer os Exception Types.
- ✓ Trabalhar com Exception Types.

Exception Types

Nesta aula iremos apresentar alguns dos Exception Types existentes em Python e como podemos trabalhar com eles.

Anotações

Exception Types

Quando definimos um bloco de código utilizando o try/except temos duas opções: podemos definir uma **exception genérica**, e qualquer exceção cairá neste bloco ou, podemos definir **exception específica** para determinadas situações.

O melhor, é definirmos as exceptions específicas, assim conseguimos maior controle da nossa aplicação.

Anotações

Trabalhando com Exception Types

Para definir uma exceção genérica:

```
try:  
    x = 5  
    y = 0  
    z = x/y  
  
except Exception as e:  
    print ('Erro: {}'.format(e))
```

Anotações

Trabalhando com Exception Types

Podemos utilizar Exception Types específicas, por exemplo:

NameError: quando o nome de uma variável não é encontrado.

TypeError: quando o tipo do objeto é inapropriado.

IndexError: quando o índice de um dicionário não é encontrado.

KeyError: quando a chave de um dicionário não é encontrada.

Anotações

NameError

```
#!/usr/bin/python3

try:
    print ('A Linguagem é: {}'.format(linguagem))

except NameError as e:
    print(e)
```

Anotações

TypeError

```
#!/usr/bin/python3

try:
    numero = 10
    print('O número é: ' +
numero)

except TypeError as e:
    print(e)
```

Anotações

IndexError

```
#!/usr/bin/python3

try:
    linguagem = ['python']
    print ('A Linguagem é: {}'.format(linguagem[2]))

except IndexError as e:
    print ('Erro: {}'.format(e))
```

Anotações

KeyError

```
#!/usr/bin/python3

try:
    linguagem = {'curso': 'fundamentals'}
    print ('A Linguagem é: {}'.format(linguagem['nome']))

except KeyError as e:
    print ('Erro: {}'.format(e))
```

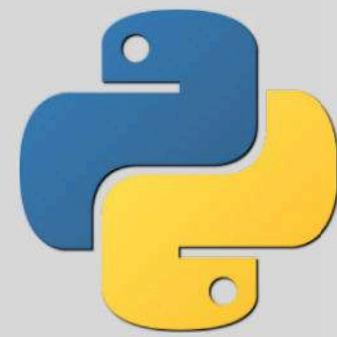
Anotações

Definindo mais de um Exception Type:

Podemos definir mais de um Exception Type, observe:

```
except NameError as e:  
    print ('Erro: {}'.format(e))  
except KeyError as e:  
    print ('Erro: {}'.format(e))
```

Anotações



Python Fundamentals

Raise Exception

4.3

Anotações

Objetivos

- ✓ Compreender Raise Exception.
- ✓ Trabalhar com Raise.

Raise

Nesta aula iremos entender o que são as Raise Exceptions e como podemos trabalhar com elas.

Anotações

No Python podemos criar nossas próprias exceções, possibilitam a aplicação ou script tratar determinada situação sem parar a execução.

A declaração raise permite criar estas exceções, ou seja, forço o disparo de uma exceção para o meu sistema.

Por exemplo, podemos criar uma verificação de id de usuário, caso este número retorne vazio, criaremos uma exceção e desta forma a nossa aplicação continuará funcionando.

Anotações

- ✓ Raise aceita qualquer tipo de Type Error, semelhante ao Except.
- ✓ Para que funcione como desejamos, o mesmo Type Error deverá ser definido em raise e no Except.
- ✓ Normalmente ao utilizar raise, o colocamos em um bloco de código if/else.
- ✓ Caso o retorno não seja o esperado, forçará uma exceção. o script continuará rodando.

Anotações

Sintaxe

```
try:  
    comandos  
    raise Except_Type  
except Except_Type:  
    comandos / raise
```

```
tente:  
    comandos  
    criando tipo_da_exceção  
exceção tipo_da_exceção:  
    comandos
```

Anotações

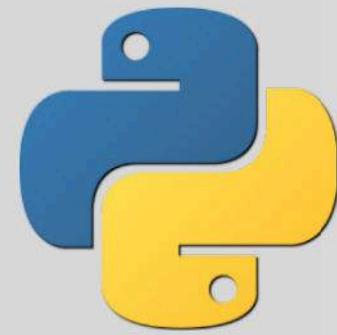
Trabalhando com raise

```
nome = 'python'
try:
    if nome == '4linux':
        print ('o nome esta correto')
    else:
        raise ValueError('O campo nome não
                          é correspondente')
except ValueError as e:
    print (e)
```

Raise

Definimos um bloco de condição caso o variável não tenha o valor que esperamos, criamos uma exceção e logo em seguida tratamos a mesma.

Anotações



Python Fundamentals

Manipulação de Arquivos

4.4

Anotações

Objetivos

- ✓ Compreender leitura de arquivos.
- ✓ Compreender gravação de dados.

Manipulação de Arquivos

Nesta aula trabalharemos com arquivos, veremos métodos disponíveis para realizar a leitura e escrita em arquivos de texto.

Anotações

- ✓ Para trabalhar com arquivos, teremos que criar um objeto que nos disponibilizará métodos como read ou write.
- ✓ O acesso ou leitura ao arquivo dependerá do modo em que o objeto foi criado.
- ✓ Existem duas categorias de objetos de arquivo: arquivos binários e arquivos de texto.

Manipulação de Arquivos

Arquivos binários: um objeto de arquivo do tipo binário, é capaz de ler e escrever em bytes, exemplo: gzip ou exe. Para abrir este tipo de arquivo, utilizamos: 'rb', 'wb' ou 'rb+'.

Arquivos de texto: este tipo de arquivo é capaz de ler e escrever objetos do tipo “string” que podem ser abertos com os seguintes comandos: 'r', 'w' ou 'r+'.

Modos e trabalho com o arquivo.

r – utilizada para leitura de arquivos.

w – usada para a escrita em arquivos.

r+ – empregada para a leitura e escrita em arquivos.

Em arquivos binários colocamos o “b” na frente dos modos, para especificar que o arquivo é deste tipo.

Anotações

- ✓ Todos os dados usados até agora foram inseridos pelo usuário ou colocados diretamente no código.
- ✓ Ao trabalhar com arquivos, a sintaxe será:

```
open(nome_do_arquivo, 'modo')
```

- ✓ “**open()**” retornará um objeto de arquivo, sempre utilizado seguindo o padrão: nome do arquivo e o modo, leitura ou escrita.

Anotações

Modos disponíveis

Os modos disponíveis para trabalhar com arquivos são:

Podemos combinar alguns modos.
Exemplo: r+ (abrirá o arquivo, permitindo leitura e escrita).

Modo	Significado
'r'	Abre o arquivo para leitura
'w'	Abre o arquivo para escrita (sobrescreve)
'x'	Abre para criação (falha caso o arquivo exista)
'a'	Abre para escrita (acrescenta no arquivo)
'+'	Abre um arquivo para atualização (leitura e escrita)

Modos de abertura

Podemos utilizar outros modos quando trabalhamos com arquivos. Há um modo específico para trabalhar com arquivos do tipo binário.

A documentação do Python 3 mostra como utilizar este modo:

<https://docs.python.org/3/library/functions.html#open>

Anotações

Escrevendo em arquivos

```
#/usr/bin/python3  
with open('arquivo', 'w') as f:  
    f.write('Curso de Python')
```



Desta forma os dados serão sobreescritos. Este modo é indicado para gravar os dados inicialmente.

Trabalhando com arquivos

Quando trabalhamos com arquivos, não é obrigatório o uso do “**with**”. Porém, trata-se de uma boa Prática que recomendamos utilizar. Elimina a necessidade de fechar o arquivo após finalizar a execução do necessário, esta é a principal vantagem que nos trás.

Para fazer a gravação sem utilizar o “**with**” podemos fazer da seguinte forma:

```
f = open( 'workfile' , 'w' )  
f.write('Curso de Python')  
f.close()
```

Note, é necessário fechar o arquivo após fazer a inserção dos dados.

Anotações

Podemos escrever em um determinado ponto do arquivo:

```
#/usr/bin/python3  
with open('arquivo', 'w') as f:  
    f.seek(0)  
    f.write('Curso de Python')
```

“seek” definirá a posição do meu ponteiro, onde 0 (zero) indica a primeira linha.

Anotações

Lendo arquivos

```
#/usr/bin/python3
with open('arquivo', 'r+') as f:
    print(f.read())
```

read permite ler todo o arquivo. Também contamos com **readlines** que trará todo o conteúdo em forma de lista.

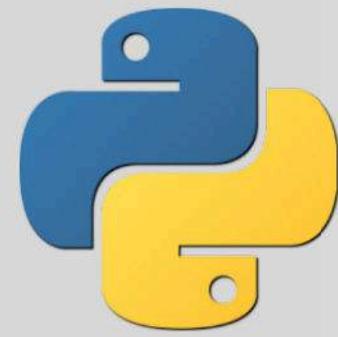
Lendo arquivos

Podemos empregar os dois métodos ao utilizar conteúdo a partir de arquivos de texto. Para obter uma linha específica do arquivo, podemos fazer da seguinte forma:

```
/usr/bin/python3
with open('teste', 'r+') as f:
    b = f.readlines()
    print b[1]
```

Como o retorno de `readlines` é uma lista, o acesso a um objeto se dá através do índice, a partir de então, consigo ter o retorno de uma só linha.

Anotações



Python Fundamentals

Escopo e Sintaxe

5.1

Anotações

Objetivos da Aula

- ✓ Entender funções.
- ✓ Definir funções.
- ✓ Entender escopo de uma função.
- ✓ Definir funções de escopo local e global.

Escopo global e local

Nesta aula apresentaremos o conceito de funções, sua sintaxe, escopo e como trabalhar com variáveis globais e locais.

Anotações

Entendendo funções

- ✓ Função é uma sequência de instruções que realiza uma operação.
- ✓ Podemos definir nossas próprias funções para resolver um problema em nossa aplicação.
- ✓ É possível utilizar qualquer nome para a nossa função.
- ✓ Porém, PEP8 recomendado que os nomes sejam sempre escritos em minúsculo.
- ✓ Em caso de caso de necessidade, nome separado por underline `_`, facilitando a legibilidade.

Entendendo funções

Podemos utilizar as funções para reaproveitamento de código, imagine ter uma aplicação que executará um determinado trecho de código. Há duas opções: reescrever diversas vezes ou criar uma estrutura que execute esta fração de código quando necessário.

Criar funções nos ajuda a deixar o código menor, mais organizado e até mais legível.

Para saber mais sobre funções, pesquise a documentação oficial:

<https://docs.python.org/3/tutorial/controlflow.html#define-functions>

Anotações

Criando funções

A sintaxe para definir uma função:

```
#!/usr/bin/python3
def nome_da_funcao(parametros):
    comandos
```



Uma função, pode ter a quantidade de comandos que for necessária. Contudo, lembramos que devem ser **identados** no escopo da função.

Criando funções

Ao definir um parâmetro, significa obrigatoriamente que a função espera receber algum dado. Se a função for implementada desta maneira e não passarmos nenhuma informação, ocorrerá erro. Para definir uma função sem nenhum parâmetro, utilize: def nome_da_funcao()
A seguir um exemplo de programa utilizando funções. A primeira, possui 1 parâmetro obrigatório. A segunda não recebe parâmetros.

Nas funções, **parâmetro** é o valor entre parênteses, o valor esperado. **Argumento** é o valor que passamos como parâmetro para a função. Por exemplo, a variável produto dentro do while.

```
#!/usr/bin/python3
produtos = []

def cadastrarProduto(produto):
    global produtos
    produtos.append(produto)

def listarProdutos():
    global produtos
    for p in produtos:
        print(p)

produto = ""
while produto != "sair":
    produto = input("Digite o produto que deseja cadastrar: ")
    cadastrarProduto(produto)
    print("produtos cadastrados")
    listarProdutos()
```

As funções também podem possuir parâmetros opcionais, que podem ser omitidos durante a chamada de função:

```
# /usr/bin/python3
def nome_funcao(parametro=padrao):
    comandos
```



Atenção: para que um parâmetro não seja obrigatório, deve possuir um valor padrão, que será assumido caso o valor não seja passado.

Funções com parâmetro padrão

Quando escrevemos uma aplicação, em geral utilizamos o “pass”. Um exemplo de função com parâmetro opcional: o cálculo de uma compra com cupom de desconto. O valor do desconto será calculado, caso seja informado um cupom válido.

```
#!/usr/bin/python
carrinho = []
produto1 = {"nome": "Tenis", "valor": 21.70}
produto2 = {"nome": "Meia", "valor": 10}
produto3 = {"nome": "Camiseta", "valor": 17.30}
produto4 = {"nome": "Calca", "valor": 100.00}
carrinho.append(produto1)
carrinho.append(produto2)
carrinho.append(produto3)
carrinho.append(produto4)

def totalCarrinho(carrinho):
    return sum(produto["valor"] for produto in carrinho)

def cupomDesconto(cupom=""):
    if cupom == "xyzgratis":
        return 0.50
    else:
        return 1

print("o total da sua compra e: ",
      (totalCarrinho(carrinho)*cupomDesconto()))
print("utilizando o cupom xyzgratis o valor sera de ",
      (totalCarrinho(carrinho)*cupomDesconto("xyzgratis")))
```

Ao definir uma função, é necessário escrever suas instruções. Porém, apenas quando estabelecemos a estrutura da nossa aplicação.

É comum utilizar a declaração “**pass**”, que ignora a função sem causar erros de sintaxe.

```
#!/usr/bin/python3
def nome_da_funcao(parametros):
    pass
```

Código da função

Outro benefício do emprego do “**pass**”, conseguimos dedicação a uma outra parte do código sem esquecer de retornar para implementar a funcionalidade determinada para aquela função. Quando usamos o “**pass**” o Python interpretará como, “não execute nada”. Desta maneira não acontecerá nenhum erro.

Conheça mais sobre o **pass** na documentação oficial:

<https://docs.python.org/3/tutorial/controlflow.html#pass-statements>

Anotações

Escopo da função

Uma função pode utilizar variáveis de escopo local. Assim podemos determinar a mesma variável em funções diferentes.

```
def sistema():
    nome = 'Linux'

def curso():
    nome = 'Python'
```

Escopo Local

Ao definir variáveis de escopo local, podemos utilizar a mesma variável em outra parte do código, não sendo sobreescrita. Variáveis definidas dentro de uma função, não podem alterar o valor de variáveis estabelecidas fora desta função.

Quando a aplicação está em execução, buscará as variáveis locais, que fazem parte da função e, caso não seja encontrada, pesquisará em variáveis globais.

```
#!/usr/bin/python3

servidor = "192.168.0.2"

def classe_a():
    servidor = '192.168.0.1'
    print(servidor)

def classe_b():
    print(servidor)

if __name__ == '__main__':
    print(servidor)
    classe_a()
    classe_b()
```

Escopo global

Funções podem utilizar variáveis de escopo global. Só é possível realizar alterações em uma variável global, ao especificar, na função, o emprego deste tipo de variável.

```
In [1]: var = 10
In [2]: def escopo():
...:     global var
...:     print(var)
...:     var = 5
...:     print(var)
...:
In [3]: escopo()
10
5
In [4]: var
Out[4]: 5
```

Escopo Global

In [1]: Definimos uma variável global.

In [2]: Estabelecemos uma função que lê uma variável global, exibe seu valor, depois altera seu valor e exibe novamente.

In[3]: Executamos a chamada da função.

In[4]: Exibindo a variável para confirmar os passos anteriores.

Anotações

Passando parâmetros para função

Passamos um parâmetro para a função, da seguinte forma:

```
#!/usr/bin/python3
def nome_da_funcao(parametro):
    print(x)
    nome_da_funcao('Aqui eu passo o parametro')
```

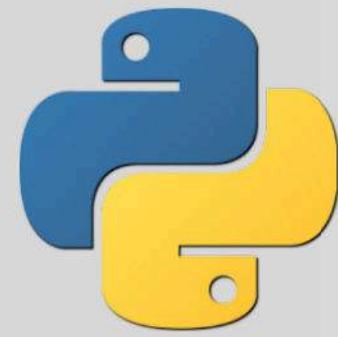
Parâmetros

Ao estabelecer uma função, optamos por receber ou não algum parâmetro, sendo possível passar mais de um parâmetro para a função, veja como:

```
#!/usr/bin/python3
def alterarServidor(atual,novo):
    print("O IP atual é: ",atual)
    print("O novo IP é: ",novo)

alterarServidor("192.168.100.0","10.10.10.1")
```

Anotações



Python Fundamentals

Args e kwargs

5.2

Anotações

Objetivos

- ✓ Entender Args.
- ✓ Entender Kwargs.

Args e Kwargs

Nesta aula apresentaremos Args e Kwargs, seu uso e o entendimento sobre como empregar estes parâmetros em nossas aplicações.

Anotações

Entendendo Args e Kwargs

- ✓ Quando não sabemos quantos argumentos serão passados a uma função, utilizamos *args ou **kwargs.
- ✓ ***args** transforma argumentos em uma tupla.
- ✓ ****kwargs** transforma argumentos em um dicionário.
- ✓ Não é necessário usar a nomenclatura args e kwargs.
- ✓ Os caracteres * ou ** realizarão a conversão,
- ✓ O uso destes caracteres é uma convenção.

Entendendo Args e Kwargs

O uso dos parâmetros *args e **kwargs é comum quando não sabemos a quantidade de argumentos que receberemos. A variação poderá ser de 1 até inúmeros.

Para conhecer mais sobre estes parâmetros, veja a documentação oficial:
<https://docs.python.org/3/tutorial/controlflow.html#unpacking-argument-lists>

Anotações

Args

Para criar uma função utilizando args:

```
#!/usr/bin/python3
def nome_da_funcao(*args):
    comandos
```



Desta forma, todos os argumentos passados serão convertidos em uma tupla.

Anotações

Utilizando o Args

```
#!/usr/bin/python3

def alterarServidor(*args):
    print ("O IP atual é: ",args[0])
    print ("O novo IP é: ",args[1])
alterarServidor("192.168.100.0","10.10.10.1")
```

Args

Ao empregar Args e receber os argumentos em formato de tupla, devemos trata-la em nossa aplicação para obter o resultado desejado.

Por exemplo: ao chamar uma função, informo os parâmetros: 4linux e python. Utilizo o conceito de tupla, para usar estas informações em minha aplicação da seguinte maneira. Usarei args[0] para exibir 4linux e, args[1] para mostrar python, lembre-se, que tuplas são acessadas através de índice.

