Felipe Moz / moz.felipe@gmail.com

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, tunning 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

- SUSE — Sistema Operacional e Openstack
- mailguard — Anti-Virus a Anti-Spam
- The PHP Company — Zend Advanced Partner
- zimbra — Servidor de e-mail e colaboração
- puppet labs — Gerência de configurações
- LPI — Certificação Linux
- EDB ENTERPRISEDB — Banco de Dados PostgreSQL
- moodle — Ambiente de Ensino On-line
- Monitoramento de Ativos de TI — ZABBIX

Saiba mais

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

História e Mercado

Curso: 520

Python Fundamentals

Versão: 4.2

Índice

História e Mercado

Objetivos da Aula

- ✔ Conhecer sua história;
- ✔ Fazer uma introdução ao Python;
- ✔ Quem usa Python.

História e Mercado

Nesta aula iremos apresentar a história do Python e as empresas que o utilizam.

Anotações:

Felipe Moz / moz.felipe@gmail.com

História e Mercado

python™

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

4

4LINUX

O Python surgiu em meados de 1989, criado por Guido Van Rossum, programador holandês que, com base na linguagem ABC que tinha como propósito ensinar programação e prototipagem de programas, seguiu na mesma linha com o Python.

O nome Python foi temporariamente dado pelo autor que era fã da série Monty Python, mas acabou ficando definitivo.

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

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

Em 2015 o Python foi considerado 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

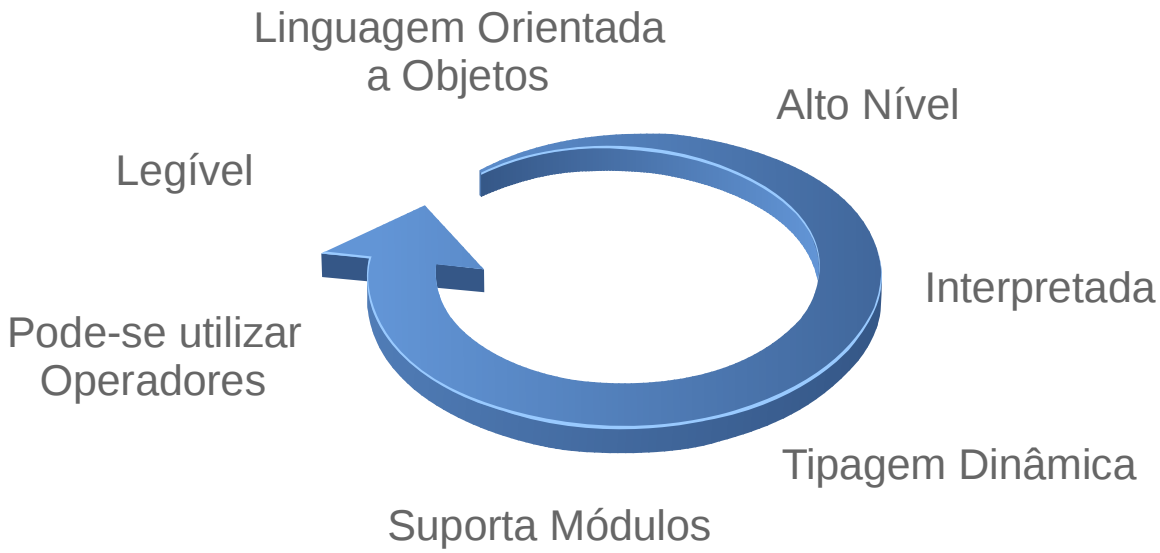
Ficou também em 4º lugar no ranking da RedMonk:
http://redmonk.com/sogrady/2015/07/01/language-rankings-6-15/

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

Anotações:

História e Mercado

Características



5

4LINUX

Conhecendo o Python

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

Interpretada: Linguagem interpretada é uma linguagem de programação em que não há a 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 é o tipo de valor em uma variável, para criar basta atribuir um valor a esta. Ex: idade = 30 mesmo eu não tendo definindo o tipo de valor, ele irá assumir o tipo "int" (inteiro).

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

Pode-se utilizar Operadores: Conseguimos utilizar operadores para adição, subtração, multiplicação, entre outros.

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

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

OpenStack

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

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

OpenStack: https://github.com/openstack

Anotações:

História e Mercado

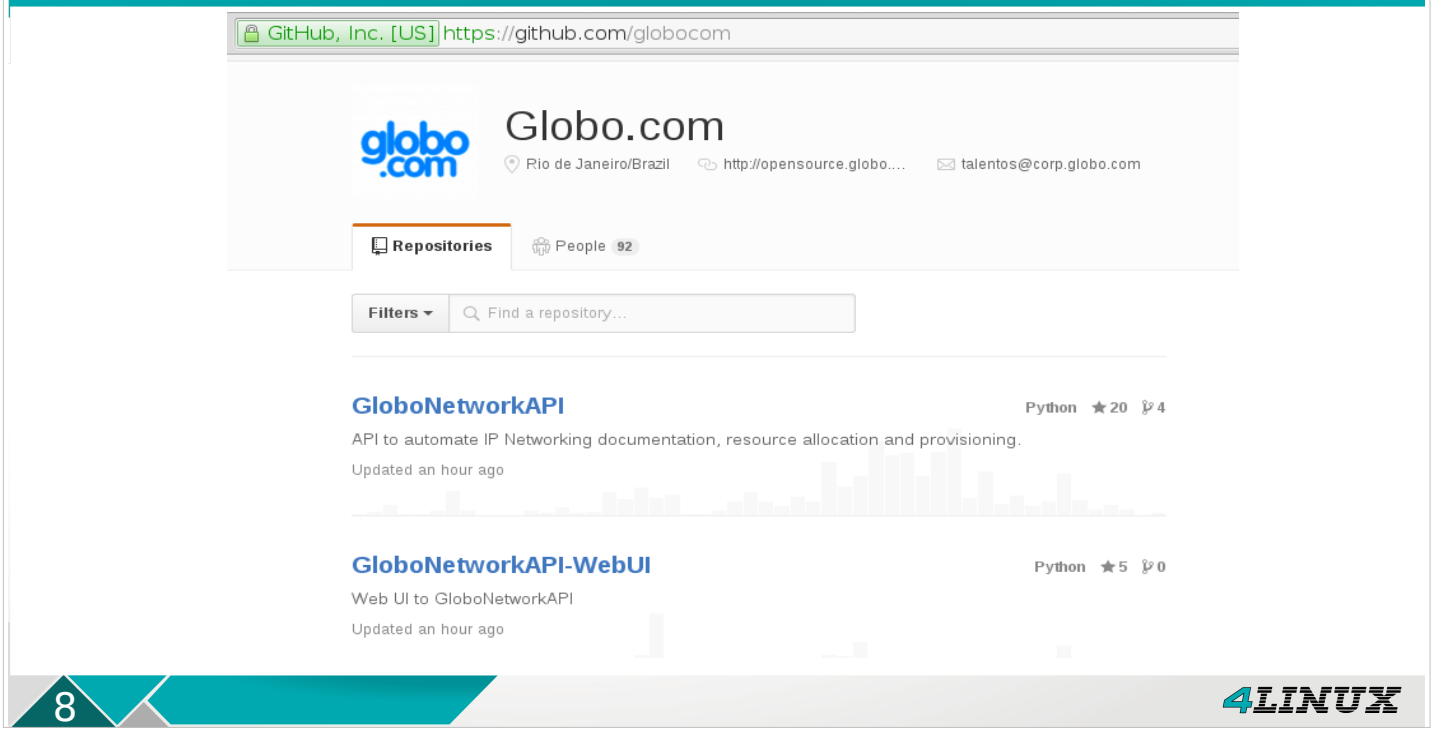
Ansible

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

Ansible também foi escrito em python e o código fonte está disponível no github no seguinte link: https://github.com/ansible/ansible

Anotações:

História e Mercado



Globo

A Globo.com também usa fortemente a linguagem de programação. É possível comprovar isso verificando a quantidade de repositórios na linguagem python que existe no github:

https://github.com/globocom

Anotações:

Sintaxe básica do Python

Objetivos da Aula

- ✔ Conhecer o console interativo;
- ✔ Conhecer Virtual Environment;
- ✔ Interpretador do Python;
- ✔ Identação;
- ✔ Variaveis.

Sintaxe básica do Python

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

Anotações:

Sintaxe básica do Python

Console Interativo

O Python possui um modo interativo que permite “conversar” diretamente com o interpretador e 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 irá imprimir 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 “...” é porque o interpretador está aguardando completar o comando.

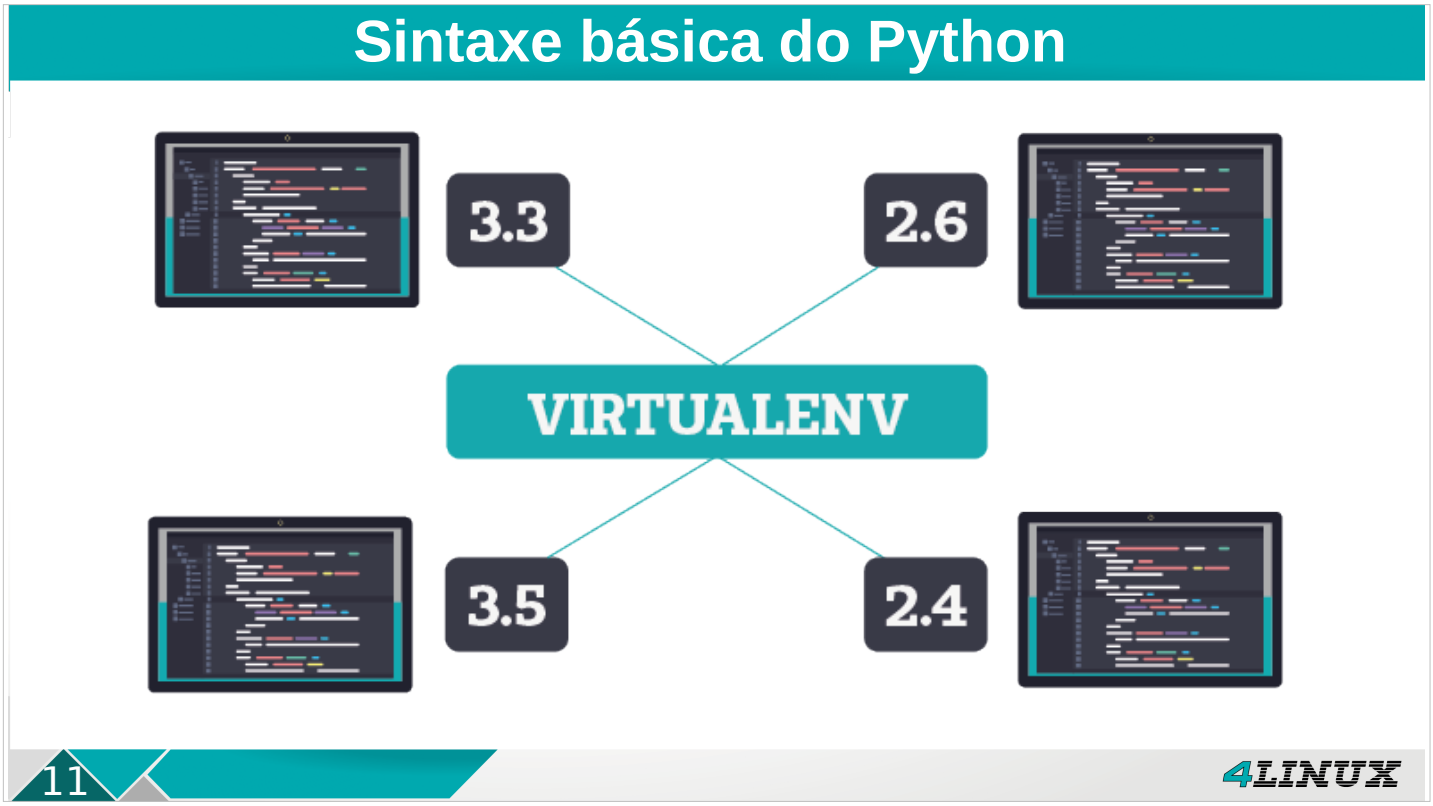
É utilizado para a realização testes antes de colocarmos o código em nossa aplicação, desta forma, conseguimos ver se determinada função ou classe funciona ou tem o retorno esperado.