No exemplo a seguir, apresentamos o calculo da área do quadrado e do retângulo utilizando o parâmetro *args.

```
#!/usr/bin/python3

def calcular(*args):
    if len(args) == 1:
        print ("A area do quadrado e: ",(args[0]*args[0]))
    elif len(args) == 2:
        print ("A area do retangulo e: ",(args[0]*args[1]))
    elif len(args) == 3:
        print ("A area do paralelepipedo e: ",(args[0]*args[1]*args[2]))

calcular(2)
calcular(4,2)
calcular(1,2,3)
```

Kwargs

Para criar uma função utilizando Kwargs:

```
#!/usr/bin/python3  
  
def nome_da_funcao(**kwargs):  
    comandos
```



Desta forma, todos os argumentos passados serão convertidos em um dicionário.

Anotações

Utilizando o Kwargs

```
#!/usr/bin/python3

def classe(**kwargs):
    print ("O IP atual é: ", kwargs["ip"])
    print ("O novo IP é: ", kwargs["novo"])
classe(ip="192.168.0.1", novo="10.10.0.1")
```

Kwargs

Ao utilizar Kwargs os argumentos serão recebidos em forma de dicionário (chave-valor), para usá-lo na aplicação, devemos acessar a sua chave e assim obter o seu valor.

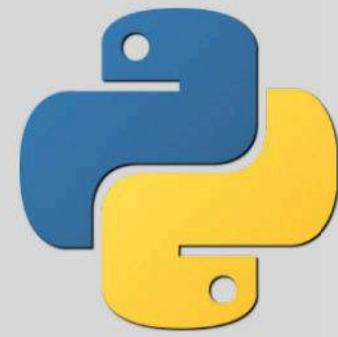
Por exemplo: ao chamar uma função, passo os seguintes parametros: sistema=linux, linguagem=python. Para utilizar estas informações na aplicação usando o conceito de dicionário, acesso os valores da seguinte forma: kwargs["linguagem"] e kwargs["sistema"].

Ainda é possível trabalhar com o dicionário como um todo, para isto, basta usar apenas "kwargs". No exemplo que segue, veremos todas as chaves e os valores do dicionário informado.

```
#!/usr/bin/python3

def descobreDicionario(**kwargs):
    print (kwargs)
    for k in kwargs.keys():
        print("Chave: {}".format(k))
        print ("Tem o valor: {}".format(kwargs[k]))

descobreDicionario(nome="servidor",
                    ip="192.168.0.1",
                    dominio="4linux.com.br")
```



Python Fundamentals

Funções Anônimas

5.3

Anotações

Objetivos

- ✓ Entender funções anônimas.
- ✓ Utilizar funções anônimas.

Funções Anônimas

Nesta aula exibiremos as funções anônimas, como funcionam e mostraremos exemplos.

Anotações

Entendendo Funções Anônimas

- ✓ Funções anônimas não estão vinculadas a um nome.
- ✓ Em Python podem ser chamadas como “expressões lambda”.
- ✓ Muito utilizadas no meio acadêmico na resolução de cálculos matemáticos.
- ✓ Funções lambda podem ser definidas dentro de uma função, sendo normalmente atribuídas a uma variável da função principal.

Entendendo Funções Anônimas

Em Python usamos a palavra reservada “lambda” para definir funções anônimas, utilizando esta palavra o retorno da expressão será um objeto de função, por este motivo, são chamadas de funções anônimas.

Podem ser usadas para cálculos matemáticos ou, quando precisamos de uma função que não seja tão complexa, possibilitando execução em uma única linha.

Anotações

Sintaxe

Ao definir uma função anônima, utilizamos a sintaxe:

```
lambda argumentos: expressão
```



Importante lembrar que “lambda” é uma palavra reservada em Python, para evitar erro, não devemos definir nenhuma variável com este nome.

Anotações

Utilizando Funções Anônimas

```
#!/usr/bin/python3  
  
var = lambda x: x*2  
  
print (var(2))
```

No exemplo, a função espera receber um argumento, (valor de x), para depois utilizar este número na multiplicação.

Utilizando Funções Anônimas

Uma função lambda é útil para resolver cálculos matemáticos. Um exemplo do mundo real, quando acontece a Black Friday um produto terá 50% de desconto, o código para essa situação pode ser

```
#!/usr/bin/python3  
  
carrinho = [ {"nome":"Tenis","valor":21.70},  
             {"nome":"Camiseta","valor":10.33} ]  
  
black_friday = lambda x: x / 2  
  
for c in carrinho:  
    print ("Nome do produto: ",c["nome"])  
    print ("Valor original: ",c["valor"])  
    print ("Valor com desconto: ",black_friday(c["valor"]))  
    print ("=====")
```

Passar mais de um número

Podemos passar mais de um número para fazer cálculos:

```
#!/usr/bin/python3

lamb = lambda a,b,c: ((b ** 2) - (4 * a * c))

print (lamb(3,-2,-5))
```

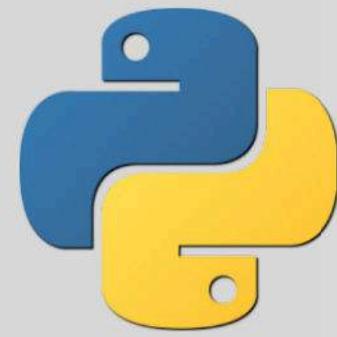
Anotações

Criando mais de uma função

Veja como fazer:

```
#!/usr/bin/python3
anonimas = [lambda x: x** 2,
            lambda x: x** 3,
            lambda x: x** 4]
for a in anonimas:
    print(a(10))
```

Anotações



Python Fundamentals

Orientação a Objeto

6.1

Anotações

Objetivos

- ✓ Compreender o que é Orientação a Objetos.
- ✓ Conhecer conceitos básicos.

Orientação a Objeto

Nesta aula abordaremos o conceito de Orientação a Objeto, porque é importante implementar POO (Programação Orientada a Objeto) em nossos projetos.

Anotações

- ✓ Orientação a objetos é um paradigma de programação, busca abstrair a programação para coisas do mundo real.
- ✓ Python é uma linguagem que aceita diversos paradigmas, no escopo de paradigmas, possui suporte maduro para desenvolvimento orientado a objetos.
- ✓ Em orientação a objetos, um software não é composto por um grande bloco de funcionalidades específicas, mas sim, por vários blocos com funcionalidades distintas e independentes que juntas formam um sistema.

Anotações

Origem

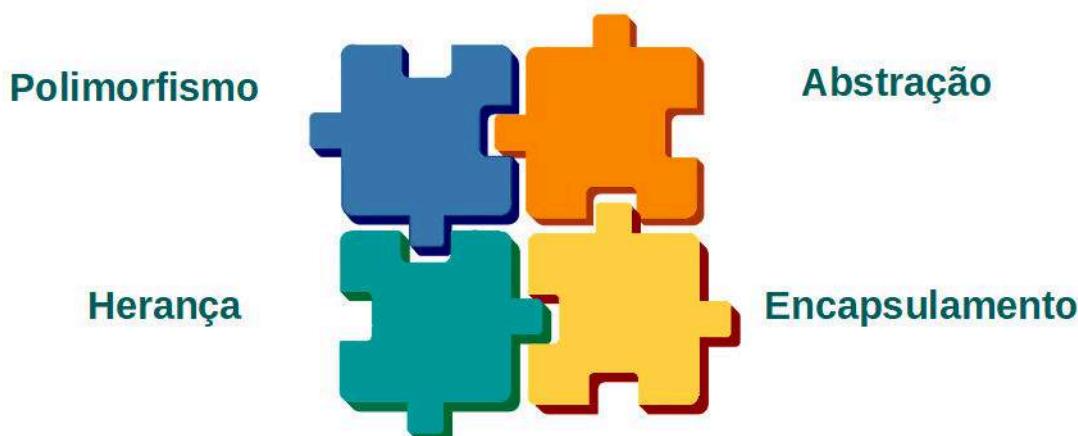
- ✓ Um dos criadores desse conceito foi Alan Kay.
- ✓ Possuía conhecimento em biologia, acreditava que um programa de computador poderia trabalhar como o corpo humano:
- ✓ células independentes que juntas trabalham para alcançar um objetivo.
- ✓ uma das primeiras linguagens orientadas a objetos foi o SmallTalk, tornou possível a criação da interface gráfica para os computadores.



Anotações

Características

Uma linguagem caracteriza-se como Orientada a Objetos quando atende aos quatro requisitos:



Características

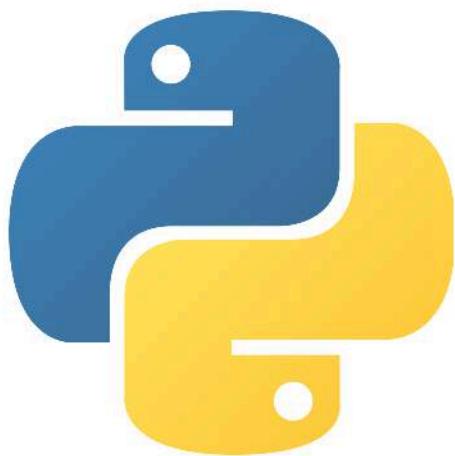
Abstração: utilizada para a definição de entidades do mundo real, onde são criadas as classes. Essas entidades são consideradas tudo que é real, levando em consideração suas características e ações.

Encapsulamento: técnica utilizada para esconder uma ideia, não expondo detalhes internos para o usuário. Torna as partes do sistema o mais independentes possível. Por exemplo, quando um controle remoto estraga, somente o controle é trocado ou consertado e não a televisão inteira. Neste exemplo, demonstra a forma clássica de encapsulamento, quando o usuário muda de canal, ao efetuar tal ação, não sabe que programação acontece entre o televisor e o controle.

Herança: possui o mesmo significado que no mundo real. Assim como um filho pode herdar alguma característica do pai, na Orientação a Objetos é permitido que uma classe herde atributos e métodos da outra. Há apenas uma restrição para a herança, os modificadores de acessos das classes, métodos e atributos, devem ter visibilidade public e protected para serem herdados.

Polimorfismo: princípio pelo qual duas ou mais classes derivadas de uma mesma superclasse, podem invocar métodos que têm a mesma identificação, assinatura, porém, comportamentos distintos, especializados para cada classe derivada, usando para tanto, uma referência a um objeto do tipo superclasse.

A linguagem Python foi criada e conformidade ao conceito de orientação a objetos, possibilitando utilizar vários desses recursos nativamente.



A principal função da Orientação a Objetos consiste o reuso do código, traz ainda outros benefícios como facilitar a organização do código e a sua legibilidade.

Anotações

Vantagens

Entre as vantagens da orientação a objetos, destacam-se:

- ✓ Reutilização de código poupando tempo, tanto programando quanto no debug.
- ✓ Escalabilidade, adicionar código mais facilmente.
- ✓ Manutenibilidade, código mais fácil de manter.
- ✓ Agilidade de desenvolvimento.

Anotações

Conceitos básicos

- ✓ Abstração de Dados.
- ✓ Classes.
- ✓ Objetos.
- ✓ Atributos / Propriedades.
- ✓ Métodos.
- ✓ Operador de Acesso.
- ✓ Assinatura.
- ✓ Parâmetros.
- ✓ Retorno.

Conceitos Básicos de Orientação a Objeto

Classes: Constitui a abstração de alguma coisa do mundo real, em forma de código. Exemplo: quando pensamos no mundo DevOps, temos em mente um servidor. Este, assemelha-se a nossa classe Principal.

Objetos: consiste uma instância dessa classe. Exemplo: um Servidor tem suas características como: endereço IP, usuários, serviços, etc.

Atributos/Propriedades: são as características de um determinado objeto. São descritos na classe e, cada objeto possui seus próprios valores para essas propriedades. Exemplo: no servidor nós temos o endereço IP, serviços sendo executados, memória, disco, CPU etc.

Anotações

Métodos: métodos são ações ou comportamentos que cada objeto possui. Exemplo: na classe servidor, podemos realizar acesso, alterar endereço Ip, criar usuários, instalar serviços, apagar arquivos, etc.

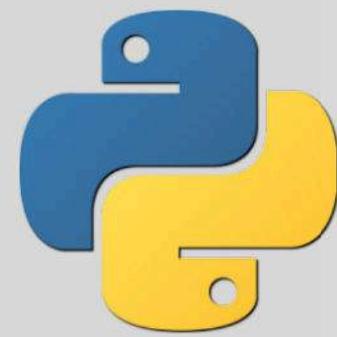
Operador de Acesso: para acessar os métodos e atributos de um objeto usamos o ponto (.), empregado na maioria das linguagens de programação. Os atributos e métodos de uma classe são realizados da mesma maneira, a única diferença é que nos métodos temos () parênteses no final.

Assinatura: significa o retorno do método. Exemplo: uma função que faz a soma de dois valores deve retornar um valor numérico, então a assinatura do método pode ser float, double ou integer.

Parâmetros: os métodos, assim como funções comuns, podem receber parâmetros. Os parâmetros podem não ser obrigatórios (endereço = None).

Retorno: quando queremos resgatar e trazer alguma informação. Usamos a palavra reservada `return` para resgatar o valor esperado.

Anotações



Python Fundamentals

Classes Propriedades e Método

6.2

Anotações

Objetivos

- ✓ Conhecer Classes e Objetos.
- ✓ Aprender Propriedades e Métodos.

Classes, propriedades e métodos

Nesta aula abordaremos o conceito de Classes, Objetos e trabalharemos com as definições de propriedades e métodos.

Anotações

Entender Classes e Objetos

Antes de iniciar a prática com Orientação a Objetos, é preciso entender algo que confunde os iniciantes.



Classes definem características e o comportamento dos seus objetos. Cada característica é representada por um atributo e, cada comportamento é estabelecido por um método. Porém, essêncial saber que uma classe **NÃO É** um objeto!

Entendendo o que são Classes e Objetos

Objetos são **entidades** que possuem um determinado **papel** num sistema. Na criação de um programa orientado à objetos, o primeiro passo, é determinar **quais** objetos existirão.

Mas como determino isso? Dependerá do sistema! Não existe uma receita, cada programador tem a sua visão e experiência. Dessa forma, julgam de modo diferente acerca dos objetos de um determinado sistema. Quanto mais complexo o sistema, mais difícil e diferente será essa análise. Observamos que em sistemas mais simples o juízo é mais uniforme.

Veja o exemplo de um programa para clínicas médicas:

*"O sistema deve cadastrar **pacientes**, **médicos**, e **consultas**. Será possível anexar **uma ou mais receitas na consulta** e, também possibilitará relacionar pacientes com **exames** e seus **resultados**."*

As palavras em negrito no texto, fazem referência às entidades observadas no sistema. Note, que mesmo em casos bem simplificados há divergências: resultado do exame, é ou não um objeto? Dependerá do programador.

Se observar que o resultado do exame desempenha um papel importante no sistema e/ou que por si só, possui atributos vinculados e comportamento separado, podemos estruturá-lo como um objeto.

Considerando que seja meramente uma propriedade do objeto exame, então poderá ficar fora.

- ✓ Classes são como a planta baixa de uma casa, a especificação técnica de uma cadeira ou o projeto de um veículo.
- ✓ A classe apenas estipula como será o objeto, mas não é o objeto em si.

Podemos construir 15 casas a partir de uma mesma planta baixa, Correto



Anotações

- ✓ Reconhecemos então, que todas essas “casas” são objetos produzidos a partir de uma classe.
- ✓ Dito de outra forma, os objetos = “casas”, possuem todas as características e comportamentos definidos na especificação da classe = “planta”.



Anotações



Portanto, uma classe NÃO é um objeto mas sim uma abstração de sua estrutura, na qual podemos definir as características que vão compor um objeto.

Anotações

Estrutura de uma classe

Para criar classes em Python, existe palavra reservada “class”:

```
class NomeDaClasse:  
    def metodo(self):  
        codigo
```

Anotações

Método construtor

Permite executar algo quando instanciamos uma classe.
Exemplo: setar atributos obrigatórios e defaults ou, exibir alguma mensagem na tela.

```
class NomeDaClasse:  
    def __init__(self):  
        atributos
```

Anotações

Atributos e métodos

Uma classe é composta de atributos e métodos, juntos criam a funcionalidade de um objeto.

- ✓ Atributos e métodos possuem o que chamamos de visibilidade.
- ✓ Métodos podem ou não, receber parâmetros que podem ou não ser obrigatórios.
- ✓ Métodos podem ou não retornar informação.

As características de um objeto, comumente chamadas de atributos ou propriedades, devem ser descritas na classe:

```
#!/usr/bin/python3

class Servidor:
    def __init__(self):
        self.cpu = None
        self.memoria = None
        self.disco = None
```

Definimos os atributos dentro do método construtor, referenciando o parâmetro `self` que alude a própria classe.

Anotações

Criando classe com atributos

```
class NomeDaClasse:  
    atributo = "valor"  
  
    def metodo(self):  
        pass
```

Anotações

Acessando Métodos e Atributos

- ✓ Em Python, utilizamos o ponto (.) para acessar atributos e métodos.
- ✓ Para acessar um método ou atributo, sempre usaremos uma instância da classe:

```
objeto = Classe()
```

A sintaxe para criação de atributos é a mesma para criação de variáveis.

```
#!/usr/bin/python3

class Servidor:
    memoria = None
    disco = None
    cpu = None

dns = Servidor()

dns.memoria = 2048
dns.disco = 50
dns.cpu = 2

print("O servidor tem as seguintes\
      configuracoes: CPU {},
      Memoria: {},
      Disco {} GB ".format(dns.cpu,dns.memoria,dns.disco))
```

Nas linhas destacadas, conferimos valores aos atributos, de modo que podemos resgatar seus valores posteriormente.

A saída deste código é: "o servidor tem as seguintes configurações:
CPU 2, Memoria: 2048, Disco 50 GB".

Lembre-se, os valores atribuídos à essas propriedades, podem ser de qualquer tipo, semelhante ao praticado com variáveis normais. Podemos atribuir um array, uma string, um número, etc.

Acessando Métodos e Atributos

```
class NomeClasse:  
    def __init__(self):  
        print("Acessando método construtor")  
    def metodo(self):  
        print("Acessando método")  
classe = NomeClasse()  
classe.metodo()
```

Anotações

Além dos atributos de um objeto, devemos colocar nessa estrutura, as ações que aquele objeto executará, seus Métodos.

Tais ações, diferentemente dos atributos, precisam ser declaradas e implementadas. Precisamos enumerar todas as ações do nosso objeto e codificar o que cada uma realizará, há exceção que será explicada posteriormente, em Métodos Abstratos.

```
#!/usr/bin/python3

class Servidor:
    memoria = 1024
    disco = 50
    cpu = 1

    def contratarMemoria(self,memoria):
        self.memoria += memoria

    def contratarCpu(self,cpu):
        self.cpu += cpu

    def contratarDisco(self,disco):
        self.disco += disco

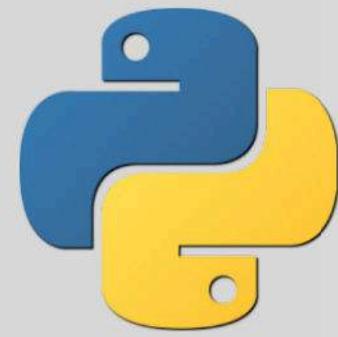
dns = Servidor()

dns.contratarMemoria(1024)
dns.contratarCpu(3)
dns.contratarDisco(50)

print ("O servidor tem as seguintes configurações:
       CPU {}, Memória: {}, Disco {} GB ".format(
dns.cpu,dns.memoria,dns.disco))
```

Os métodos, são chamados como as funções, porém, necessitamos, do nome do objeto na frente seguido por ponto. O primeiro parâmetro **self** é obrigatório, somente com ele o python consegue diferenciar métodos de funções .

Anotações



Python Fundamentals

Herança e Polimorfismo

6.3

Anotações

Objetivos da Aula

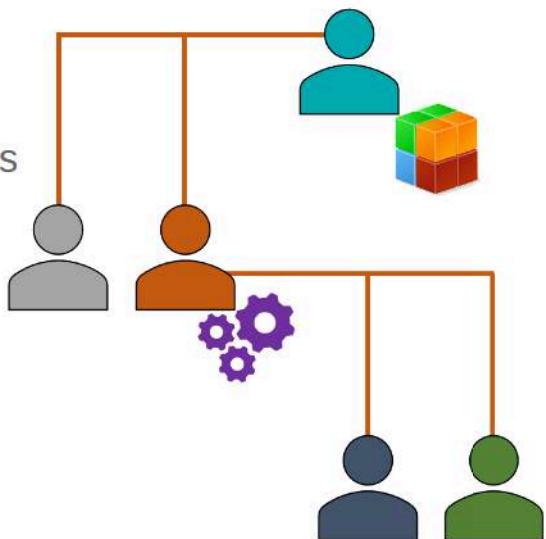
- ✓ Conhecer o conceito de Herança.
- ✓ Conhecer Herança Múltipla.
- ✓ Trabalhar com Polimorfismo.

Herança e Polimorfismo

Nesta aula você aprenderá os conceitos de Herança e Polimorfismo, com exemplos e prática, criará uma aplicação.

Anotações

- ✓ Herança de classes, é um dos conceitos fundamentais da orientação a objetos.
- ✓ Possibilita implementar uma série de outros conceitos, tornando mais claro o reúso de código.
- ✓ Permite que uma classe estenda outra.
- ✓ Assim, a classe filha, herdará todos os atributos e métodos da classe pai.
- ✓ Poderá então, possuir novos métodos e atributos, como reescrever métodos já existentes.



Entendendo Herança

Herança é um dos recursos mais interessantes da OO. Imagine que você está modelando seu sistema, observa que alguns objetos possuem comportamentos parecidos ou, até mesmo iguais. Algo a fazer para que a manutenção nesse código, parecido ,não seja muito complicada, seria centralizar tal código num único lugar. Facilitando assim, que a manutenção seja feita, em um único ponto.

Este é o conceito de generalização. Suponha a criação de um objeto Professor e um objeto Aluno. Se a modelagem tiver o básico de informações, notará que ambos possuem certas características e comportamentos iguais. Dito isto, podemos colocar tudo que for igual em uma nova classe, digamos Pessoa.

Deste modo, o funcionamento do sistema permanecerá o mesmo, em ambas as classes, Aluno e Professor, neste caso, será necessário colocar uma nova notação, indicando que ambas possuem como base a classe Pessoa.

Anotações

Define-se a Herança colocando o nome da classe pai como parâmetro da classe filha.

```
class NomeDaClasse(ClassePai):
```

- ✓ Todos os métodos e atributos estão na classe filha.
- ✓ O Python possui herança múltipla.
- ✓ É possível estender a partir de uma classe filha.

Anotações

Herança e Polimorfismo



```
class ClassePai:  
  
    def metodo(self):  
        print("Acessando a Classe Pai")  
  
class ClasseFilho(ClassePai):  
  
    def __init__(self):  
        print("Acessando a Classe Filho")  
  
classe = ClasseFilho()  
classe.metodo()
```



Entendendo Herança

A classe filho, herda todos os métodos e atributos da classe pai. Porém a classe filho, pode se estender obtendo seus próprios métodos e atributos que diferem da classe pai.
Criamos um objeto da instância filho que herda da pai. O método construtor foi definido na classe filho e logo exibe uma mensagem, também acessamos o método herdado da classe pai.

Anotações

Polimorfismo: significa muitas formas. Com esse recurso conseguimos ter métodos com nomes iguais, porém, executando coisas diferentes.

```
1 class ClassePai:
2     def metodo(self):
3         print("Método da classe pai")
4
5
6 class ClasseFilho(ClassePai):
7     def metodo(self):
8         print("Sobrescrevendo o método da classe pai.")
9
10
11 classe = ClasseFilho()
12 classe.metodo()
```

Polimorfismo

In [1]: definimos a instância da classe pai.

In [2]: estabelecemos o método da classe pai.

In [6]: determinamos a instância da classe filho, herdando da classe pai.

In [7]: sobrescrevemos o método herdado da classe pai realizando o polimorfismo, o qual as classes detém métodos com nomes iguais, fazendo coisas diferentes.

In [11]: criamos o objeto da instância filho.

In [12]: Verificamos o método no qual foi realizado o polimorfismo.

Anotações

Herança Múltipla

Utilizada quando uma classe precisa herdar os atributos e características de mais de uma classe.

```
1 class ClassePai:
2     a = 'caracteristica da classe pai'
3
4
5 class ClasseMae:
6     b = 'caracteristica da classe mae'
7
8
9 class ClasseFilho(ClassePai, ClasseMae):
10    c = 'caracteristica da classe filho'
11
12
13 classe = ClasseFilho()
14 print(classe.a, classe.b, classe.c)
```

Herança Múltipla

In [1]: definimos a instância da classe pai.

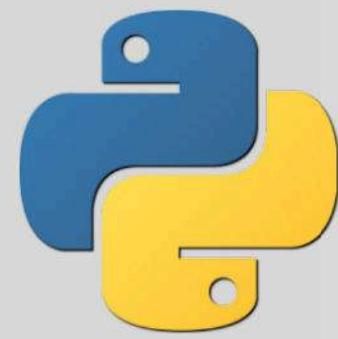
In [5]: determinamos a instância da classe mãe.

In [9]: estabelecemos a instância da classe filho, herdando da classe pai e mãe, realizando a herança múltipla.

In [13]: criamos o objeto da instância filho.

In [14]: verificamos a herança múltipla que herda de atributos de duas classes.

Anotações



Python Fundamentals

Módulos e Instaladores de Pacotes

7.1

Anotações

Objetivos

- ✓ Compreender a criação de módulos.
- ✓ Trabalhar com instalação utilizando o easy_install.
- ✓ Trabalhar com instalação utilizando o pip.

Instaladores de Pacotes

Nesta aula abordaremos o conceito de módulos, como cria-los, sua instalação via easy_install e pip.

Anotações

- ✓ Um módulo, nada mais é que um arquivo com várias funções utilizáveis em outra parte do código.
- ✓ No Python utilizamos alguns módulos já prontos ou criamos os nossos próprios módulos.
- ✓ É extremamente importante modularizar aplicações, desta forma reutilizamos os módulos em várias partes do projeto, facilitando manter e controlar.

Anotações

Crie um arquivo, com o nome module.py, defina duas funções:

```
#!/usr/bin/python3
def mod():
    print("Modulo 1")

def mod2():
    print("Modulo 2")
```

Anotações

Como usá-lo na aplicação. O primeiro passo, importar o módulo:

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

Anotações

Veja informações sobre o módulo:

```
>>> dir(module)
[ ... 'mod', 'mod2' ]
```

Chame o módulo:

```
>>> module.mod()
Modulo 1
```

Anotações

Para importar um módulo específico, execute:

```
>>> from module import mod  
>>> mod()  
Modulo 1
```

Anotações

Python dispõem de um repositório oficial chamado PyPI (Python Package Index), onde são disponibilizados módulos para uso em aplicação.

<https://pypi.python.org/pypi>



O PyPI (Python Package Index) possui uma lista enorme de pacotes utilizáveis. Atualmente existem mais de 100 mil pacotes disponíveis.

Existem módulos para fazer integração com diversas ferramentas do dia a dia como:

Jenkins: <https://pypi.python.org/pypi/python-jenkins/0.4.12>

GitLab: <https://pypi.python.org/pypi/pyapi-gitlab/7.8.5>

MongoDB: <https://pypi.python.org/pypi/pymongo/3.2>

Há módulos para trabalhar com testes:

Teste Funcional

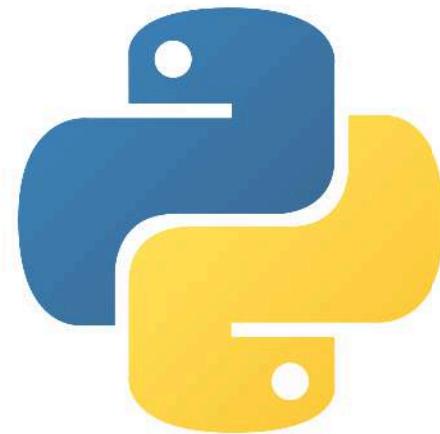
Selenium: <https://pypi.python.org/pypi/selenium>

Teste Unitário

Pytest: <https://pypi.python.org/pypi/pytest>

Utilize o pacote python-setuptools que disponibiliza o comando easy_install usado para a instalação dos módulos .

```
easy_install pip3  
easy_install -U pip3
```



O comando **easy_install** origina-se do pacote setuptools, desenvolvido em 2004 como a primeira ferramenta para instalação de pacotes do python. Logo destacou-se, devido a facilidade em conectar no repositório PyPI, resolvendo as dependências.

Anteriormente, a instalação acontecia baixando o código fonte do pacote e, utilizando os seguintes comandos para efetuar a instalação: python setup.py install

Anotações

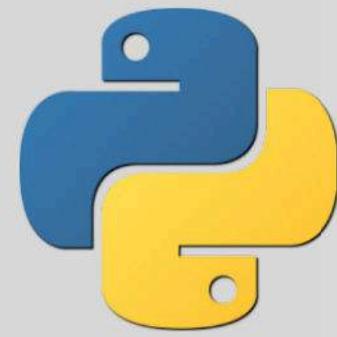
Outra maneira de instalar: use o pacote python3-pip que disponibiliza o comando pip.

```
pip install modulo = instala um módulo
pip search modulo = faz a busca pelo nome do módulo
pip list = lista todos os módulos instalados
pip uninstall = exclui um módulo
```

Como o comando **pip** foi baseado no pacote setuptools, tolera ser instalado através do comando `easy_install`.

pip foi desenvolvido em 2008 como alternativa ao `easy_install`, tornou-se bem popular devido a facilidade de uso. Além de possuir mais opções que o comando `easy_install`, é mais rápido, não realiza a instalação de um pacote a partir do zero, provê apenas os metadados com códigos fonte essenciais para o funcionamento do módulo.

Anotações



Python Fundamentals

Módulos Nativos

7.2

Anotações

Objetivos

- ✓ Compreender o que são Módulos Nativos.
- ✓ Conhecer os módulos os, sys, datetime, json e csv.

Módulos Nativos

Nesta aula apresentaremos módulos Nativos, você aprenderá sobre suas funcionalidades e porque utilizá-los em aplicações.

Anotações

- ✓ Em Python, criamos novos módulos utilizando módulos externos (prontos) ou empregamos módulos nativos.
- ✓ Os módulos nativos são aqueles embutidos na linguagem, não é necessário instalar, basta realizar o “import” e utilizá-los conforme a sua sintaxe.

Anotações

Os: possibilita usar funcionalidades do sistema operacional.

Sys: permite acessar parâmetros e funções específicas.

Datetime: traz alguns tipos de data e hora.

Json: codifica e decodifica no formato JSON.

Csv: lê e escreve arquivos no formato CSV (planilhas).

Anotações

- ✓ Utilize sempre ao criar um script que precise executar algum comando no linux.
- ✓ Exemplo: criar diretórios, ver informações do sistema, listar arquivos, ver informações de usuário, entre outras coisas.

Anotações

```
import os  
  
#ver qual o usuário logado  
  
print(os.getlogin())  
  
#listar o conteúdo de um diretório  
  
print(os.listdir('/caminho/do/diretorio'))  
  
#renomear um arquivo  
  
print(os.rename('nome_atual', 'novo_nome'))  
  
#executar qualquer comando  
  
os.system('digite o comando aqui')
```

Modulo OS

Utiliza-se este módulo quando se executa algo no sistema operacional Linux. Consiste o módulo Python que fará chamadas de sistema, executando o comando.

Utilizando “os.system(‘comando’), executamos o módulo. Porém, recomenda-se consultar na documentação a melhor forma para a execução, pode existir um modo específico para realizar o que planeja.

Para maiores informações sobre o módulo os, verifique a documentação oficial:
<https://docs.python.org/3/library/os.html#module-os>

Anotações

- ✓ Fornece acesso a algumas variáveis ou funções executadas pelo Python.
- ✓ Permite trabalhar com o interpretador, conseguimos ver a versão do Python, em que sistema está em execução, passar parâmetros, entre outras coisas.

Anotações

```
import sys

#ver em qual plataforma está executando o script
print(sys.platform)

#ver os módulos em execução pelo Python
print(sys.builtin_module_names)

#passar argumentos
print(sys.argv)

#finalizar o script
sys.exit()
```

Modulo Sys

Traz informações sobre o interpretador (python). Funcionalidade muito interessante, permite passar argumentos, veja o exemplo:

```
#!/usr/bin/python3

import sys


for i in range(len(sys.argv)):
    if i == 0:
        print("Function name: {}".format(sys.argv[0]))
    else:
        print("{} . argument: {}".format(i,sys.argv[i]))
```

`argv` retornará os argumentos como uma lista, ao passar 3 argumentos, para acessá-los: `sys.argv[1]`, `sys.argv[2]` e `sys.argv[3]`. `sys.argv[0]`, corresponde ao nome do Script.

Para conhecer mais sobre o módulo, veja a documentação oficial:
<https://docs.python.org/3/library/sys.html#module-sys>

- ✓ Fornece formas de manipular data e hora, de maneiras simples a complexas.
- ✓ É muito importante saber trabalhar com data e hora, por esta razão a linguagem oferece este módulo nativo.

Anotações

```
import datetime
#mostra a data atual
print(datetime.datetime.now())
# mostra a data após 7 dias
print(datetime.timedelta(7))
# definir um horário, por exemplo: 14hrs
print(datetime.time(14,0,0))
# definir uma data, por exemplo: 1 de janeiro de 2017
print(datetime.date(2017,1,1))
```

Modulo Datetime

Permite a manipulação de data e horário. Veja o exemplo, exibir token com validade de 1h:

```
#!/usr/bin/python3
from datetime import datetime
acesso = datetime(2018,01,22,00,00,00)
atual = datetime(2018,01,22,01,01,00)

if (atual - acesso).total_seconds() > 3600:
    print("Seu token expirou")
else:
    print("Acesso liberado")
```

Neste exemplo prático, de cálculo do término de tempo de um token de acesso, realizamos a subtração de 2 objetos Datetime, a operação retorna um outro objeto Datetime com a diferença de datas. Empregamos o método total_seconds, que converterá o resultado em segundos, comparando com 3600 segundos o equivalente a 1 hora. Se o total de segundos for superior 3600 o token será expirado.

Para saber mais sobre o módulo datetime, veja a documentação oficial:
<https://docs.python.org/3/library/datetime.html#module-datetime>

- ✓ Permite mudar determinados caracteres para o formato JSON e vice-versa.
- ✓ Módulo muito importante. Ao trabalhar com API's, normalmente usamos este formato, sendo necessário realizar a conversão para utilizar os dados na aplicação.

Anotações

```
import json  
  
#transformar o json em string  
print(json.dumps({ "nome": "python",  
                    "modulo": "fundamentals" })  
  
#transformar a string em json  
print(json.loads(' { "nome": "python",  
                    "modulo": "fundamentals" } '))
```

Modulo Json

A integração com outras ferramentas, normalmente requer uso de APIs para envio e recebimento de dados. É comum tais dados virem no formato JSON. Como um dicionário, as vezes a aplicação não combina com este tipo de formato, sendo necessário converter este JSON para string e vice-versa. No código apresentado, conferimos cada tipo.

```
#!/usr/bin/python3  
  
import json  
  
print(type(json.dumps({ "nome": "python",  
                        "modulo": "fundamentals" })))  
  
print(type(json.loads(' { "nome": "python",  
                        "modulo": "fundamentals" } ')))
```

Estude a documentação oficial para conhecer mais sobre o módulo json:
<https://docs.python.org/3/library/json.html#module-json>

- ✓ Viabiliza que um script trabalhe com planilhas, fazendo importação e exportação.
- ✓ Permite ler e escrever arquivos deste tipo, usando os dados em nossos scripts.

Anotações

```
import csv

with open('teste.csv', 'r') as csvfile:
    arquivo = csv.reader(csvfile, delimiter=' ')
    for a in arquivo:
        print(a)
```

Modulo csv

Assemelha-se a abertura de arquivos txt e binários do python. Porém, dispõe de parâmetros que facilitam o trabalho com este tipo de formato. Por exemplo o parâmetro 'delimiter', na leitura, determina qual separação de dados ocorrerá por vírgulas, espaços, pipes etc...

Anotações

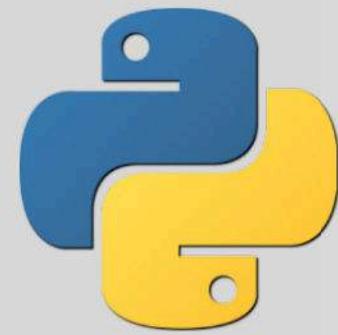
```
import csv

with open('arquivo.csv', 'w', newline='') as csvfile:
    arquivo = csv.writer(csvfile, delimiter=',')
    arquivo.writerow(['Teste'] * 5)
    arquivo.writerow(['4linux',
                      'Python'])
```

Modulo csv

Possibilita realizar métodos de escrita avançados, facilitando a inserção de dados formatados em massa que serão lidos por algum software (leitor de planilhas).

Anotações



Python Fundamentals

Ambientes Isolados

7.3

Anotações

Objetivos

- ✓ Compreender a utilidade de ambiente isolado.
- ✓ Conhecer o módulo virtualenv.

Ambiente isolado

Você compreenderá o que é ambiente isolado e a vantagem ao utilizar o modulo virutalenv.

Anotações

- ✓ Dispositivo que permite criar ambientes virtuais isolados para projetos Python.
- ✓ Por exemplo: possibilita criar ambientes virtuais para dois projetos, A e B, um ambiente para cada projeto.
- ✓ Ao criar os ambientes, por exemplo: venvA e venvB, serão gerados diretórios com o nome de cada ambiente.