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

Se quisermos saber mais sobre o que é programar em Python, podemos digitar “import this”. Estes princípios também podem ser encontrados em:

https://www.python.org/dev/peps/pep-0020/

Anotações:



Virtual Environment

O virtualenv é uma ferramenta que utilizamos para criar ambientes isolados quando trabalhamos com o Python.

Temos um problema básico que precisa ser resolvido quando temos diversos projetos, vamos imaginar a seguinte situação: temos uma aplicação que precisa da versão 1 do paramiko (módulo para conexão SSH), mas uma outra aplicação também utiliza este módulo e precisa da versão 2.

Então conseguimos usar o Virtual Environment para manter as duas versões do mesmo módulo sem que isto cause algum conflito.

Anotações:

Sintaxe básica do Python

Instalação e criação do ambiente

1 A instalação do virtualenv é feita através do gerenciador de pacotes pip

```
# pip3 install virtualenv
```

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

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

3 Para ativar as variáveis do projeto criado

```
# source projeto/bin/activate
```

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

```
# deactivate
```


Anotações:

Sintaxe básica do Python

Testando o virtualenv

Após ativarmos o projeto, vamos instalar um módulo e testar se o ambiente irá funcionar.

```
root@developer:/# source projeto/bin/activate
(projeto)root@developer:/# pip install wget
  Downloading wget-3.2.zip
Successfully installed wget
```


Virtual Environment

É importante lembrar que ao instalarmos um módulo dentro do ambiente e posteriormente carregar o bash default ou outros ambientes ele não estará instalado, então caso precise utilizar o módulo em outro ambiente, será necessário instalar novamente.

Isto acontece para evitar o conflito de versão de módulos que pode ocorrer quando trabalhamos com vários projetos diferentes.

Anotações:

Sintaxe básica do Python

Testando o virtualenv

Depois de instalarmos o módulo, podemos testar para ver a versão do módulo instalado e verificar que ele não está instalado fora do projeto:

```
(projeto)root@developer:/# pip show wget
Name: wget
Version: 3.2
(projeto)root@developer:/# deactivate

root@developer:/# pip show wget
```


Anotações:

Sintaxe básica do Python

Interpretador do Python

Ao criarmos scripts em Python, devemos especificar o caminho do interpretador para que ele seja executado na versão correta. Desta forma, vamos adicionar no inicio do nosso script:

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

Para executar o script, podemos utilizar:

```
python3 script.py
```


Interpretador do Python

Para executar os scripts em Python, além da forma descrita acima ainda podemos utilizar ./script.py neste caso é obrigatório ter o interpretador especificado para que não ocorra 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 podermos 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:

Sintaxe básica do Python

Identação

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, diferente de outras linguagens, não possui chaves para separar os blocos de código.

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

No Python é fácil saber quando você está programando errado. Se o seu código estiver muito para a direita, já é possível saber que existem muitas funções aninhadas que provavelmente podem ser separadas, facilitando a manutenção e evitando a duplicação de código.

Anotações:

Sintaxe básica do Python

Comentários

Podemos adicionar **comentários** em nossos códigos. Eles são 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:

Sintaxe básica do Python

Regras básicas para a criação de variáveis

1º	Nunca inicie o nome de variáveis por números.
2º	Nunca utilize caracteres especias, somente o underline (_).
3º	Nunca utilize espaços em branco.
4º	Crie variáveis com nomes abreviados.
5º	Não crie variáveis com nomes sem sentido dentro do código.
6º	Não utilize letras maiúsculas em variáveis.

Anotações:

Sintaxe básica do Python

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")
```


Anotações:

Sintaxe básica do Python

Python Code Style

1º	A Identação deve ser de 4 espaços (Sem Tabs).
2º	Limite 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 da de cima.
6º	Não usar espaço depois de abrir ou fechar um parênteses.

Exemplos:

O python não tem muitas regras quanto à sua forma de programar. Esse code style foi baseado nas práticas dos desenvolvedores do Flask.

Alguns exemplos com as boas práticas são:

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”:
```

É uma recomendação no caso de comparadores sempre usar a variável antes da string ou número a ser comparado.

Caso sejam valores booleanos, fazer 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 comparados is, já foi criado com o propósito de comprar esses valores, então devem ser usados.

Anotações:

Tipos de Dados Básicos

Objetivos da Aula

- ✔ Conhecer como o Python utiliza os dados;
- ✔ Conhecer os Tipos de Dados Básicos.

Tipos de Dados Básicos

Nesta aula iremos conhecer como o Python utiliza os dados e quais são os tipos de dados existentes..

Anotações:

Tipos de Dados Básicos

Tipos de dados em Python

O Tipo do dado é uma forma que utilizamos para classificar a informação. Por exmplo, dentro do meu código números inteiros serão processados diferentes de números decimais.

Toda a informação é obrigatóriamente de um “tipo”.

Em Python, temos os built-ins, que são os os tipos que já estão embutidos no núcleo da linguaguem.

Tipagem de Dados

Muitas linguagens de programação exigem que as aplicações contenham as especificações de que tipo são determinadas variáveis, como por exemplo em JAVA para definir um valor inteiro seria 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 dinamica, ou seja, não precisamos especificar qual é o tipo de uma informação.