Anotações

4LINUX / Virtualenv

189



3.3



2.6

VIRTUALENV



3.5



2.4

Anotações

- ✓ Viabiliza manter as duas versões do mesmo módulo sem causar conflito.
- ✓ Exemplo: uma aplicação requer a versão 1 do modulo (X), entretanto, outra aplicação usa o mesmo módulo da versão 2.

Anotações

1 Instale o Virtualenv através do gerenciador de pacotes pip:

```
# pip3 install virtualenv
```

2 Para criar o virtualenv, basta informar o interpretador default e o nome do virtualenv:

```
# virtualenv -p /usr/bin/python3 venv
```

3 Para ativar as variáveis do projeto criado, execute:

```
# source venv/bin/activate
```

4 Para desativar as variáveis e voltar a utilizar o bash padrão, execute:

```
# deactivate
```

Virtualenv

Constitui boa prática manter uma virtualenv para cada projeto com seus respectivos módulos, evitando conflito de versionamento.

Anotações

1 Para visualizar os módulos instalados:

```
# pip3 freeze
```

2 Direcionar todos os módulos do projeto para um arquivo txt:

```
# pip3 freeze > requirements.txt
```

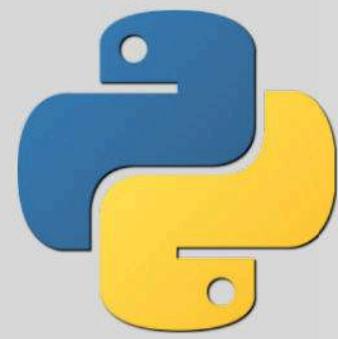
3 Instalar todos os módulos escritos no arquivo txt.

```
# pip3 install -r requirements.txt
```

Virtualenv

Consiste uma boa prática, manter em cada projeto um arquivo txt com seus respectivos módulos, assegurando que plataformas, até mesmo outros colaboradores do projeto, consigam instalar todos os módulos necessários para rodar a aplicação.

Anotações



Python Fundamentals

SQL

8.1

Anotações

Objetivos da Aula

- ✓ Entender o que são Banco de Dados.
- ✓ Introdução à linguagem SQL.

SQL

Nesta aula abordaremos conceitos, características e possibilidades de trabalho com a linguagem SQL.

Anotações

- ✓ Banco de dados é um conjunto de dados guardados para a extração de informações no futuro.
- ✓ A linguagem padrão universal para manipular banco de dados relacionais é o SQL (Structured Query Language).
- ✓ Esta linguagem permite interagir com um SGBD (Sistema Gerenciador de Banco de Dados) para executar tarefas como: inserir registros, gerenciar usuários, criar consultas, entre outras.



Introdução ao SQL

O conceito sobre Banco de Dados deve ser compreendido com clareza.

Exemplo: 09/06/2017

09/06/2017 é somente um dado, aparentemente não apresenta nenhum significado. Porém, se mudarmos o contexto para: O Cliente João foi cadastrado na data de 09/06/2017.

Atribuído um contexto para o dado extraído, traz informação com a qual podemos trabalhar. Sendo assim, utilizamos o banco de dados para conservar registros que recuperados em determinado contexto, geram informações que podem resultar em valor para o negócio.

Anotações

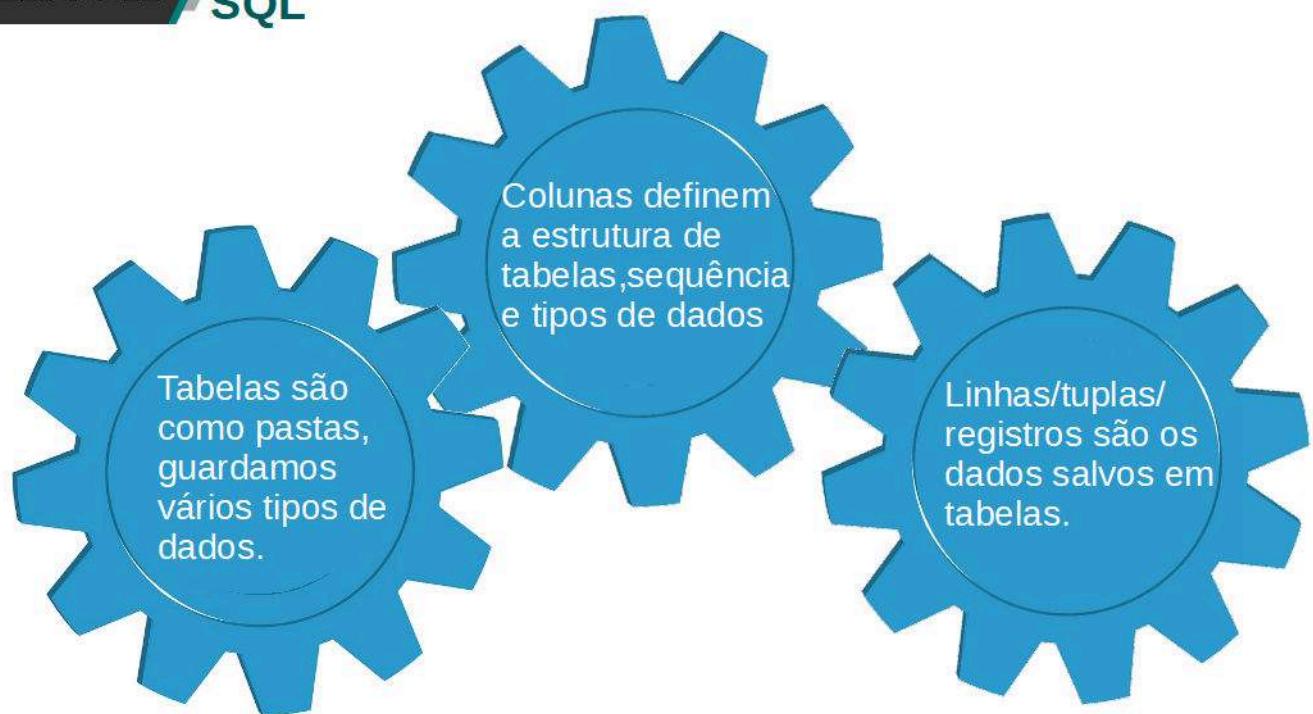
- ✓ Na computação utilizamos ferramentas SGBD (Sistemas Gerenciadores de Bancos de Dados) para armazenamento e o manuseio desses dados.
- ✓ Dados são organizados em tabelas, colunas e linhas.



Durante a maior parte do tempo como desenvolvedor, você criará ou encontrará aplicações que utilizam alguma forma de armazenamento de dados. Estes dados podem ser armazenados em simples arquivos, no sistema de arquivos de uma máquina ou, armazenados em estruturas mais complexas, como um banco de dados.

Bancos de dados, em sua grande maioria, são constituídos com base nas relações existentes entre suas entidades, daí o nome: Bancos de Dados Relacionais.

Anotações



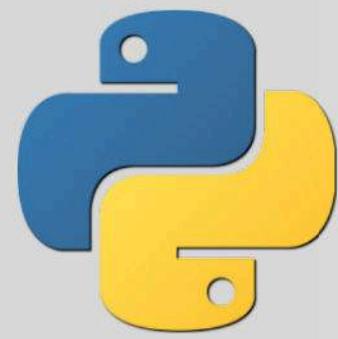
Anotações

DDL: Data Definition Language, para manipular a estrutura da tabela, com comandos como: CREATE, ALTER, DROP.

DML: Data Manipulation Language, manipulamos os dados, através de comandos como: INSERT, UPDATE e DELETE.

DQL: Data Query Language, para consulta de dados, por meio de comandos como: SELECT, SHOW.

Anotações



Python Fundamentals

PostgreSQL

8.2

Anotações

Objetivos da Aula

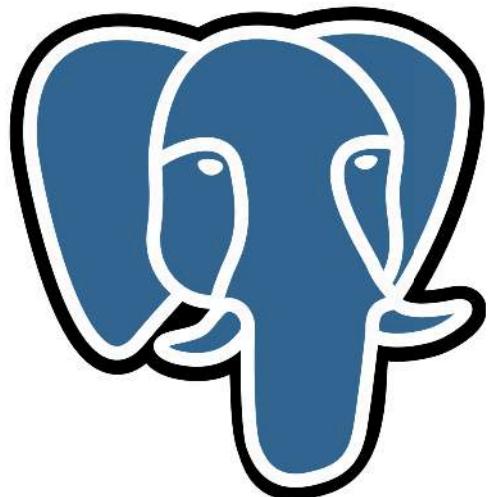
- ✓ Conhecer o PostgreSQL.
- ✓ Trabalhar com PostgreSQL.

PostgreSQL

Nesta aula você aprenderá como trabalhar com PostgreSQL, criando tabelas, inserindo dados e fazendo consultas.

Anotações

- ✓ Constitui um Sistema Gerenciador de Banco de Dados (SGBD) de código aberto.
- ✓ Otimizado para aplicações complexas, que trabalham informações críticas ou grandes volumes de dados.
- ✓ Exemplo: e-commerce de médio porte.
- ✓ PostgreSQL é mais indicado, por ser capaz de lidar com o volume da dados gerado pelas consultas de venda.



O PostgreSQL é um poderoso sistema gerenciador de banco de dados objeto-relacional de código aberto e 100% livre. Considerado o banco de dados livre, mais avançado do mundo, por várias Razões.

Suporta largamente os padrões ANSI-SQL 92/99 e respeita a normativa ACID. É altamente extensível, detém vários tipos de índices para diversos tipos de aplicações. Utilizado em larga escala por empresas como Skype, Caixa Econômica Federal e Datasus. Python trabalha muito bem com o PostgreSQL, possui módulos de trabalho, assim como ocorre com MySQL.

Anotações

Quando não há usuário criado, o sistema o disponibiliza para gerenciar o banco.

1 Logue com o usuário root:

```
$ su - root
```

2 Logue com usuário postgres e acesse o banco:

```
# su - postgres
```

```
# psql
```

Anotações

1 Crie um usuário para acesso a aplicação:

```
=# CREATE USER admin PASSWORD '4linux';
```

2 Crie um banco de dados para a aplicação:

```
=# CREATE DATABASE projeto OWNER admin;
```

3 Saia do sistema com o usuário postgres:

```
=# \q
```

4 Logue no banco denominado 'projeto' com usuário 'admin':

```
=# psql -h localhost -U admin projeto
```

Anotações

1 Verifique as tabelas criadas:

```
=# \dt
```

2 Gere uma tabela para inserir scripts:

```
=# CREATE TABLE scripts(id SERIAL, nome VARCHAR(50), conteudo TEXT);
```

3 Veja as informações sobre a tabela:

```
=# \d scripts
```

4 Veja o conteúdo da tabela:

```
=# SELECT * FROM scripts;
```

Anotações

1 Acrescente dados na tabela de scripts:

```
=# INSERT INTO script(nome, conteudo) VALUES ('hello.py', 'print ("script de teste")');
```

2 Faça update:

```
=# UPDATE scripts SET nome='update.py' WHERE id = 1;
```

3 Delete um registro:

```
=# DELETE FROM scripts WHERE id = 1;
```

4 Apague todos os registros:

```
=# TRUNCATE scripts;
```

Anotações

1 Para contar dados específicos, execute:

```
=# SELECT * FROM scripts WHERE id=1;
```

2 Para consultar dados por um intervalo, execute o seguinte comando:

```
=# SELECT * FROM scripts WHERE id BETWEEN 1 AND 4;
```

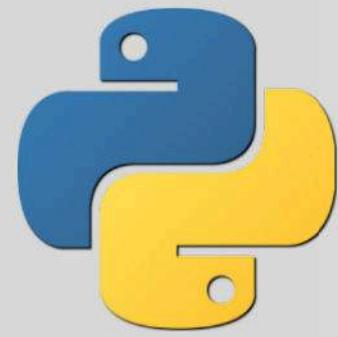
3 Consultas a dados por parte do campo, execute:

```
=# SELECT * FROM scripts WHERE nome LIKE '%.py';
```

4 Ordene dados de forma decrescente com o comando:

```
=# SELECT * FROM scripts WHERE nome LIKE '%.py%' ORDER BY id DESC;
```

Anotações



Python Fundamentals

MySQL

8.3

Anotações

Objetivos da Aula

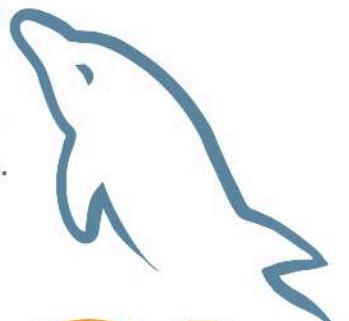
- ✓ Conhecer o MySQL.
- ✓ Trabalhar com MySQL.

MySQL

Nesta aula demonstraremos como trabalhar com MySQL criando tabelas, inserindo dados e fazendo consultas.

Anotações

- ✓ Consiste um dos SGBD's mais populares.
- ✓ Amplamente empregado em serviços de hospedagem de sites.
- ✓ Focado em agilidade, efetivo se a aplicação demanda retornos rápidos, não envolvendo operações complexas.
- ✓ Exemplo: armazenamento de conteúdo de site/fórum.



Anotações

1 Faça login no MySQL:
\$ mysql -u root -p

2 Verifique os bancos existentes:
> SHOW DATABASES;

3 Para criar um banco de dados, execute:
> CREATE DATABASE projetos;

4 Acesse a base:
> USE projetos;

Anotações

- 5 A fim de inserir usuário para administrar o banco, execute:
➢ GRANT ALL PRIVILEGES on projetos.* to admin@'localhost'
IDENTIFIED BY '4linux' WITH GRANT OPTION;

- 6 Atualize os acessos:
➢ FLUSH PRIVILEGES;

- 7 Saia do MySQL:
➢ EXIT;

- 6 Acesse com o novo usuário:
mysql -u admin -p

Anotações

1 Verifique as tabelas criadas:

> SHOW TABLES;

2 A fim de criar uma tabela para inserir clientes, execute:

> CREATE TABLE clientes(id INTEGER PRIMARY KEY NOT NULL AUTO_INCREMENT, nome VARCHAR(255), endereco VARCHAR(255));

3 Veja informações sobre a tabela:

> DESCRIBE clientes;

4 Para ver o conteúdo da tabela, execute:

> SELECT * FROM clientes;

Anotações

1 Para inserir dados na tabela de scripts, execute o seguinte comando:
➤ `INSERT INTO clientes(nome, endereco) VALUES ('Mariana', 'Rua Vergueiro');`

2 Para realizar update:
➤ `UPDATE clientes SET endereco='Paulista' WHERE nome = 'Mariana';`

3 Deletar um registro com o comando:
➤ `DELETE FROM clientes WHERE id = 1;`

4 Para apagar todos os registros:
➤ `TRUNCATE clientes;`

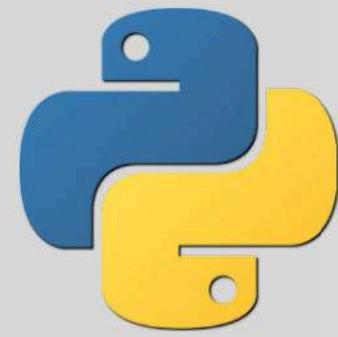
Anotações

- 1** Criar uma nova tabela chamada projetos, execute o comando:
>CREATE TABLE projetos(id INTEGER PRIMARY KEY AUTO_INCREMENT NOT NULL, nome VARCHAR(255) NOT NULL, cliente VARCHAR(255));

- 2** Faça a inserção de um novo registro:
> INSERT INTO projetos(nome,cliente) VALUES ('Virtualizacao', 'Mariana');

- 3** Verificar apenas o cliente que contratou algum projeto
>SELECT * FROM projetos WHERE cliente IN (SELECT nome FROM clientes);

Anotações



Python Fundamentals

8.4

Estrutura de Dados no MongoDB

Anotações

Objetivos da Aula

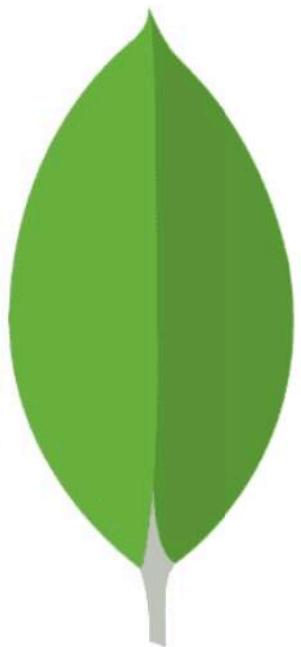
- ✓ Conhecer o MongoDB.
- ✓ Compreender sua estrutura de dados.

Estrutura de Dados no MongoDB

Nesta aula você conhecerá o MongoDB e a sua estrutura de dados.

Anotações

- ✓ Trata-se de um banco de dados do tipo noSQL (Not Only SQL).
- ✓ Os dados são armazenados no formato JSON, visando a alta performance.
- ✓ Bancos de dados noSQL, normalmente ocupam mais espaço em disco, uma vez que não dispõem do recurso JOIN entre os dados, sua leitura e escrita porém, são bem superiores.



Estrutura de Dados no MongoDB

MongoDB não é um substituto dos bancos relacionais, embora seja possível trabalhar somente com ele, sem a necessidade de outros bancos de dados.

A escolha pelo MongoDB, se dá com a necessidade de velocidade em leitura e escrita de dados. Em seu site, encontramos benchmarks mostrando sua performance, efetuando cerca de 270 mil operações de leitura e escrita por segundo.

Também é muito utilizado para o controle de filas. Existem muitos artigos na internet, mostrando como é possível substituir o RabbitMQ pelo MongoDB.

MongoDB Performance Testing:

<https://www.mongodb.com/blog/post/performance-testing-mongodb-30-part-1-throughput-improvements-measured-ycsb>

Anotações

- ✓ Trabalha com documentos do tipo Chave – Valor.
- ✓ Armazena documentos do tipo JSON, o formato mais usado atualmente para enviar dados na web através de API's que utilizam o protocolo REST.

Anotações

✓ Sua estrutura de armazenamento de dados também é diferente do SQL, então temos:

Collections: armazena os documentos, semelhante as tabelas nos bancos SQL.

Documentos: guarda os dados no formato JSON, equivale aos registros do SQL.

Em MongoDB, um registro é um documento, este, consiste uma estrutura de dados composta por chave e valor (como um dicionário em Python). Uma chave pode ter qualquer valor (até outro dicionário ou uma lista).

Exemplo:

```
{  
    nome: "Mariana",  
    cargo: Analista,  
    grupos: ["consultoria", "treinamento"]  
}
```

Para saber mais sobre collections, visite o site com a documentação oficial:
<https://docs.mongodb.com/v3.2/core/databases-and-collections/>

Anotações

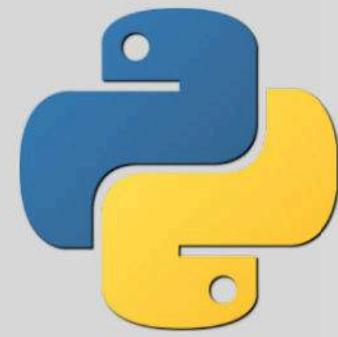
```
{  
    nome : "Daniel Prata" ,  
    projetos : [  
        {nome : "Automação Python" ,  
         descricao : "Projeto de  
automação"} ,  
  
        {nome : "API Bevops" ,  
         descricao : "Projeto de api"}  
    ]  
}
```

Anotações

Instale da seguinte maneira:

```
#fazer a instalação  
$apt-get install mongodb  
#iniciar o serviço  
$ service mongodb start  
#acessar o banco  
$ mongo
```

Anotações



Python Fundamentals

Comandos no MongoDB

8.5

Anotações

Objetivos da Aula

- ✓ Criar e excluir bases.
- ✓ Inserção, atualização, exclusão e consultas no MongoDB.

Trabalhar com comandos no MongoDB

Nesta aula você aprenderá como trabalhar com o MongoDB, criando collections, inserindo dados e fazendo as consultas.

Anotações

4LINUX / Utilizando o MongoDB

224

1 Acesse o MongoDB:
\$ mongo

2 Para ver os bancos criados, execute:
\$ > show dbs ou show databases

3 Para acessar ou criar um banco, use o seguinte comando:
\$ > use dexterops

4 Para deletar um banco, utilize o comando:
\$ > db.runCommand({dropDatabase:1})

No console do MongoDB, encontramos comandos criados para a compatibilidade com usuários do MySQL. Exemplo:

```
show databases  
show tables
```

O comando show tables foi criado especificamente para essa compatibilidade, já que esse banco de dados trabalha com documentos JSON ao invés de tabelas.

Recomenda-se cuidado ao digitar o nome do banco de dados, pois o MongoDB cria um novo banco de dados cada vez que for digitado um nome errado.

Sendo assim, é importante verificar os bancos existentes, sempre apagando as criações errôneas, evitando assim a existência do que chamamos de “sujeira” em banco de dados.

4LINUX / Utilizando o MongoDB

225

- 1 Verifique as collections com o comando:
 > show collections OU show tables

- 2 Para inserir um registro, utilize:
 > db.fila.insert({ "_id": 1, "servico": "Intranet", "status": 0 })

- 3 Remova um documento com o seguinte comando:
 > db.fila.remove({ "_id": 1 })

- 4 Para verificar documentos, execute:
 > db.fila.find()

Além dos comandos mostrados, contamos com os seguintes:

db.help() — mostra comandos para administrar o MongoDB.

db.<nome-da-collection>.help() — exibe as opções de comandos existentes para você administrar a sua collection.

show users — exibe os usuários criados. Por padrão mongoDB não vem com usuários criados.

Para criar um usuário digite o seguinte comando:

```
db.addUser(  
  { "user": "administrador",  
    "pwd": "4linux",  
    "roles": [ "readWrite", "dbAdmin" ]  
})
```

Na versão 2 do MongoDB o comando é: addUser().

1 Para verificar documentos de modo mais fácil quanto a leitura, execute:
`> db.file.find().pretty()`

2 Pesquisas com filtro, são realizadas com o comando:
`> db.fila.find({"_id":2}).pretty()`

3 Para fazer alterações no documento e apagar dados, execute:
`> db.fila.update({"_id":1}, {"status":1})`

4 Para fazer uma alteração no documento e permanecer com os dados, execute:
`> db.fila.update({"_id":1}, {"$set":{"status":1}})`

O parâmetro `_id` não é obrigatório, porém, caso não seja informado, MongoDB criará um id parecido com esse: `{ "_id" : ObjectId("569946c973fdfb8dacf877f3")}`

Esta ação, garante a unicidade dos documentos, possibilitando inserir o mesmo documento várias vezes sem informar o `_id`. MongoDB acrescentará esses documentos, fazendo com que pelo menos um parâmetro assegure a diferença entre eles.

Ao utilizar o `update`, nunca esquecer de usar a chave `$set` para atualizar um valor em específico. Caso a chave não seja informada, o documento que se pretende atualizar, será alterado para o parâmetro passado.

Exemplo:

```
db.file.update({"_id":1}, {"status":1})
```

O resultado será:

```
{"_id":1,"status":1}
```

Com o `$set` conforme o exemplo, o resultado será:

```
{"_id":1,"servico":"intranet","status":1}
```

Também importante, cuidado ao utilizar o método `remove`, se não forem especificados parâmetros, serão removidos todos os documentos.

```
db.fila.insert({ "empresa": "4linux",  
                 "cursos": [  
                     { "nome": "Python Fundamentals",  
                       "carga horaria": 40 },  
                     { "nome": "Linux Fundamentals",  
                       "carga horaria": 40 } ] })
```

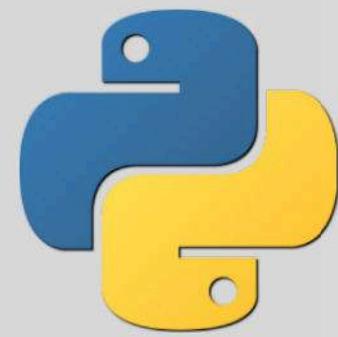
Anotações

```
db.fila.update({ "cursos.nome":  
    "Python Fundamentals"},  
    { "$set": { "cursos.$.nome": "Novo" } })
```

Anotações

```
db.fila.update({ "cursos.nome" :  
    "Python Fundamentals" },  
    { "$pull": { "cursos":  
        { "nome": "Python Fundamentals" } } })
```

Anotações



Python Fundamentals

Python com PostgreSQL

9.1

Anotações

Objetivos

- ✓ Conhecer o módulo para conexão.
- ✓ Fazer inserção, alteração, exclusão e consulta de dados.

Python com PostgreSQL

Nesta aula você aprenderá sobre o módulo de conexão com o PostgreSQL, como trabalhar, inserir dados e realizar consultas.

Anotações

Instale o módulo que dá suporte ao PostgreSQL.

```
sudo apt-get install python3-psycopg2
```

- ✓ Este módulo em específico, deve ser instalado pelo apt-get.
- ✓ Postgres precisa de outras bibliotecas do sistema operacional que o pip não consegue resolver, já que foi criado para resolver dependências do python e não de Linux.

Anotações

```
import psycopg2

con = psycopg2.connect("host=IP dbname=BANCO
                        user=USUARIO password=SENHA")
```

- ✓ Primeira, importa o módulo do postgres.
- ✓ Em seguida, abrimos uma conexão com o banco.
- ✓ Será utilizada para executar comandos dentro do banco.

Anotações

```
cur = con.cursor()
```

- ✓ A variável cur, recebe o cursor do banco de dados a partir da conexão.
- ✓ Este cursor possibilita efetuar as operações CRUD (Create, Retrieve, Update, Delete).

Anotações

```
cur.execute("insert into  
TABELA(COLUNA1,COLUNA2)  
values ('VALOR1','VALOR2'))  
con.commit()
```

- ✓ Nesta código a variável cur recebeu o cursor para navegar no banco, executamos o comando insert.
- ✓ Logo abaixo, a variável com é empregada para fazer o commit.
- ✓ Enquanto não for efetuado o commit, não será gravado nenhum dado no banco.

Segue um exemplo de inserção no banco de dados:

```
#!/usr/bin/python3  
  
import psycopg2  
  
try:  
    con = psycopg2.connect("host=127.0.0.1  
                           dbname=projeto  
                           user=admin  
                           password=4linux")  
    cur = con.cursor()  
    cur.execute("insert into scripts(nome,conteudo) values('devops',  
                                                       'projeto de python')")  
    con.commit()  
    print("Registro criado com sucesso")  
except Exception as e:  
    print("Erro: {}".format(e))  
    print("Fazendo rollback")  
    con.rollback()  
  
finally:  
    print("Finalizando conexão com o banco de dados")  
    cur.close()  
    con.close()
```

```
#atualizar um registro  
cur.execute("update TABELA set COLUNA=VALOR  
where COLUNA=VALOR")  
#deletar um registro  
cur.execute("delete from TABELA where  
COLUNA=VALOR")
```

- ✓ execute é utilizado para atualizar e deletar registros.
- ✓ Depende de commit para efetuar a modificação no banco.

Segue exemplo de update e delete no banco de dados:

```
#!/usr/bin/python3  
  
import psycopg2  
  
try:  
    con = psycopg2.connect("host=127.0.0.1 dbname=projeto  
                           user=admin password=4linux")  
    cur = con.cursor()  
    cur.execute("update scripts set nome='4linux' where id=1")  
    print("Registro atualizado com sucesso")  
    cur.execute("delete from scripts where id=2")  
    print("Registro removido com sucesso")  
    con.commit()  
except Exception as e:  
    print("Erro: {}".format(e))  
    print("Fazendo rollback")  
    con.rollback()  
  
finally:  
    print("Finalizando conexao com o banco de dados")  
    cur.close()  
    con.close()
```

```
#voltar o último comando  
cur.rollback()  
  
#fecha a conexão  
cur.close()  
con.close()
```

- ✓ **cur.rollback()** volta a última transação caso ocorra algum erro.
- ✓ **cur.close()** e **con.close()** finalizam a conexão com o banco.

Segue exemplo forçando um rollback quando digitamos um update errado:

```
#!/usr/bin/python3  
  
import psycopg2  
  
try:  
    con = psycopg2.connect("host=127.0.0.1  
                           dbname=projeto  
                           user=admin  
                           password=4linux")  
    cur = con.cursor()  
    cur.execute("insert into projeto(nome,conteudo) values('devops',  
                                                       'projeto de python')")  
    con.commit()  
    print("Registro criado com sucesso")  
except Exception as e:  
    print("Erro: {}".format(e))  
    print("Fazendo rollback")  
    con.rollback()  
  
finally:  
    print("Finalizando conexão com o banco de dados")  
    cur.close()  
    con.close()
```

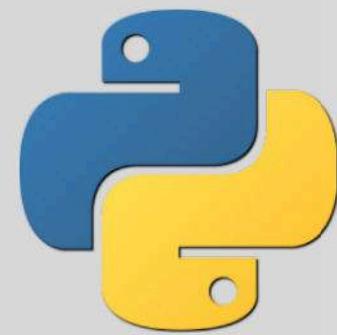
```
#fazer a busca  
cur.execute("select * from TABELA")  
#retornar o primeiro registro  
cur.fetchone()  
#retornar todos os registros  
cur.fetchall()
```

- ✓ **fetchone()**: realiza busca, retorna só o primeiro registro do banco.
- ✓ **fetchall()**: realiza busca retornando todos os registros do banco.

Abaixo temos um exemplo buscando o primeiro registro de uma tabela, na sequência todos os registros:

```
#!/usr/bin/python3  
  
import psycopg2  
  
try:  
    con = psycopg2.connect("host=127.0.0.1 dbname=projeto  
                           user=admin password=4linux")  
    cur = con.cursor()  
    cur.execute("select * from scripts")  
    print("O primeiro registro é: ", cur.fetchone())  
    print("Todos os registros: ", cur.fetchall())  
except Exception as e:  
    print("Erro: {}".format(e))  
    print("Fazendo rollback")  
  
finally:  
    print("Finalizando conexão com o banco de dados")  
    cur.close()  
    con.close()
```

Note que o `commit()` e o `rollback` foram removidos deste exemplo. No caso de querys, não são feitas alterações na base de dados, então eles não são necessários.



Python Fundamentals

Python com MySQL

9.2

Anotações

Objetivos da Aula

- ✓ Conhecer o módulo para conexão.
- ✓ Inserção, atualização, exclusão e consulta no banco.

Python com MySQL

Nesta aula apresentaremos o módulo de conexão com o banco de dados MySQL, mostrando como criar tabelas, inserir dados e realizar consultas através do Python.

Anotações

Instale módulo que dá suporte ao MySQL.

```
sudo apt-get install python3-mysqldb
```

- ✓ Este módulo em específico, será instalado pelo apt-get.
- ✓ MySQL precisa de outras bibliotecas do sistema operacional Linux que o pip não consegue resolver.

Anotações

Efetuando a conexão com um banco de dados MySQL:

```
import MySQLdb
con =
MySQLdb.connect(host="127.0.0.1", user="usuario",
passwd="senha", db="banco")
```

Python não possui função nativa para realizar conexão com o MySQL. Por isso, instalamos o módulo MySQLdb. Este módulo, fornece todos os comandos necessários para interagir com o banco de dados.

Em funções, explicamos o conceito de tratamento de exceções, essenciais quando trabalhamos com bancos de dados.

Podemos usar o bloco do try, para efetuar a conexão com o banco de dados, o bloco do except para exibir o erro, caso não efetue a conexão com o banco e o bloco de finally para finalizar a conexão, após efetuar as operações.

É uma boa prática finalizar as conexões do MySQL após utilizá-las. Por padrão, o mysql deixa uma conexão aberta durante 8 horas, então, caso não sejam fechadas após o seu uso, caso seu script tenha muitas conexões de banco, pode ocasionar o erro: MySQL server has gone away, causando a indisponibilidade do banco de dados.

No MySQL também é necessário cursor para interagir com o banco de dados.

```
cur = con.cursor()
```

- ✓ Utilizamos a variável cur que recebeu o cursor para navegar no banco de dados.

Anotações

con.commit(): grava os dados no banco.

```
cur.execute("insert into TABELA(COLUNA1,COLUNA2)
              values ('VALOR1','VALOR2')")
con.commit()
```

Segue um exemplo de inserção no banco de dados:

```
#!/usr/bin/python3

import MySQLdb

try:
    con = MySQLdb.connect(host="127.0.0.1",db="projetos",
                          user="admin",passwd="4linux")
    cur = con.cursor()
    cur.execute("insert into projetos(nome,cliente)
                  values('Python','Dexter Courier')")
    con.commit()
    print("Registro criado com sucesso")
except Exception as e:
    print("Erro: {}".format(e))
    con.rollback()

finally:
    print("Finalizando conexao com o banco de dados")
    cur.close()
    con.close()
```

O registro criado utilizando a DML Insert, ficará gravado na memória até que seja efetuado o commit. Se o python não encontrar este comando, não haverá a persistência de dados ao término da execução do script .

Podemos fazer update e delete:

```
#atualizar um registro
cur.execute("update TABELA set COLUNA=VALOR where
COLUNA=VALOR")
#deletar um registro
cur.execute("delete from TABELA where COLUNA=VALOR")
```

Segue exemplo de update e delete no banco de dados:

```
#!/usr/bin/python3

import MySQLdb

try:
    con = MySQLdb.connect(host="127.0.0.1",db="projetos",
                          user="admin",passwd="4linux")
    cur = con.cursor()
    cur.execute("update projetos set nome='4linux' where id=1")
    print('Registro atualizado com sucesso!')
    cur.execute("delete from projetos where id=2")
    print('Registro deletado com sucesso')
    con.commit()
    print("Registro criado com sucesso")
except Exception as e:
    print ("Erro: {}".format(e))
    con.rollback()

finally:
    print ("Finalizando conexao com o banco de dados")
    cur.close()
    con.close()
```

```
#voltar o último comando  
cur.rollback()  
#fecha a conexão  
cur.close()  
con.close()
```

- ✓ **cur.rollback()** volta a última transação caso ocorra algum erro.
- ✓ **cur.close()** e **con.close()** finalizam a conexão com o banco de dados.

Segue exemplo forçando um rollback quando digitamos um update errado:

```
#!/usr/bin/python3  
  
import MySQLdb  
  
try:  
    con = MySQLdb.connect(host="127.0.0.1",db="projetos",  
                          user="admin",passwd="4linux")  
    cur = con.cursor()  
    cur.execute("insert into proj(nome,cliente)  
                values('Python','Dexter Courier')")  
    con.commit()  
    print("Registro criado com sucesso")  
except Exception as e:  
    print("Erro: {}".format(e))  
    con.rollback()  
  
finally:  
    print("Finalizando conexão com o banco de dados")  
    cur.close()  
    con.close()
```

Neste exemplo foi digitado errado o nome da tabela.

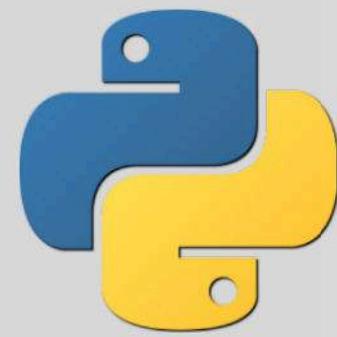
```
#fazer a busca  
cur.execute("select * from TABELA")  
#retornar o primeiro registro  
cur.fetchone()  
#retornar todos os registros  
cur.fetchall()
```

- ✓ **fetchone()**: realiza busca, retorna só o primeiro registro do banco.
- ✓ **fetchall()**: realiza busca retornando todos os registros do banco.

Segue exemplo buscando o primeiro registro de uma tabela e, na sequência todos os registros:

```
#!/usr/bin/python3  
  
import MySQLdb  
  
try:  
    con = MySQLdb.connect(host="127.0.0.1",db="projetos",  
                          user="admin",passwd="4linux")  
    cur = con.cursor()  
    cur.execute("select * from projetos")  
    print("Primeiro registro da tabela: ",cur.fetchone())  
    print("Todos os registros: ",cur.fetchall())  
except Exception as e:  
    print("Erro: {}".format(e))  
  
finally:  
    print("Finalizando conexão com o banco de dados")  
    cur.close()  
    con.close()
```

Note que o `commit` e o `rollback` foram removidos deste exemplo. No caso de querys não são feitas alterações na base de dados, então eles podem ser omitidos.



Python Fundamentals

Python com MongoDB

9.3

Anotações

Objetivos da Aula

- ✓ Conhecer o módulo de conexão com o MongoDB.
- ✓ Trabalhar com Python e MongoDB.

Python com MongoDB

Nesta aula você conhecerá o módulo para conexão e integração do Python com MongoDB, aprenderá a fazer consultas, inserções, alterações e exclusões de documentos e subdocumentos.

Anotações

Para conectar o MongoDB com scripts em Python, é necessário instalar o módulo PyMongo.

```
pip install pymongo
```

- ✓ Poucas coisas irão mudar na sintaxe shell do MongoDB para a sintaxe do Python.