Anotações:

Tipos de Dados Básicos

Tipos Numéricos

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

Inteiros	Flutuantes	Complexos	Booleano
`>>> x = 10`	`>>> x = 3.12`	`>>> x= 1 - 2j`	`>>> x = True`
`>>> type (x)`	`>>> type (x)`	`>>> type (x)`	`>>> type (x)`

Tipos de Dados - Numéricos

Em Python existem três tipos de dados numéricos: números inteiros, números de ponto flutuante, números complexos e os Booleanos que são considerados um subtipo dos inteiros.

Os inteiros são identificados por int e são descritos como números de precisão fixa, os de ponto flutuante podemos identificar por float e são números de precisão variável, os números complexos podemos identificar por complex e têm uma parte real e uma parte imaginária e os Booleanos são identificados por bool e sempre teremos um retorno de True ou False.

Anotações:

Tipos de Dados Básicos

Tipos de Sequencia

Os tipos de sequência no Python são: tupla, lista e dicionário.

Tupla

```
>>>x=("a","b")
>>> print (x)
('a', 'b')
>>> type (x)
<class'tuple'>
```

Lista

```
>>> x=[1,"a"]
>>> print (x)
[1, 'a']
>>> type (x)
<class 'list'>
```

Dicionário

```
>>>x={"num":10}
>>> print (x)
{'num': 10}
>>> type (x)
<class 'dict'>
```

25

Tipos de Dados - Sequenciadores

Em Python existem três tipos de dados sequenciadores: tupla, lista e dicionário.

A Tupla é identificada por tuple e são sequências imutáveis geralmente utilizadas parra armazenar um tipo de dado específico, por exemplo números ou string.

A Lista é identificada por list e são sequência mutáveis e dentro delas podemos armazenar qualquer tipo de dados, inclusive podemos ter uma lista dentro de outra lista.

O Dicionário é identificado po dict e trabalham com dados seguinto o tipo CHAVE:VALOR (formato identico ao JSON, são indispensávies quando se trabalha com API. Normalmente não utilizamos um valor numerico no campo “chave”.

Anotações:

Tipos de Dados Básicos

Tipos de Sequencia de Texto

Os dados textais no Python são identificados por “strings”.

```
                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, eles podem ser definidos de duas formas:

Citações simples: 'Python Fundamentals – 4linux'

Aspas: “Python Fundamentals – 4linux”

Anotações:

Estrutura de dados e Conversão de tipos

Objetivos da Aula

- ✔ Entender a Estrutura de dados;
- ✔ Conhecer os métodos;
- ✔ Conversão de tipos.

Estrutura de dados e Conversão de tipos

Nesta aula iremos entender como é a Estrutura dos dados, seus métodos e a conversão de tipos.

Anotações:

Estrutura de dados e Conversão de tipos

Estrutura de dados em Python

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.

Tipagem de Dados

Muitas linguagens de programação exigem que as aplicações contenham as especificações de que tipo são determinadas variáveis, como por exemplo em JAVA para definir um valor inteiro seria 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 dinamica, ou seja, não precisamos especificar qual é o tipo de uma informação.

Anotações:

Estrutura de dados e Conversão de tipos

String Methods

As strings são sequências imutáveis e possuem métodos adicionais, como podemos ver abaixo:

```
>>> x = "Python"
>>> str.upper(x)
'PYTHON'
>>> str.lower(x)
'python'
>>> print (x.replace("on", "ON"))
PythON
```


Outros tipos de String Methods

Além dos métodos demonstrados acima, ainda podemos utilizar:

- **expandtabs**: substitui o espaçamento ou um ou mais espaços.

```
>>> x = '4\tLinux'
>>> x.expandtabs(tabsize=8)
'4       Linux'
```

- **swapcase**: converte os caracteres maiúsculos em minusculos e vice-versa.

```
>>> x = 'PYthon'
>>> x.swapcase()
'pyTHON'
```

Podemos encontrar outros String Methods na documentação oficial do Python:

https://docs.python.org/3/library/stdtypes.html#string-methods

Anotações:

Estrutura de dados e Conversão de tipos

Formas de Concatenar Variáveis

Podemos concatenar variáveis de duas formas, a mais rápida e mais utilizada é utilizando o “%s”. Como por exemplo:

```
nome = "Guido"
idade = 59
x = "O nome do criador do Python é %s e sua
     idade é %s" %(nome,idade)
print (x)
```


Concatenando Strings

Desta forma a primeira vez que colocamos %s será preenchida pela primeira variável (“nome”) e a segnda vez que colocamos %s será preenchida pela segunda variável (“idade”) e assim por diante.

A variável %s pode ser utilizada para concatenar qualquer tipo de dados, é considerada mais rápida e também mais legível no seu código.

Anotações:

Estrutura de dados e Conversão de tipos

Formas de Concatenar Variáveis

A outra forma para concatenar uma string com uma variável string utiliza-se o sinal de mais **(+)** e para concatenar uma string com outros tipos utiliza-se a vírgula **(,)**.