Anotações

Crie um script, contendo instância destinada a objeto MongoClient para fazer a conexão com o banco de dados.

```
from pymongo import MongoClient  
client = MongoClient('127.0.0.1')  
db = client['dexterops']
```

Anotações

Para inserir um novo documento, execute:

```
db.fila.insert({ "_id": 1,  
                 "servico": "Intranet",  
                 "status": 0 })
```

Para deletar todos os documentos de uma collections:

```
db.fila.remove()
```

Inserção de Dados

Podemos inserir os documentos e subdocumentos como demonstrado no script a seguir:

```
from pymongo import MongoClient  
client = MongoClient('127.0.0.1')  
db = client['dexterops']  
  
def inserir_dados():  
    try:  
        db.fila.insert({ "_id": 1, "empresa": "4linux",  
                         "cursos": [ { "nome": "Python Fundamentals",  
                                      "carga horaria": 40 },  
                                      { "nome": "Linux Fundamentals",  
                                      "carga horaria": 40 } ] })  
    except Exception as e:  
        print("Erro: {}".format(e))  
  
inserir_dados()
```

Para realizar busca de dados utilize:

```
db.fila.find()
```

A busca nos retornará um objeto, para exibir informações, execute:

```
for r in db.fila.find():
    print("Serviço:{} ".format(r['servico']))
```

Busca de Dados

Para a busca de dados utilizando subdocumentos, faça com a seguir:

```
from pymongo import MongoClient
client = MongoClient('127.0.0.1')
db = client['dexterops']

def buscar_dados():

    for r in db.fila.find():
        print('Empresa: {}'.format(r['empresa']))
        for c in r['cursos']:
            print('===== ')
            print('Nome: {} \n Carga Horária: {} \n'.format(
                (c['nome'],c['carga horaria'])))

buscar_dados()
```

Para adicionar um subdocumento:

```
db.fila.update({ "_id": 1 }, { $addToSet:  
    { "servidores":  
        { "nome": "dns",  
          "endereco": "192.168.0.40" }  
    } } )
```

Adicionar subdocumentos

Para adicionar os instrutores disponíveis, veja o exemplo:

```
from pymongo import MongoClient  
client = MongoClient('127.0.0.1')  
db = client['dexterops']  
  
def adicionar_sub():  
    db.fila.update({ "_id": 1 }, { "$addToSet":  
        { "instrutores": { 'nome': 'Mariana',  
                          'email': 'mariana.albano@4linux.com.br' } } })  
  
adicionar_sub()
```

Para update em subdocumento:

```
db.fila.update({ "_id":1, "servidores.nome": "dns" },  
                { "$set": { "servidores.$.endereco":  
                           "10.0.0.1" } })
```

Update em subdocumentos

Para update dentro do subdocumento instrutores, podemos fazer desta forma:

```
from pymongo import MongoClient  
client = MongoClient('127.0.0.1')  
db = client['dexterops']  
  
def update_sub():  
    db.fila.update({ "_id":2, "instrutores.nome": "Mariana" },  
                   { "$set": { "instrutores.$.nome": "Mari" } })  
  
update_sub()
```

Remover um subdocumento:

```
db.fila.update({ "_id":1, "servidores.nome": "dns" },
                { "$pull": { "servidores.nome": "dns" } })
```

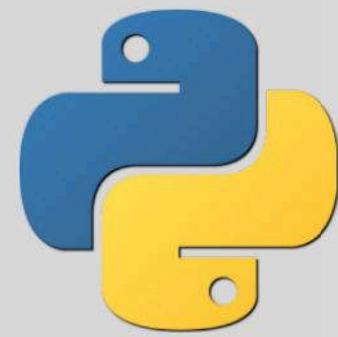
Remover em subdocumentos

Para remover um objeto dentro do subdocumento instrutores, podemos fazer deste modo:

```
from pymongo import MongoClient
client = MongoClient('127.0.0.1')
db = client['dexterops']

def remove_sub():
    db.fila.update({ "_id":2, "instrutores.nome": "Mari" },
                  { "$pull": { "instrutores": { "nome": "Mari" } } })

remove_sub()
```



Python Fundamentals

SQLAlchemy

9.4

Anotações

Objetivos da Aula

- ✓ Compreender o que é sqlalchemy.
- ✓ Por que usar.
- ✓ Conhecer tipos de dados.
- ✓ Executar Operações CRUD.

SQLAlchemy

Nesta aula apresentaremos a definição de sqlalchemy, a razão da sua utilização, seus tipos de dados, como fazer operações crud

Anotações

- ✓ Trata-se de uma biblioteca, criada por Mike Bayer em 2005.
- ✓ Permite interagir com grande variedade de bancos de dados.
- ✓ Possibilita criar modelos de dados e consultas de maneira mais confortável com códigos Python.

Anotações

- ✓ Abstrair seu código de SQL, aproveitar declarações e tipos comuns de instruções SQL que sejam criadas de forma eficiente.
- ✓ Evitar problemas com ataques de SQLInject.
- ✓ SQLAlchemy consiste escrita diferente, constitui biblioteca extensível que trabalha com Oracle, MySQL, Postgres etc.
- ✓ Pode ser estendida facilmente para outros bancos relacionais, trabalha em dois modos diferentes (Core e ORM).
- ✓ SQLAlchemy Core: Uma maneira Pythonica de representar seus dados sem perder o sotaque do SQL.
- ✓ Focado diretamente no banco e seu esquema.

Anotações

Metadados: facilitam a indexação e busca a tabela.

Table: tabela.

Column: coluna da tabela.

Index: Acelera uma busca em um field específico.

Anotações

SQLAlchemy

BigInteger
Boolean
Date
Date Time
Enum
Float
Integer
Interval
Large Binary
Numeric
Unicode
Text
Time

Python

Int
bool
datetime.datetime
datetime.datetime
str
Float or Decimal
int
datetime.timedelta
byte
decimal.Decimal
unicode
str
datetime.time

SOL

BIGINT
BOOLEAN OR SMALLINT
DATE(SQLite:STRING)
DATETIME(SQLite:STRING)
ENUM OR VARCHAR
FLOAT OR REAL
INTEGER
INTERVAL OR DATE from epoch
BLOB OR BYTEA
NUMERIC or DECIMAL
UNICODE or VARCHAR
CLOB or TEXT
DATETIME

Anotações

1 Instale a biblioteca SQLAlchemy:

```
# pip3 install sqlalchemy
```

2 Instale SQLite, execute:

```
# apt-get update  
# apt-get install sqlite3  
# apt-get install libsqlite3-dev  
# apt-get install sqlitebrowser
```

Anotações

4LINUX / Criando Tabela com SQLAlchemy

264

Na estrutura de pasta, crie um arquivo **core.py** com o código:

```
1  from sqlalchemy import (create_engine, MetaData, Column,
2                                Table, Integer,
3                                String, DateTime)
4  from datetime import datetime
5
6
7  engine = create_engine('sqlite:///teste.db', echo=True)
8  metadata = MetaData(bind=engine)
9
10 user_table = Table('usuarios',
11                     metadata,
12                     Column('id', Integer, primary_key=True),
13                     Column('nome', String(40), index=True),
14                     Column('idade', Integer, nullable=False),
15                     Column('senha', String),
16                     Column('criado_em', DateTime, default=datetime.now),
17                     Column('atualizado_em', DateTime,
18                           default=datetime.now, onupdate=datetime.now))
19
20
21 metadata.create_all(engine)
```

Criando tabelas com sqlalchemy

In [1]: do modulo sqlalchemy importamos create_engine, o método de conexão Metadata que define a base de dados, Column, o método que define colunas, Table, método que estabelece as tabelas, Integer, String e DateTime que são tipos de dados como apresentado anteriormente.

In [4]: importamos o datetime, módulo nativo do python para trabalhar com data e hora.

In [7]: definimos a variável de conexão que recebe os parâmetros de conexão com o banco e, o parâmetro echo, logging que visualiza os processos realizados no terminal, este parâmetro pode receber falso para ter uma saída mais limpa no terminal.

In [8]: estabelecemos a variável que cria a base de dados recebendo o parâmetro da variável de conexão.

In [10]: determinamos a variável de criação da tabela que usa os seguintes parâmetros, nome da tabela, variável base de dados, colunas, cada coluna recebe os seguintes parâmetros, nome da coluna, tipo de dado, especificação da coluna.

In [21] : usamos o método da base de dados para fazer commit da criação da tabela, passando o parâmetro de conexão.

Crie um arquivo **core_insert.py** com o código:

```
1 from core import user_table, engine
2
3 con = engine.connect()
4 ins = user_table.insert()
5 # insert pessoa
6 new = ins.values(idade=24, nome='teste', senha='testando')
7 con.execute(new)
8 # insert pesoas
9 con.execute(user_table.insert(),
10             {'nome': 'marcio', 'idade': 20, 'senha': 'semsenha'},
11             {'nome': 'gustavo', 'idade': 18, 'senha': 'abacaxi123'},
12             {'nome': 'guilherme', 'idade': 22, 'senha': 'goiaba123'})
13 ])
```

Inserindo dados na tabela com sqlalchemy

In [1]: do arquivo core, importamos as variáveis, tabela e conexão.

In [3]: definimos uma variável de conexão que recebe o método connect.

In [4]: estabelecemos uma variável tabela, que recebe o método de inserção.

In [6]: determinamos uma variável de inserção que recebe os valores a serem inseridos.

In [7]: utilizamos o método execute da conexão, com parâmetro, variável de inserção para fazer commit dos valores na tabela.

In [9]: usamos o método execute da conexão, com os seguintes parâmetros, variável tabela, com método insert, uma lista de dicionário , na qual cada dicionário é equivalente a um insert na tabela.

Anotações

Crie um arquivo **core_select.py** com o código :

```
1 from sqlalchemy import select
2 from core import user_table
3
4 selecione = select([user_table])
5 selecione_01 = select([user_table]).where(user_table.c.nome == 'teste')
6
7 print([x for x in selecione_01.execute()])
8 print([x for x in selecione.execute()])
```

Select de dados na tabela com sqlalchemy

In [1]: do modulo sqlalchemy importamos select.

In [2]: do arquivo core, importamos a tabela.

In [4]: definimos a variável de select, que recebe uma lista com as tabelas em que serão realizados os selects.

In [5]: mesmo procedimento anterior, porém passando uma condição where, que recebe o parâmetro tabela.coluna.nomecoluna e a condição igual a 'teste'.

In [7]: utilizamos o listcomprehension para percorrer o cursor do objeto e imprimir.

In [8]: usamos o listcomprehension para percorrer o cursor do objeto e imprimir.

Anotações

Crie um arquivo **core_update.py**:

```
1  from sqlalchemy import update
2  from core import user_table, engine
3
4  con = engine.connect()
5
6  atualizar = update(user_table).where(user_table.c.nome == 'teste')
7
8  atualizar = atualizar.values(nome='daniel')
9  result = con.execute(atualizar)
10 print(result.rowcount)
11
12 atualizar = update(user_table).where(user_table.c.nome == 'daniel')
13 atualizar = atualizar.values(idade=(user_table.c.idade - 1))
14 result = con.execute(atualizar)
15 print(result.rowcount)
```

Update de dados na tabela com sqlalchemy

In [1]: do modulo sqlalchemy importamos update.

In [2]: do arquivo core, importamos a tabela e a conexão.

In [4]: definimos a variável conexão com o método connect.

In [6]: determinamos a variável de update com where, semelhante ao select, para definir o usuário que receberá a alteração.

In [8]: estabelecemos os valores da variável update que serão alterados.

In [9]: determinamos o commit na conexão com o update.

In [10]: verificamos quantas linhas sofreram alterações.

In [12]: realizamos o mesmo passo da linha [6].

In [13]: definimos os valores da variável update, passando uma operação no dado.

In [14]: realizamos o mesmo passo da linha [9].

In [15]: realizamos o mesmo passo da linha [15].

Anotações

Crie um arquivo **core_delete.py** com o código :

```
1 from sqlalchemy import delete
2 from core import user_table, engine
3
4 con = engine.connect()
5 d = delete(user_table).where(user_table.c.nome == 'daniel')
6
7 result = con.execute(d)
8 print(result.rowcount)
```

Delete de dados na tabela com sqlalchemy

In [1]: do modulo sqlalchemy importamos delete.

In [2]: do arquivo core, importamos a tabela e a conexão.

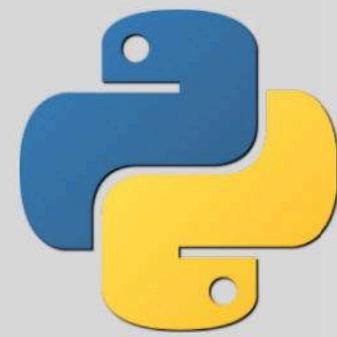
In [4]: definimos a variável conexão com o método connect.

In [5]: determinamos a variável de delete com where, semelhante ao select, para definir o usuário que será deletado.

In [7]: estabelecemos o commit na conexão com o delete.

In [8]: verificamos quantas linhas sofreram alterações.

Anotações



Python Fundamentals

Introdução a Teste Unitário

10.1

Anotações

Objetivos da Aula

- ✓ Entender o que é teste.
- ✓ Entender a razão para testar.
- ✓ Utilizar o assert e fazer testes simples.
- ✓ Utilizar o modulo unittest.

Introdução a teste unitário

Aprenderemos o fundamental sobre teste de software, utilizaremos assert e o módulo unittest, para criar nossos testes.

Anotações

Teste é o processo de validação da conquista dos objetivos esperados por parte do software .

Desenvolvida uma solução para determinado problema, o teste é mandatório como meio de confirmar que o software pode ser colocado em produção, com a garantia de que tudo funcionará perfeitamente.

Anotações

- ✓ Garantir que as funcionalidades foram entregues.
- ✓ Assegurar que o nosso software continua funcionando a cada atualização.
- ✓ Manter proatividade na correção de falhas.
- ✓ Sustentar confiança ao alterar nossa software.

Anotações

assert <valor> condição <valor>, <expressão>

Caso a condição seja verdadeira o teste é aprovado e nada acontece. Se a condição for falsa, o teste é reprovado e executa o Traceback.

```
>>> a = 2
>>> b = 2
>>> assert a == b, 'a é diferente de b'
>>> assert a != b, 'a é igual a b'
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    assert a != b, 'a é igual a b'
AssertionError: a é igual a b
```

Sintaxe da função assert

Definimos duas variáveis ao utilizar o assert. Este, espera da sintaxe, valor, condição, valor,, semelhante a blocos de condição if. Se o resultado da condição for True, o teste foi aprovado e nada acontece, caso contrário, o teste falha, executa o parâmetro de especificação do erro junto com AssertionError.

Anotações

```
1 def somar(x, y):
2     return x + y
3
4 def subtrair(x, y):
5     return x - y
6
7 def dividir(x, y):
8     return x / y
9
10 def multiplicar(x, y):
11     return x * y
12
13 # Testar função somar
14 assert somar(2, 2) == 4, "Obtido:{}, Esperado:4".format(somar(2, 2))
15 assert somar(3, 3) == 6, "Obtido:{}, Esperado:6".format(somar(3, 3))
16 # Testar função subtrair
17 assert subtrair(2, 2) == 0, "Obtido:{}, Esperado:0 ".format(subtrair(2, 2))
18 assert subtrair(5, 3) == 2, "Obtido:{}, Esperado:2".format(subtrair(5, 3))
19 # Testar função dividir
20 assert dividir(4, 2) == 2, "obtido:{}, Esperado:2 ".format(dividir(4, 2))
21 assert dividir(9, 3) == 3, "obtido:{}, Esperado:3".format(dividir(9, 3))
22 # Testar função multiplicar
23 assert multiplicar(10, 2) == 20, "Obtido:{}, Esperado:20 ".format(multiplicar(10, 2))
24 assert multiplicar(1, 3) == 3, "Obtido:{}, Esperado:3".format(multiplicar(1, 3))
```

Testando funções e resultados esperados

Definimos algumas funções: soma, subtrai, divide e multiplica, depois estabelecemos os testes unitários, verificando se o retorno da função é igual ao valor esperado, caso falhe, será exibida uma mensagem personalizada com o valor obtido e o valor esperado.

Anotações

O ('teste unitário') consiste um módulo nativo do python. similar aos principais frameworks de teste encontrados. Suporta automação de testes, compartilhamento de código de configuração, desligamento e agregação de testes em coleções além de independência dos testes da estrutura de relatório.

Para mais informações acesse a documentação oficial:
<https://docs.python.org/3/library/unittest.html>

Anotações

```
1  from unittest import TestCase, main
2
3  def soma(x, y):
4      return x + y
5
6  def sub(x, y):
7      return x - y
8
9  class Testes(TestCase):
10     def test_soma(self):
11         self.assertEqual(soma(5, 5), 10) # Equivalente a ==
12         self.assertLess(soma(5,5), 11) # Equivalente a <
13     def test_sub(self):
14         self.assertEqual(sub(5,5), 0)
15         self.assertLessEqual(sub(15, 5), 10) # Equivalente a <=
16
17 if __name__ == '__main__':
18     main()
```

Utilizando o unittest

In [1]: do módulo unittest importamos TestCase e main.

In [3]: definimos duas funções para realizar os testes.

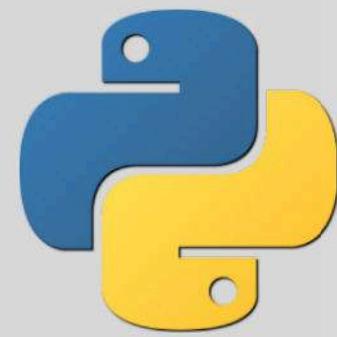
In [9]: instanciamos a classe, Testes, que herda do TestCase. Determinamos alguns métodos, como os atributos específicos que recebem dois valores, no parâmetro, para realizar a condição do teste. No próximo slide, apresentamos uma tabela com a estrutura de atributos existentes neste objeto para automação de testes.

In [17]: criamos o bloco de condição para que o teste seja executado apenas se for iniciado por ele mesmo, evitando que outros scripts o importem e executem.

Anotações

Método	Verifica que	Novo em
<code>assertEqual(a, b)</code>	<code>a == b</code>	
<code>assertNotEqual(a, b)</code>	<code>a != b</code>	
<code>assertTrue(x)</code>	<code>bool(x) is True</code>	
<code>assertFalse(x)</code>	<code>bool(x) is False</code>	
<code>assertIs(a, b)</code>	<code>a is b</code>	3,1
<code>assert IsNot(a, b)</code>	<code>a is not b</code>	3,1
<code>assertIsNone(x)</code>	<code>x is None</code>	3,1
<code>assert IsNotNone(x)</code>	<code>x is not None</code>	3,1
<code>assertIn(a, b)</code>	<code>a in b</code>	3,1
<code>assertNotIn(a, b)</code>	<code>a not in b</code>	3,1
<code>assertIsInstance(a, b)</code>	<code>isinstance(a, b)</code>	3,2
<code>assertNotIsInstance(a, b)</code>	<code>not isinstance(a, b)</code>	3,2

Anotações



Python Fundamentals

TDD

10.2 Test Driven Development

Anotações

Objetivos da Aula

- ✓ Compreender o que é TDD.
- ✓ Conhecer suas vantagens e desvantagens.
- ✓ Aprender como utilizar esta cultura.