```
nome = "Guido"
idade = 59
x = "O nome do criador do Python é: "+nome
y = "A idade dele é: ",idade
print (x)
print (y)
```


Concatenando Strings

Além dos métodos demonstrados acima, podemos utilizar:

```
x = “O nome do criador do Python é:” +nome+ “e a sua idade é: “,idade
print (x)
```

Ou seja, podemos concatenar no meio de uma frase utilizando +variavel+ para strings ou ,variavel, para qualquer outro tipo de dados.

Anotações:

Estrutura de dados e Conversão de tipos

Dict Methods

Os dicionários possuem alguns métodos adicionais, como podemos ver abaixo:

```
>>> x = {'curso':'python', 'escola':'4linux'}
>>> print (x)
{'escola': '4linux', 'curso': 'python'}
>>> x.keys()
dict_keys(['escola', 'curso'])
>>> x.values()
dict_values(['4linux', 'python'])
```


Outros tipos de Dict Methods

Além dos métodos demonstrados acima, ainda podemos utilizar:

sorted: retorna as chaves de forma ordenada:

>>> x = {'curso':'python', 'escola':'4linux'}

>>> list(x.keys())

['escola', 'curso']

del: apaga uma chave e o seu valor.

>>> x = {'curso':'python', 'escola':'4linux'}

>>> del x['curso']

>>> print (x)

{'escola': '4linux'}

Anotações:

Tipos de Dados Básicos

Tipos Numéricos

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

Inteiros

```
>>> x = 10
>>> type (x)
```

Flutuantes

```
>>> x = 3.12
>>> type (x)
```

Complexos

```
>>> x= 1 - 2j
>>> type (x)
```

Inteiros

```
>>> x = True
>>> type (x)
```


Tipos de Dados - Numéricos

Em Python existem três tipos de dados numéricos: números inteiros, números de ponto flutuante, números complexos e os Booleanos que são considerados um subtipo dos inteiros.

Os inteiros são identificados por int e são descritos como números de precisão fixa, os de ponto flutuante podemos identificar por float e são números de precisão variável, os números complexos podemos identificar por complex e têm uma parte real e uma parte imaginária e os Booleanos são identificados por bool e sempre teremos um retorno de True ou False.

Anotações:

Tipos de Dados Básicos

Tipos de Sequencia

Os tipos de sequência no Python são: tupla, lista e dicionário.

Tupla

```
>>>x=("a","b")
>>> print (x)
('a', 'b')
>>> type (x)
<class'tuple'>
```

Lista

```
>>> x=[1,"a"]
>>> print (x)
[1, 'a']
>>> type (x)
<class 'list'>
```

Dicionário

```
>>>x={"num":10}
>>> print (x)
{'num': 10}
>>> type (x)
<class 'dict'>
```


Tipos de Dados - Sequenciadores

Em Python existem três tipos de dados sequenciadores: tupla, lista e dicionário.

A Tupla é identificada por tuple e são sequências imutáveis geralmente utilizadas parra armazenar um tipo de dado específico, por exemplo números ou string.

A Lista é identificada por list e são sequência mutáveis e dentro delas podemos armazenar qualquer tipo de dados, inclusive podemos ter uma lista dentro de outra lista.

O Dicionário é identificado po dict e trabalham com dados seguinto o tipo CHAVE:VALOR (formato identico ao JSON, são indispensávies quando se trabalha com API. Normalmente não utilizamos um valor numerico no campo “chave”.

Anotações:

Tipos de Dados Básicos

Tipos de Sequencia de Texto

Os dados textais no Python são identificados por “strings”.

```
                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, eles podem ser definidos de duas formas:

Citações simples: 'Python Fundamentals – 4linux'

Aspas: “Python Fundamentals – 4linux”

Anotações:

Entrada e Saída de Dados

Objetivos da Aula

- ✔ Entender como interagir com os usuários;
- ✔ Compreender como exibir os dados.

Entrada e Saída de Dados

Nesta aula iremos entender como interagir com o usuário usando o input e compreender as formas de exibir os dados para o usuário.

Anotações:

Entrada e Saída de Dados

Entrada de Dados

Até este momento trabalhamos de forma estática, ou seja, sem interagir com o usuário. O Python permite que o usuário faça inserção de dados.

Os dados inseridos pelo usuário, sempre serão interpretador como “string” e deverão ser tratados em nosso programa.

O que foi inserido pelo usuário é armazenado em uma variável e utilizada no programa.

Entrada de Dados

No Python 3 a entrada de dados é feita apenas do comando “input” e ele irá tratar todas as inserções da mesma forma (como string).

No Python 2 existem dois comandos para fazer a entrada de dados, sendo “raw_input” para guardar dados do tipo texto e “input” para guardar qualquer outro tipo de dado. Como 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:

Entrada e Saída de Dados

Utilizando o Input

Para interagir com usuário, utilizaremos o “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)
```


Anotações:

Entrada e Saída de Dados

Utilizando o Input

```
#/usr/bin/python3

animais = []

animal = input('Digite o nome do animal: ')

animais.append(animal)

print (animais)
```


Utilização do Input

Quando colocamos o valor de uma variável como um **"input"** esta variável receberá o que foi digitado e a partir de então assumirá este valor.

Por exemplo, se eu tenho uma variável "nome" com um input "Qual o seu nome?" e o usuário digita "Maria" então a minha variável nome receberá o nome de Maria.

Sabendo disso, eu posso utilizar esta variável em outras partes do meu programa e seu valor será igual ao que o usuário digitou.

Conseguimos trabalhar com listas, como visto acima ou com outras funcionalidades do Python (como por exemplo estrutura de decisão e laços de repetição).

Anotações:

Entrada e Saída de Dados

Saída de Dados

Utilizamos a função “print ()” para retornar os dados do programa para a saída padrão (a tela).

Esta função também é utilizada caso o nosso programa esteja com algum comportamento inesperado, com ela conseguimos ver se estamos recebendo os valores esperados.

Além desta função, podemos enviar a saída de dados para um arquivo e armazena-la para uso futuro.

Saída de Dados

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

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

Podemos notar também que agora no Python 3 conseguimos imprimir uma quantidade de valores e é possível definir o seu separador. É possível ver todas as modificações na documentação oficial:

https://docs.python.org/3.0/whatsnew/3.0.html

Anotações:

Entrada e Saída de Dados

Utilizando o Print

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

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

Podemos definir o separador quanto retornamos dois valores:

```
produto = "Computador"
quantidade = 10
print (animais, produto, sep = ".")
```


Concatenando Strings

Além dos métodos demonstrados acima, podemos utilizar:

```
x = “O nome do criador do Python é:” +nome+ “e a sua idade é: “,idade
print (x)
```

Ou seja, podemos concatenar no meio de uma frase utilizando +variavel+ para strings ou ,variavel, para qualquer outro tipo de dados.

Anotações:

Entrada e Saída de Dados

Métodos para Saída de Dados

Podemos filtrar e arredondar valores do tipo float:

```
#/usr/bin/python3
nota = 5.4578921456
print (round (nota,1))
print (round (nota,2))
print (round (nota,3))
```

O retorno do script acima deve ser a nota com apenas uma casa decimal, com duas casas decimais e com três casas decimais.

Saída de Dados

Podemos utilizar o método **"format"** e criar uma saída utilizando determinadas variáveis no próprio print:

```
print ("O produto é {produto} e a quantidade é {quantidade}"
                                                .format(
                                                produto='Computador',
                                                Quantidade='10'))
```

O método **"zfill"** irá preencher a esquerda com zeros (este método só funciona para texto):

```
num = '3.156'
print (num.zfill(8))
```

Anotações:

Manipulação de Arquivos

Objetivos da Aula

- ✔ Compreender leitura de arquivos;
- ✔ Compreender gravação de dados.

Manipulação de Arquivos

Nesta aula iremos trabalhar com arquivos, veremos os métodos disponíveis para realizar a leitura e escrita dentro de arquivos de texto.

Anotações:

Manipulação de Arquivos

Manipulação de Arquivos

Para trabalharmos 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. Vamos trabalhar com os arquivos de texto.

Manipulação de Arquivos:

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

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

Modos e trabalhar com o arquivo:

r – utilizada para leitura de arquivos

w – utilizada para a escrita em arquivos

r+ – utilizada para a leitura e escrita em arquivos

Em arquivos binários colocamos o “b” na frente dos modos para especificar que o tipo de arquivo é um binário.

Manipulação de Arquivos

Todos os dados usados até agora foram inseridos pelo usuário ou colocados diretamente no código. Agora vamos trabalhar com arquivos. A sintaxe será:

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

"open()" retornará um objeto de arquivo, e é sempre utilizado seguindo o padrão: nome do arquivo e o modo (leitura ou escrita).

Anotações:

Manipulação de Arquivos

Modos disponíveis

Os modos disponíveis para trabalhar com arquivos são:

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

Podemos combinar alguns modos, como por exemplo: r+ (irá abrir o arquivo e permitir a leitura e escrita).

Modos de abertura

Podemos utilizar outros modos quando trabalhamos com arquivos existe um modo específico para trabalharmos com arquivos do tipo binário, a documentação do Python 3 nos mostra como utilizar este modo:

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

Anotações:

Manipulação de Arquivos

Escrevendo em arquivos

```
#/usr/bin/python3

with open('arquivo', 'w') as f:

   f.write('Curso de Python')
```

Desta forma os dados serão sobreescritos, ou seja, este modo é indicado para gravar os dados inicialmente.

Trabalhando com arquivos

Quando trabalhamos com arquivos não é obrigatório o uso do **"with"** porém, utilizamos como uma boa prática. A principal vantagem que ele nos trás é eliminar a necessidade de fechar o arquivo após terminar de executar o que for necessário.

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

Anotações:

Manipulação de Arquivos

Podemos escrever em um determinado ponto do arquivo do arquivo:

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

O **"seek"** é quem definirá a posição de meu ponteiro, onde o 0 (zero) indica a primeira linha.

Anotações:

Manipulação de Arquivos

Lendo arquivos

```
#/usr/bin/python3

with open('arquivo', 'r+') as f:

   print f.read()
```

O read permite que você leia o arquivo todo. Temos também o **readlines** que trará todo o conteúdo em forma de lista.

Lendo arquivos

Podemos utilizar os dois métodos quando utilizamos o conteúdo a partir de arquivos de texto.

Para obter uma linha especifica do arquivo, poderímos 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 um dicionário, eu acesso um objeto através do seu indice e a partir de então consigo ter o retorno de uma só linha.

Anotações:

Manipulação de Arquivos

Métodos para Saída de Dados

Podemos filtrar e arredondar valores do tipo float:

```
#/usr/bin/python3
nota = 5.4578921456
print (round (nota,1))
print (round (nota,2))
print (round (nota,3))
```

O retorno do script acima deve ser a nota com apenas uma casa decimal, com duas casas decimais e com três casas decimais.

Saída de Dados

Podemos utilizar o método **"format"** e criar uma saída utilizando determinadas variáveis no próprio print:

```
print ("O produto é {produto} e a quantidade é {quantidade}"
                                                .format(
                                                produto='Computador',
                                                Quantidade='10'))
```

O método **"zfill"** irá preencher a esquerda com zeros (este método só funciona para texto):

```
num = '3.156'
print (num.zfill(8))
```

Anotações:

Trabalhar com if e else

Objetivos da Aula

- ✔ Conhecer os operadores;
- ✔ Aplicar estruturas do tipo IF/ELSE;
- ✔ Trabalhar com ELIF.

Trabalhar com if e else

Nesta aula iremos os operadores, trabalhar com estruturas de condição do tipo if/else e ver o uso do elif.

Anotações:

Trabalhar com if e else

Tipos de Operadores

No Python temos os dois tipos de operadores:

Os **operadores aritméticos** que são utilizados quando vamos fazer algum tipo de cálculo.

Os **operadores relacionais** que são utilizados para comparações, retornando **true** ou **false** em seus resultados.

Os **operadores lógicos** que são utilizados para testes condicionais.

Anotações:

Trabalhar com if e else

Os **operadores aritméticos** são:

```
#!/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:

Trabalhar com if e else

Podemos trabalhar com eles de forma abreviada:

```
#!/usr/bin/python3
number = 1;  # A variável number recebe 1
number += 2; # Somamos 2 a variável
number -= 2; # Subtraimos 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:

Trabalhar com if e else

Os **operadores relacionais** são:

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


Anotações:

Trabalhar com if e else

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

Os operadores são meios para manupular dois valores (operandos), como por exemplo:

5 + 10 = 15

Neste caso o 5 e 10 são chamados de operandos e o + é chamado de operador.

Os operadores nos ajudam principalmente quando trabalhamos com estruturas de decisão.

Anotações:

Trabalhar com if e else

Estruturas de Decisão

Estruturas de decisão servem para 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                if condição:
   comandos;                       print “verdadeiro”
caso contrário faça             else:
   comandos;                       print “falso”
```


Estruturas de decisão

As estruturas de controle irão testar uma determinada condição, caso esta seja verdadeira ela executará o bloco do “if”, se ela for falsa executará o bloco do “else”.

Os comandos que devem ser 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:

Trabalhar com if e else

Estruturas de Decisão

A estrutura de decisão IF irá testar uma condição e caso ela seja verdadeira, o que estiver dentro do seu bloco de código será executado, como por exemplo:

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


Anotações:

Trabalhar com if e else

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, como por exemplo, imagine que para dirigir você precisa ter no mínimo 18 anos e precisa ter carteira de habilitação ou estar acompanhado dos pais. O código neste caso ficaria da seguinte forma:

```
#!/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')
```

Anotações:

Trabalhar com if e else

Encadeamento de condição

Em programação existe ainda o que nós chamamos de encadeamento de condições, ou seja, podemos ter diversos ifs dentro de uma mesma estrutura.

```
se(condição) faça
   comandos;
senãose(condiçao)
   comandos;
senão
   comandos
fim
```

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


4LINUX

Encadeamento de condição

Usamos o elif para verificar mais de uma condição, precisamos deste comando quando é necessário entrar em somente um dos ifs encadeados. Caso precise fazer 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')
```

Anotações:

Conhecer os laços de repetição for e while

Objetivos da Aula

- ✔ Conhecer o for;
- ✔ Conhecer o while;
- ✔ Entender a diferença entre for e while.

Conhecer os laços de repetição for e while

Nesta aula iremos conhecer os laços de repetição for e while, entender as diferenças entre eles e criar alguns exemplos práticos.

Anotações:

Conhecer os laços de repetição for e while

Laço de repetição while

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

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

```
while condiçao:
    comandos
```

Atenção: É muito comum não nos atentarmos em como o laço irá terminar. Para que o laço termine precisamos fazer com que a condição se torne false em algum momento.

Anotações:

Conhecer os laços de repetição for e while

Utilizando o while

Podemos utilizar o while da seguinte forma:

```
#!/usr/bin/python3
x = 1
while x < 10:
    print ("Número: %s" %x)
    X += 1
print ("O while acabou :)")
```


Operadores

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

Como por exemplo:

```
#!/usr/bin/python3
  X = 1
  while True:
      print ("Número: %d!" % (x))
      x += 1
```

Em Python não existe o laço de repetição DO/WHILE como em outras linguagens, na PEP 0315 o autor explica o porque de não ter implementado:

https://www.python.org/dev/peps/pep-0315/

Anotações:

Conhecer os laços de repetição for e while

Laço de repetição for

A instrução **for** é perfeita quando trabalha-se com listas no python, mas ela também pode ser utilizada da maneira tradicional, indo de um valor inicial para um valor final.

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

```
for item in lista:
   print item
```

Atenção: O laço for é utilizado para executar um montante fixo. Como por exemplo percorrer uma lista ou executar até uma quantidade de vezes.

Anotações:

Conhecer os laços de repetição for e while

Utilizando o for

Podemos utilizar o for da seguinte forma:

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

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


Anotações:

Conhecer os laços de repetição for e while

No for podemos utilizar uma função chamada “range”, ela 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() é uma forma muito mais fácil de gerar uma sequencia de números sem ter que 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 ("%s esta na posicao %s"%(item,num))
```

Anotações:

List Comprehension

Objetivos da Aula

- ✔ Conhecer List Comprehension;
- ✔ Para que utilizar List Comprehension;
- ✔ Utilizando List Comprehension.

List Comprehension

Conhecer o List Comprehension, entender porque utilizamos e como podemos usar em nossa aplicação.

Anotações:

List Comprehension

List Comprehesion

No Python temos uma construção chamada List Comprehension que gera uma lista de forma clara e concisa.

Como conseguimos colocar qualquer objeto dentro de uma lista, também é possível colocar uma expressão e obter um resultado a partir disto.

A List Comprehension sempre irá retornar uma lista como o resultado.

List Comprehension

O List Comprehension irá gravar o resultado da expressão que foi aplicada dentro da lista.

É uma maneira simples e legível de criar expressões e gravar seu resultado, ou seja, conseguimos utilizar um for e/ou if dentro de uma lista e o seu resultado poderá ser utilizado na aplicação posteriormente.

Utilizar List Comprehension é também uma forma de deixar o seu código mais limpo e organizado.

Para obter maiores informações sobre isto, podemos consultar a documentação oficial:

https://docs.python.org/3/tutorial/datastructures.html

Anotações:

List Comprehension

Sintaxe

Para conseguirmos chegar ao resultado, podemos utilizar da seguinte forma:

```
[<expressão> for <item> in <sequencia>]
```

Ainda podemos combinar o uso do for com if:

```
[<expressão> for <item> in <sequencia> if <condição>]
```


Anotações:

List Comprehension

Utilizando List Comprehension

Imagine que temos uma instrução for para percorrer uma lista:

```
lista = [0, 1, 2, 3, 4]
for item in lista:
   print (item)
```

Podemos transformar isto em apenas uma linha:

```
num = [ item for item in range(10) ]
print (num)
```


Anotações:

List Comprehension

Utilizando List Comprehension

Imagine que temos uma instrução para saber apenas os números pares em uma lista:

```
#!/usr/bin/python3
lista = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
for l in lista:
    if l %2 == 0:
        print (l)
```


Anotações:

List Comprehension

Utilizando List Comprehension

Para fazer o mesmo mas utilizando List Comprehension:

```
#!/usr/bin/python3
a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
pares = [p for p in a if p %2 == 0]
print (pares)
```


Anotações:

Escopo global e local

Objetivos da Aula

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

4LINUX

Escopo global e local

Nesta aula iremos apresentar as funções, sua sintaxe, o escopo de uma função e como trabalhar com variáveis globais e locais.