Desenvolvimento orientado a teste

Nesta aula apresentaremos o conceito sobre TDD, suas vantagens e desvantagens, explicaremos como implementá-lo.

Anotações

TDD

É o desenvolvimento de software orientado a testes, em inglês, Test Driven Development. Mas mais do que simplesmente testar seu código, TDD é uma filosofia, uma cultura.

Desenvolvimento orientado a testes, criamos inicialmente os testes antes de implementar a lógica do software.

Anotações

- ✓ Dificuldade em começar.
- ✓ Curva de aprendizado.
- ✓ Tempo de desenvolvimento.
- ✓ Culturas adotadas.

Por que muitos não praticam?

Dificuldade em começar — Apesar de uma extensa e clara documentação, iniciar o desenvolvimento orientado a testes pode ser um trabalho árduo para muitos. Devido o simples fato de que geralmente, muitos iniciantes tentam praticá-lo em código já existente. Este, definitivamente não é o caminho. A principal característica do desenvolvimento orientado a testes, reside na orientação a testes. Em outras palavras, o código que realizará sua lógica, deve ser criado somente após a criação do teste. Torna-se algo de difícil aceitação, pois ainda não se produziu nada e já se planeja testar.

Curva de aprendizado — Complementando o item anterior, mais um motivo que faz programadores desistirem do desenvolvimento orientado a testes. Como qualquer nova tecnologia, para a prática de TDD, leva-se tempo em dependência, disponibilidade e principalmente, da vontade do programador.

Tempo — Engana-se quem pensa que produzirá mais código pelo simples fato de utilizar TDD. A cultura na verdade chega a desacelerar a produção de código. Nos referimos em produção de código, à quantidade de linhas escritas. Mesmo neste aspecto, há vantagens, que serão descritas mais à frente.

Cultura — Muito se fala em TDD no Brasil. Porém, ao questionarmos programadores de diversas empresas, muitos apresentam os motivos já citados, para não utilizá-lo. Há muitas empresas e programadores que levam a prática a sério, a evangelizam justamente por conhecerem as vantagens oferecidas pelo TDD.

- ✓ Qualidade do código.
- ✓ Raciocínio.
- ✓ Segurança.
- ✓ Trabalho em equipe.
- ✓ Documentação.

Anotações

Vantagens

Qualidade do código — Um dos principais ensinamentos, senão o principal, do TDD: se algo não é suscetível a teste, foi desenvolvido de forma errada. Parece um pouco drástico, mas, não é. Em pouco tempo utilizando testes, o programador percebe mudanças relevantes em sua forma de programar. Em suma, o uso de TDD ajuda o programador a elaborar um código com melhor qualidade, criando objetos concisos e com menos dependências.

Raciocínio — Para tornar o código mais conciso, tenha menos acoplamentos e dependências. O programador deve forçar seu raciocínio a níveis elevados. É trabalhoso criar algo que realmente tenha um bom design. Utilizando TDD o programador praticamente obriga-se a olhar seu código de outra forma, normalmente jamais vista antes. Aí está a parte legal da coisa toda.

Segurança — Importantíssimo, para qualquer software nos dias de hoje. Mas, não se engane, não se trata de segurança da informação, todavia, de segurança ao desenvolver. Pense em uma situação na qual o programador trabalha um código em construção há cerca de um ano. Como normalmente vivemos em um mundo com inúmeros softwares desenvolvidos ao longo de cada ano, torna-se muito difícil lembrar de tudo a respeito de um caso específico que merece atenção em determinado momento. Normalmente deve-se realizar um trabalho bastante cauteloso para nova implementação em um software já em produção. Toda e qualquer alteração deve ser minuciosamente testada, garantindo que não afetará os demais módulos do software. Esta implementação manual, é realmente complexa, pois até então, não se sabe, ou recorda-se ao certo quem afeta o que no sistema. Com a prática de TDD, cada pequeno passo do software, está devidamente testado. Ou seja, neste cenário o programador pode realizar qualquer alteração sem medo e sem culpa.

Como cada pequeno passo tomado pelo sistema está testado, caso qualquer módulo ou funcionalidade sofra alteração, em poucos segundos descobre-se se houveram quebras, ainda melhor, onde foram essas quebras. Assim, a correção das quebras torna-se uma tarefa simples sem frustrar o cliente e o usuário.

Trabalho em equipe — Ao prover mais segurança, o trabalho em equipe torna-se muito mais proveitoso, eliminando discussões e dúvidas desnecessárias. Ao entrar no desenvolvimento de um projeto, o novo desenvolvedor terá apenas o trabalho de compreender qual task deve ser realizada e, ler os testes das features já desenvolvidas. Ao rodar os testes pela primeira vez, descobrirá que está no caminho para dispor entregáveis mais rapidamente e com segurança. Existem empresas em que um novo programador tem entregáveis logo no primeiro dia de trabalho. Sem testes, normalmente haveria um período de adaptação, para prévio entendimento do que há no sistema no momento de seu ingresso ao time de desenvolvimento.

Documentação — Ao criar testes descritivos, estes servem como uma excelente documentação para o software. Quando qualquer programador rodar os testes, basta habilitar o modo verbose, uma “história” será contada, eliminando o árduo trabalho de documentar um software, que em meios tradicionais tende a defasagem. A documentação tradicional raramente segue o mesmo ritmo do desenvolvimento. Com os testes unitários, a “documentação” é gerada antes mesmo da nova feature ser implementada e, permanece fiel a qualquer alteração.

```
1  from unittest import TestCase, main
2
3  def validar_par(num: int) -> bool:
4      ...
5          Função para validar um número par.
6          Args:
7              num - recebe um número do tipo inteiro
8          Retorno: Booleano
9          ...
10         # Aqui será implementado a lógica
11         pass
12
13 class Testes(TestCase):
14     def test_par(self):
15         self.assertEqual(validar_par(4), True)
16         self.assertEqual(validar_par(1000), True)
17     def test_impar(self):
18         self.assertEqual(validar_par(5), False)
19         self.assertEqual(validar_par(1001), False)
20     def test_string(self):
21         self.assertEqual(validar_par('102'), True)
22         self.assertEqual(validar_par('1059'), False)
23         self.assertEqual(validar_par('string'), None)
24
25 if __name__ == "__main__":
26     main()
```

Criando teste

In [1]: do módulo unittest importamos TestCase e main, para criar nosso teste.

In [3]: definimos uma função com o seguinte problema, validar um número par, com a seguinte descrição, receber um número inteiro e retornar um booleano.

In [13]: determinamos a nossa classe Testes, herdando de TestCase, com os seguintes métodos , testar par: caso o número recebido seja par, retornar verdadeiro, testar ímpar: caso o número seja ímpar, retornar falso, testar string: caso a string seja numérica, realizar a conversão para inteiro e retornar verdadeiro para par e, falso para ímpar , caso contrário retornar None.

Anotações

```
3 def validar_par(num: int) -> bool:
4     ...
5     Função para validar um número par.
6     Args:
7         num - recebe um número do tipo inteiro
8     Retorno: Booleano
9     ...
10    if isinstance(num, int):
11        return True if num % 2 == 0 else False
12    elif isinstance(num, str):
13        if num.isnumeric():
14            return True if int(num) % 2 == 0 else False
15    else:
16        return None
```

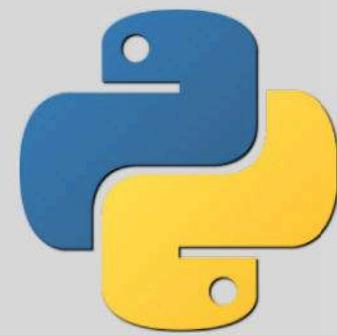
Solucionando o teste

In [10]: criamos um bloco de condição para verificar se a instância do parâmetro é inteira, caso seja, retornar verdadeiro quando o módulo de divisão por 2 for igual a 0. Falso para qualquer outro valor. Esta condição verifica se um número é par.

In [12]: criamos um bloco de condição encadeada, verificando se a instância do parâmetro é string caso seja verdadeira. Criando outro bloco de condição, confirmando se é numérico, caso verdadeiro realizar a conversão pra inteiro e fazer a verificação se o número é par.

In [15]: se o parâmetro não passar nas condições acima, a função retorna 'None'

Anotações



Python Fundamentals

10.3

BDD

Behavior Driven Development

Anotações

Objetivos da Aula

- ✓ Aprender o que é BDD.
- ✓ Conhecer o framework behave.
- ✓ Conhecer sua estrutura de pasta.
- ✓ Criar steps e funcionalidade.

Behavior Driven Development

Nesta aula você aprenderá sobre BDD, conhecerá o framework Behave, compreendendo sua estrutura de pastas, criará steps e funcionalidades, acompanhando sua saída de teste.

Anotações

Behavior Driven Development (BDD), em tradução Desenvolvimento Guiado por Comportamento, é uma técnica de desenvolvimento Ágil. Encoraja a colaboração entre desenvolvedores, setores de qualidade e pessoas não técnicas ou de negócios, num projeto de software.

Relaciona-se ao conceito de verificação e validação. Originalmente concebida em 2003, por Dan North como uma resposta à Test Driven Development (Desenvolvimento Guiado por Testes), vem se expandido bastante nos últimos anos.



Diferenças BDD e TDD

BDD — Trabalha para definir como uma demanda chega ao desenvolvedor, integra diferentes áreas da empresa, pensa a partir do ponto de vista de comportamento esperado, pelo usuário, de uma funcionalidade. Por consequência, influencia como os testes são planejados e escritos.

TDD — Busca garantir a qualidade do código, sempre pensando em 100% de cobertura de testes, melhora o que acabou de ser realizado, nunca escreve uma linha de código sem antes pensar em como garantir que irá funcionar.

Podemos concluir que BDD não se opõe ao TDD, ambos os métodos podem ser aplicados em conjunto ou, apenas um deles. Um e outro, buscam melhorar o desenvolvimento de software, constituindo parte de cultura de excelência.

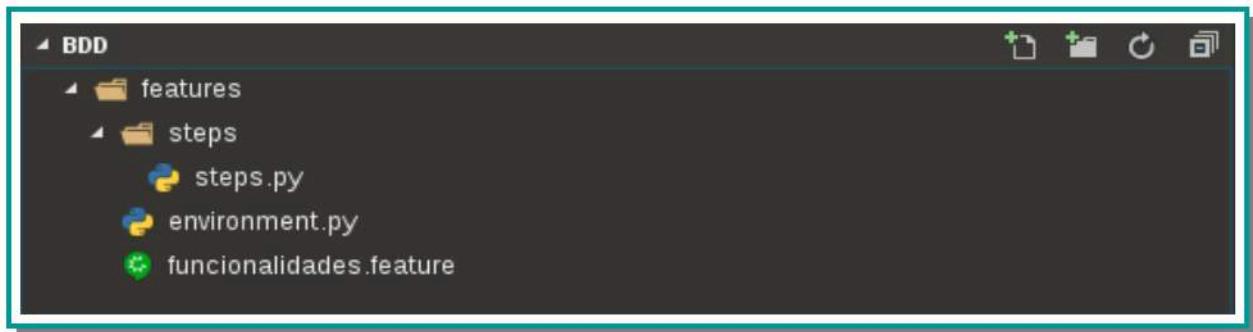
Anotações

Trata-se de uma biblioteca Python que permite a escrita de testes de comportamento em linguagem humana e, a execução dos cenários através de asserts em "linguagem de programação".

1

Instale o behave
\$ pip3 install behave

Anotações



Estrutura de Pasta

Features # Pasta de estrutura do framework.

- steps # Pasta para arquivos de definição da feature.
- step.py # arquivo python de definição para feature.
- environment.py # arquivo python para setup global.
- name.feature # feature file.

Anotações

Arquivo funcionalidade.feature

```
1 # language: pt
2
3 Funcionalidade: Soma
4   Cenario: adicao basica
5     Quando somar "2" com "2"
6     Entao o resultado deve ser "4"
```

Feature

In [1]: definimos a linguagem.

In [3]: determinamos o bloco funcionalidade e atribuímos nome a ele.

In [4]: estabelecemos o cenário com descrição do evento.

In [5] e [6]: definimos o bloco de condição.

Obs. : toda a feature escrita em português.

Anotações

Arquivo steps.py

```
1 from behave import step
2
3 def soma(x:int, y:int) -> int:
4     return x + y
5
6 @step('somar "{num1}" com "{num2}"')
7 def test_soma(context, num1, num2):
8     context.r_soma = soma(int(num1), int(num2))
9
10 @step('o resultado deve ser "{esperado}"')
11 def test_soma_result(context, esperado):
12     assert context.r_soma == int(esperado), "Descrição do erro"
```

Behave

In [1]: do módulo behave importamos step.

In [3]: definimos a função somar, retorno a soma de dois números inteiros.

In [6]: determinamos o decorator step de 2 números, recebe a função test_soma que recebe 3

parâmetros, context é o atributo da classe de objeto do framework e num1 e num2, números inteiros.

In [10]: estabelecemos o decorator step, recebe a função de resultado que obtém dois parâmetros, contexto, este, recebeu os valores do teste acima, o esperado é um valor a ser comparado a context para ter a estrutura de condição, se o teste passou ou não.

Anotações

1

Execute a feature

 behave funcionalidades.feature

```
jonsnow@stark:~$ cd BDD/
jonsnow@stark:~/BDD$ cd features/
jonsnow@stark:~/BDD/features$ behave funcionalidades.feature
Funcionalidade: Soma # funcionalidades.feature:3

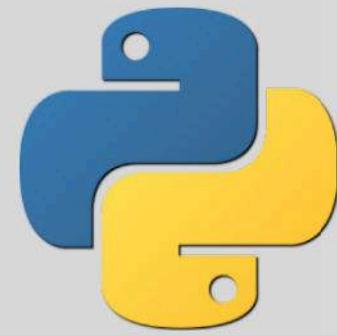
  Scenario: adicao basica          # funcionalidades.feature:4
    Quando somar "2" com "2"      # steps/steps.py:6 0.000s
    Então o resultado deve ser "4" # steps/steps.py:10 0.000s

1 feature passed, 0 failed, 0 skipped
1 scenario passed, 0 failed, 0 skipped
2 steps passed, 0 failed, 0 skipped, 0 undefined
Took 0m0.000s
```

Executando

Ao executar o arquivo da feature, teremos um resultado semelhante, colorindo de vermelho os steps reprovados na condição, os aprovados serão coloridos em verde. Também especifica quantos testes foram realizados, quantos aprovados e o tempo consumido para fazer as verificações.

Anotações



Python Fundamentals

Compreendendo o Projeto

11.1

Anotações

Objetivos da Aula

- ✓ Entender o projeto.
- ✓ Criar a estrutura do projeto.

Entendendo o Projeto

Nesta aula, você compreenderá o projeto, montando a sua estrutura.

Anotações

Criaremos um chat para comunicação interna da empresa Dexter Courier.

Devemos atender os seguintes requisitos:

- ✓ Mensagens serão armazenadas em banco de dados.
- ✓ O padrão de armazenamento será: nome, data, mensagem.
- ✓ Não preocupar-se com autenticação.



Anotações

As mensagens serão armazenadas em mongodb. Banco de dados ideal para o projeto, uma vez que não existe a necessidade de criar relacionamento entre as tabelas, sendo sua leitura/escrita mais rápida.

O armazenamento no banco será feito da seguinte forma:

```
{ 'nome' : 'Cliente', 'data' : '28/07/2017 - 15:00',  
  'mensagem' : 'olá' }
```



mongoDB

Anotações

A estrutura do projeto deverá conter os seguintes arquivos:

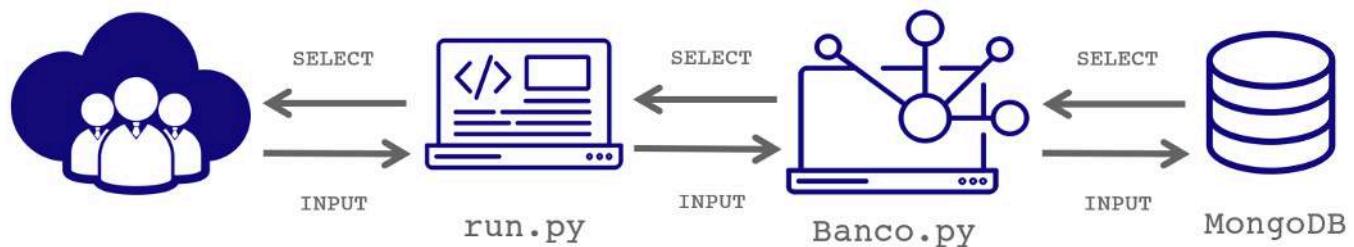


/app.py: será a interface também responsável pela execução do chat.



/modules/banco.py: responsável por consultar e gravar informações no banco.

Anotações



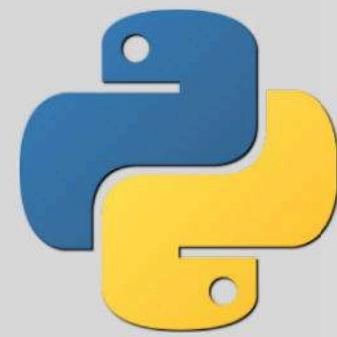
Fluxo da aplicação

O acesso do usuário na aplicação, será através do arquivo “run.py”, no qual, poderá escrever novas mensagens (input).

Este input será recebido pela aplicação e enviado ao arquivo “banco.py” que enviará a gravação da mensagem para o banco.

Após o término deste fluxo, a aplicação automaticamente fará com que as novas mensagens apareçam no chat.

Anotações



Python Fundamentals

Construindo o Projeto

11.2

Anotações

Objetivos da Aula

- ✓ Criar o arquivo de conexão com o banco de dados.
- ✓ Criar script para interação de usuários.

Construindo o Projeto

Nesta aula criaremos os arquivos responsáveis pela conexão com o MongoDB e a interação com o usuário.

Anotações

Arquivo: banco.py

```
1 from pymongo import MongoClient, DESCENDING
2 import time
3 try:
4     client = MongoClient()
5     db = client['chat']
6 except Exception as e:
7     print('ERRO: {}'.format(e))
8     exit()
```

Importando módulos e criando conexão com banco de dados

In [1]: do modulo pymongo, importamos mongoclient e descending.

In [2]: importamos o módulo time.

In [3]: criando a conexão com o banco com um tratamento de exceção, caso a conexão falhe, será encerrada com script.

Anotações

Arquivo: banco.py

```
8 def cadastrar(name, mensagem):
9     date = {
10         'nome': name,
11         'mensagem': mensagem,
12         'hora': time.strftime('%d-%m-%Y %H:%M:%S')
13     }
14     db.chat.insert(date)
```

Função para cadastrar mensagens

In [8]: criamos a função de cadastro, esta, recebe dois parâmetros, nome e mensagem, armazenados em um dicionário, junto a hora de execução da função usando o modulo time. O dicionário é inserido no mongodb.

Anotações

Arquivo: banco.py

```
17 def select():
18     ultimo = [x for x in db.chat.find().sort("_id", DESCENDING)]
19     while True:
20         date = [x for x in db.chat.find().sort("_id", DESCENDING)]
21         if date != ultimo:
22             ultimo = date
23             print('[{} {} : {} \n'.format(
24                 date[0]['hora'], date[0]['nome'], date[0]['mensagem']))
25             time.sleep(2)
```

Função para mostrar mensagens

In [17]: criamos a função select que busca o último registro do banco de dados, o armazena em uma variável, executa um loop infinito que refaz a busca no banco do último registro e o compara com a variável último, caso seja uma nova mensagem, retorna o valor de último, mostrando a mensagem, encerra com um sleep de 2 segundos.

Anotações

Arquivo: app.py

```
1 import modulos.banco as banco
2 import threading
3
4 if __name__ == "__main__":
5     user = input('Nickname: ')
6     try:
7         f = threading.Thread(target=banco.select)
8         f.start()
9     except Exception as e:
10        print('Falha ao criar thered: {}'.format(e))
11    # Enquanto thred rodar em segundo plano
12    while f.isAlive():
13        mens = input()
14        banco.cadastrar(user, mens)
```

Script

In [1]: importamos o arquivo com funções do banco de dados.

In [2]: importamos threading para executar multiprocessamento.

In [4]: verificamos o `__name__` do arquivo garantindo que o script será executado somente por ele mesmo.

In [5]: atribuímos uma variável que recebe um usuário.

In [6]: criamos um tratamento de exceção que instancia uma thread para a função select, rodará em segundo plano em um loop infinito buscando últimas mensagens para mostrar na tela.

In [12]: criamos um segundo loop infinito que, enquanto a thread estiver rodando, recebe uma mensagem e cadastra no banco dados passando usuário e mensagem.

Anotações



Python Fundamentals

Introdução ao SSH e Paramiko

12.1

Anotações

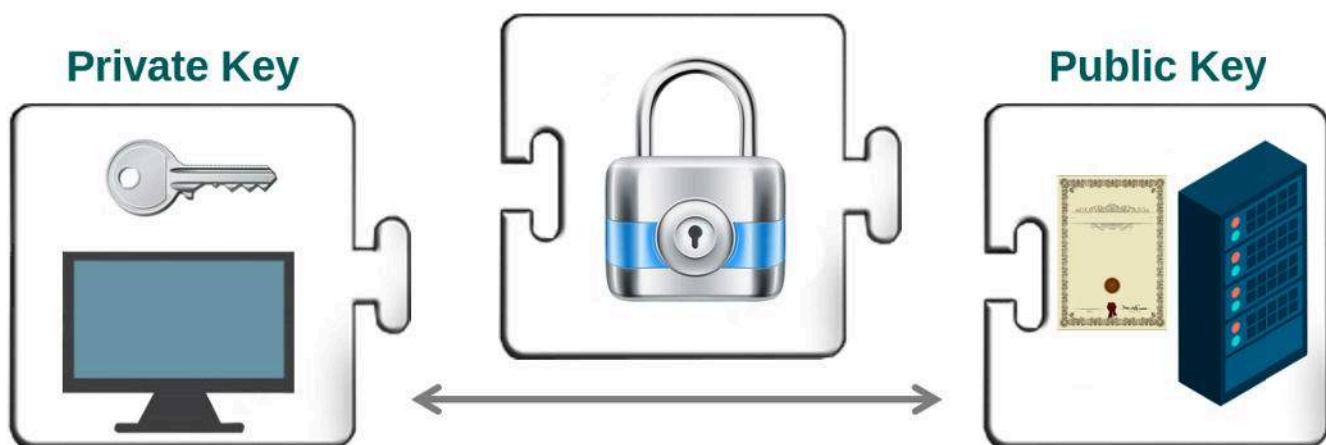
Objetivos da Aula

- ✓ Aprender o que é SSH.
- ✓ Compreender seu funcionamento.
- ✓ Fazer conexão SSH.
- ✓ Utilizar o módulo para automatizar.

Introdução ao SSH e Paramiko

Nesta aula você conhecerá SSH, aprendendo a fazer conexões e utilizará módulo Paramiko para automatizar ações.

Anotações



O que é?

SSH é o acrônimo de Secure Shell, em tradução literal, algo como “cápsula segura”. O protocolo SSH permite a um computador conectar-se em um servidor remoto pela internet de maneira segura. A conexão exige autenticação dos dois lados, servidor e computador e, é criptografada. Deste modo, se alguém interceptar o pacote de dados que está sendo transmitido, não será possível visualizar o conteúdo da mensagem, pois apenas os computadores que estão conectados entre si possuem a chave para descriptografar.

O protocolo de comunicação SSH está presente em todos os servidores de hospedagem profissionais, sendo amplamente utilizado por empresas e profissionais da área. Veremos a seguir, as vantagens que este protocolo oferece, em termos de segurança e, também quanto à agilidade na realização de diversas tarefas.

A maior razão para utilização do SSH é a segurança. Como é criptografada, somente o computador e o servidor tem acesso a chave, dificilmente esta chave será decifrada por terceiros.

Para programadores, esse tipo de conexão é altamente utilizada em sistemas de versionamento como: o GitLab, GitHub, BitBucket e afins. Em alguns casos, dependendo das integrações configuradas, estes sistemas fazem outras conexões automáticas para realizar o deploy do código rapidamente.



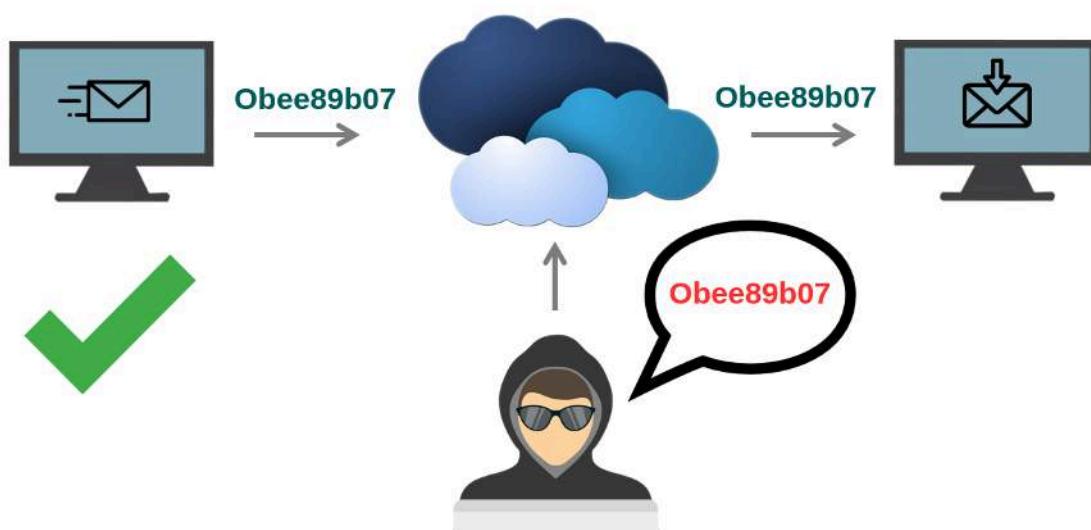
Características

Há uma variedade de ferramentas, que podem ser usadas, para romper ou interceptar dados de uma comunicação, sendo seu objetivo conseguir acesso a um sistema, por exemplo: usar um sniffer para capturar dados que estão trafegando na rede.

Com o SSH essa ameaça é quase nula, pois o cliente e o servidor SSH, usam assinaturas digitais para verificar a sua identidade. Além disso, como já afirmamos, toda a comunicação entre eles é criptografada.

Quando abordamos o assunto Cloud, é praticamente impossível não falar sobre SSH. Com o avanço do mercado de Cloud, o SSH tornou-se uma ferramenta vital para o SysAdmin, quando o assunto é administrar servidores remotos. Exatamente por isso, é extremamente importante configurar esse serviço de forma correta, visando sempre a segurança do acesso ao sistema.

Ao contratar serviços na nuvem, geralmente o cliente recebe, do fornecedor do serviço, uma chave com a qual será feita a conexão no servidor. Esta chave é acionada através do comando "ssh" com o parâmetro "-i", seguido do caminho para a chave, que será responsável pela autenticação do usuário.



Características

- 1º Após a primeira conexão, a identidade do Servidor é armazenada (known_hosts), garantindo que você sempre accesse o servidor correto. Caso a identidade seja alterada, será emitido um alerta.
- 2º O cliente transmite as informações de autenticação usando criptografia forte de 256 bits
- 3º Todos os dados usam uma criptografia de 256 bits, tornando, praticamente, impossível decifrar os dados.
- 4º Como SSH criptografa tudo, também pode ser utilizado para tunelamento em outros protocolos inseguros.

O SSH é considerado seguro por utilizar algumas funcionalidades de transmissão e autenticação de usuários, provavelmente, a principal dessas funcionalidades, é o sistema de chaves criptográficas. Uma chave criptográfica pública, autentica um computador remoto. Esse sistema de chaves, também pode ser utilizado para autenticar usuários no sistema.

```
jonsnow@stark:~$ ssh bran@127.0.0.1 -p 2222
The authenticity of host '[127.0.0.1]:2222 ([127.0.0.1]:2222)' can't be established.
ECDSA key fingerprint is SHA256:PLAzfNicxlq5rIYqlqGVnqGf1En1Iy9MQxIYPdkcKFw.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '[127.0.0.1]:2222' (ECDSA) to the list of known hosts.
bran@127.0.0.1's password:
Welcome to KDE neon User Edition 5.13 (GNU/Linux 4.15.0-30-generic x86_64)

* Documentation:  https://help.ubuntu.com
* Management:    https://landscape.canonical.com
* Support:       https://ubuntu.com/advantage

190 pacotes podem ser atualizados.
9 atualizações são atualizações de segurança.

The programs included with the KDE neon system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*copyright.

KDE neon comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

bran@stark-vbox:~$
```

Primeiro acesso

Para realizar a primeira conexão existem dois métodos:

Usuário e senha

Para realizar uma conexão no Linux, utilize SSH diretamente no terminal:

Quando aplicar o comando no terminal, aparecerá uma mensagem, que ocorre no primeiro acesso, indicando que não foi possível estabelecer a autenticidade do endereço em questão. Surgirá a pergunta, se gostaria de continuar. Avance, escreva "yes" e depois confirme com a tecla enter. Em seguida, senha do usuário será solicitada, digite normalmente. No terminal nada vai aparecer, mesmo assim, utilize novamente a tecla enter para confirmar sua senha.

Se tudo estiver certo, a partir de agora você observará que os dados do terminal foram alterados para o usuário e nome do servidor, agora você poderá fazer o que precisar na máquina do servidor.

Par de chaves públicas e privada

Esta é uma forma ainda mais segura que a anterior. As chaves, são trechos de texto que contém uma combinação de caracteres em par, utilizada para descriptografar os dados e, somente a combinação correta consegue fazer isso.

Quando utilizamos este tipo de acesso, o primeiro passo é gerar essas chaves, para isso, devemos utilizar o comando a seguir em nosso terminal:

Constitui um módulo do Python utilizado para automatizar conexões ssh. Desta forma criamos scrips que automatizam tarefas como logar em um servidor, efetuar tarefas e depois sair do servidor. Tal aspecto relaciona-se a administração de infraestrutura ágil.

1

Instalando o paramiko

pip3 install paramiko

Anotações

```
1 import paramiko
2
3 try:
4     client = paramiko.client.SSHClient()
5     client.load_system_host_keys()
6     client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
7     client.connect('127.0.0.1',
8                     username='bran',
9                     password='4linux',
10                    port='2222')
11 except Exception as e:
12     print('Erro conexão: {}'.format(e))
13     exit()
```

Conecando

In [1]: Importamos o modulo paramiko

In [3]: Criamos um tratamento de exceção para a conexão ssh, passando configurações de cliente e método connect passando parâmetros host, username, password, e port.

In[11]: Caso a conexão não seja realizada com sucesso exibira uma mensagem especificando o erro.

Anotações

```
16  stdin, stdout, stderr = client.exec_command('ls -la')
17  if stdout.channel.recv_exit_status() == 0:
18      print(stdout.read().decode('utf-8'))
19  else:
20      print(stderr.read().decode('utf-8'))
```

Executando comandos

Para executar comandos, necessitamos utilizar o método `exec_command`, com os seguintes parâmetros:

command: (string) o comando para executar.

bufsize: (integer) interpretado da mesma maneira que a função, `file`, embutida no Python.

timeout: (integer) define o tempo limite do canal do comando.

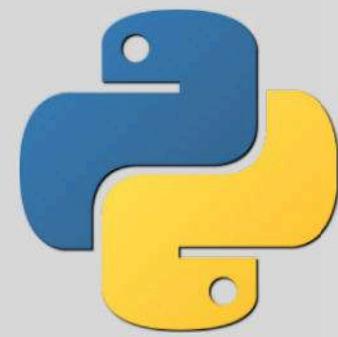
environment: (dict) um dicionário de variáveis de ambiente do shell, para ser mesclado no ambiente padrão no qual o comando remoto é executado.

Em nosso caso, passamos um comando simples, tornando possível retornar uma lista dos arquivos que temos no servidor que está sendo acessado. O retorno desse método, devolve três instâncias da classe `ChannelFile`. A primeira para entrada de dados, a segunda para receber o texto do retorno e terceira recebe os erros.

In [16]: O método `recv_exit_status` dentro da variável `stdout`, é utilizado para verificar se recebemos ou não, um código de erro. Se o valor devolvido pela função, for igual a 0, significa que tudo ocorreu como esperado.

In [18]: O método `read`, lê o texto retornado, em caso de sucesso ao executar o comando, mas devolve em valor binário. Por isso, utilizamos a função do Python: `decode`, para transformar esse dado em texto comum.

In [20]: fazemos a mesma coisa novamente, porém com o valor, caso o comando tenha retornado erro.



Python Fundamentals

Jupyter

12.2

Anotações

Objetivos da Aula

- ✓ Compreender o que é Jupyter notebooks.
- ✓ Conhecer suas funcionalidades.
- ✓ Como instalar e executar.
- ✓ Configurar primeiro acesso.
- ✓ Como utilizar.

Jupyter Notebooks

Nesta aula apresentaremos o jupyter notebooks, como realizar sua instalação e configuração e utilização.

Anotações

Trata-se de uma aplicação web, que facilita o entendimento e visualização de dados e resultados de análises, juntamente com o código.



Jupyter Notebooks

Facilita a experimentação, colaboração e publicação online. Anteriormente conhecido como IPython notebooks, atualmente permite o uso de outras linguagens, Python porém é o default. Jupyter Notebook constitui uma das ferramentas ideais, como um aplicativo da web, para ajudá-lo a obter as habilidades de ciência de dados necessárias, permitindo criar e compartilhar documentos que contenham código ativo, equações, visualizações e texto.

Anotações

- ✓ Produzidos pelo aplicativo, "Notebook" ou "documentos de notebook", denotam registros contendo elementos de código, texto, figuras, links, equações, etc.
- ✓ Devido à combinação de código e elementos textuais, esses documentos consistem meio ideal para reunir uma descrição de análise e seus resultados.
- ✓ Também podem executar a análise de dados em tempo real.

Anotações

- ✓ Ao executar o documento, o kernel associado, inicia-se automaticamente executa o cálculo, produzindo os resultados.
- ✓ Dependendo do tipo de computação, consumirá significativamente CPU e RAM.
- ✓ Observe que a RAM não é liberada até que o kernel seja desligado.

Kernel

O kernel de notebook é um "mecanismo computacional" que executa o código contido em um documento de Notebook, sendo acionando automaticamente na iniciação do documento, pode ser executado, célula por célula ou com o menu Cell -> Run All

Anotações

Por padrão o jupyter notebook rodará na porta 8888:

`http://127.0.0.1:8888`

1 Instale o jupyter, execute o comando:

`# pip3 install jupyter`

2 Execute o jupyter:

`# jupyter notebook --allow-root`

Anotações

```
[root@stark:/home/jonsnow# jupyter notebook --allow-root
[I 16:56:12.109 NotebookApp] Serving notebooks from local directory: /home/jonsnow
[I 16:56:12.109 NotebookApp] The Jupyter Notebook is running at:
[I 16:56:12.109 NotebookApp] http://localhost:8888/?token=27c6f6e80eca591db472ab21050b3dd0d1e6d7
cd7b4d7a00
[I 16:56:12.109 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice
to skip confirmation).
[C 16:56:12.110 NotebookApp]

Copy/paste this URL into your browser when you connect for the first time,
to login with a token:
    http://localhost:8888/?token=27c6f6e80eca591db472ab21050b3dd0d1e6d7cd7b4d7a00
```



Execução do jupyter notebook

Ao iniciar a aplicação jupyter notebook dentro de um diretório, será gerado um log no terminal especificando a url de acesso e um token que será usado como chave de acesso.
Ao acessar o localhost na porta especificada, haverá um campo no qual será necessário colocar o token para logar ou, poderá acessar a url informada no log com parâmetro do token, obtendo o mesmo resultado.

Anotações



Utilizamos cerquilha para títulos

Título

In [1]: nome do notebook e a informação de quando foi salvo pela última vez. Um notebook é qualquer arquivo de apresentação criado no Jupyter com a extensão .ipynb. A tela apresentada no slide, por exemplo, é um notebook.

In [2]: barra de menus.

In [3]: barra de botões. Atalhos para algumas das rotinas mais utilizadas do Jupyter.

In [4]: célula—ou Cell, em inglês. É aqui que a magia acontece.

OBS.: No campo da célula podemos criar scrips python, comentários, e campos de texto.

Anotações

Cada bloco é executado individualmente.

```
In [1]: # Bloco que calcula o factorial  
resultado = 1  
factorial = 5 # número a ser calculado  
for x in range(1, factorial+1):  
    resultado *= x  
print(resultado)
```

120

os blocos de código tem acesso as variáveis uns aos outros.

```
In [2]: print(resultado)
```

120

```
In [3]: # Bloco que calcula a Raiz quadrada  
raiz = 64 # número a ser calculado  
print(raiz ** 0.5)
```

8.0

Célula

Definimos um Campo de texto como título, utilizamos # para definir um título.

In [1]: Determinamos um bloco de código que calcula o fatorial de um número inteiro, lembrando que cada bloco de código é executado individualmente.

Estabelecemos um outro bloco de texto.

In [2]: Definimos um bloco de código, embora cada bloco seja executado individualmente, podemos acessar variáveis de outros blocos.

In [3]: Determinamos um bloco de código que calcula a raiz quadrada de um número.

Obs.: Experimente executar cada bloco de código separadamente, uma das principais funções da ferramenta é que podemos criar blocos na nossa célula para analisar dados e tirar a dependência de um bloco por outro.

Anotações