Anotações:

Escopo global e local

Entendendo funções

Função é uma sequencia de instruções que realiza uma operação desejada.

Podemos definir nossas próprias funções para resover um problema da nossa aplicação.

Podemos utilizar qualquer nome para a nossa função, porém é recomentado pela PEP8 que os nomes sejam sempre minusculo e, caso necessário, sejam separados por _ para falilcitar a legibilidade.

Entendendo funções

Podemos utilizar as funções para reaproveitamento de código, imagine que temos uma aplicação que terá que executar um determinado trecho. Temos duas opções: reescrever diversas vezes a mesma coisa ou criar uma estrutura que execute este trecho quando for necessário.
Criar funções nos ajuda a deixar o código menor, mais organizado e até mais legível. Podemos ver mais sobre funções na documentação oficial:
https://docs.python.org/3/tutorial/controlflow.html#defining-functions

Anotações:

Escopo global e local

Criando funções

A sintaxe para definir uma função:

```
#!/usr/bin/python3

def nome_da_funcao(parametros):

    comandos
```

Uma função pode ter quantos comandos forem necessários, mas devemos lembrar que devem estar identados.

Criando funções

Ao definirmos um parametro, significa que obrigatóriamente a função espera receber algum dado. Se definirmos desta forma e não passarmos nenhuma informação, ocorrerá erro. Para definir uma função sem nenhum parametro basta deixar desta forma:
def nome_da_funcao()

Abaixo segue um exemplo de programa utilizando funções. A primeira possui 1 parâmetro obrigatório e a segunda não recebe parâmetros.
Parâmetro é o valor entre parênteses nas funções, é 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(str(produto))
     print ("produtos cadastrados")
     listarProdutos()
```

Escopo global e local

As funções também podem possuir parâmetros opcionais, e 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, o mesmo deve ter um valor padrão, que será assumido caso o valor não seja passado.

Funções com parametro padrão

Quando estamos escrevendo uma aplicação é comum utilizarmos o “pass” pois Um exemplo de função com parâmetro opcional seria o cálculo de uma compra com cupom de desconto. Só será calculado o desconto 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")))
```

Escopo global e local

Quando definimos uma função é necessário escrever suas instruções. Porém quando estamos apenas definiindo a estrutura da nossa aplicação é comum utilizarmos a declaração **“pass”**

```
#!/usr/bin/python3

def nome_da_funcao(parametros):

    pass
```


Código da função

Quando estamos escrevendo uma aplicação é comum utilizarmos o “pass” pois conseguimos nos dedicar a uma outra parte do código e não esquecer de voltar para implementar a funcionalidade especifica para aquela função.
Quando definimos com o “pass” o Python interpretará como “não execute nada” e desta forma não acontecerá nenhum erro.
Podemos ver maiores informações sobre o pass na documentação oficial:
https://docs.python.org/3/tutorial/controlflow.html#pass-statements

Anotações:

Escopo global e local

Escopo da função

Uma função pode utilizar variáveis de escopo local, com isso podemos definir a mesma variável em funções diferentes.

```
def sistema():

    nome = 'Linux'

def curso():

    nome = 'Python'
```


Escopo da função

Ao definirmos variáveis de escopo local podemos utilizar a mesma variável em outra parte do código sem que ela seja sobreescrita. Ou seja, as variáveis que forem definidas dentro de uma função não pode mudar o valor de varáreis definidas fora desta função.
Quando a aplicação esta sendo executada, irá procurar as variáveis locais (que fazem parte da função) e caso não encontre ela irá procurar as variáveis globais.

```
#!/usr/bin/python3

servidor = "192.168.0.2"

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


def classe_b():
    servidor = '172.32.0.1'
    print (servidor)


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

Escopo global e local

Escopo da função

Uma função pode utilizar variáveis de escopo local, com isso podemos definir a mesma variável em funções diferentes.

```
def sistema():

    nome = 'Linux'

def curso():

    nome = 'Python'
```


Escopo da função

O escopo da função em programação serve para que possamos usar variáveis com o mesmo nome, sem Precisar sobrescrever o valor de uma variável existente. No entanto, se necessário, pode-se sobrescrever o valor de uma variável global dentro de uma função.

```
#!/usr/bin/python3

servidor = "192.168.0.2"

def alterarServidor(ip):
    global servidor
    servidor = ip


print ("Servidor atual é ",servidor)
```

Anotações:

Escopo global e local

Passando parametros para função

Para passarmos um parametro para a função, podemos fazer da seguinte forma:

```
#!/usr/bin/python3

def nome_da_funcao(parametro):

   print (x)

nome_da_funcao('Aqui eu passo o parametro')
```


Parametros

Ao definirmos uma função podemos optar se ele deve receber ou não algum parametro. Ainda é possível Passar mais de um parametro para a função, neste caso, poderíamos fazer desta forma:

```
#!/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:

Args e Kwargs

Objetivos da Aula

- ✔ Entender Args;
- ✔ Entender Kwargs.

Args e Kwargs

Nesta aula iremos apresentar o Args e Kwargs, abordar seu uso e entender como conseguimos aplicar estes parametros em nossas aplicações.

Anotações:

Args e Kwargs

Entendendo Args e Kwargs

Quando não sabemos quantos argumentos serão passados a uma função podemos utilizar o *args ou o **kwargs.

O *args é utilizado para transformar os argumentos em uma lista, enquanto o **kwargs é usado transformar os argumentos em um dicionário.

Não é necessário utilizar o *args ou **kwargs, apenas o * ou ** fará a conversão, utilizamos desta forma por convenção.

Entendendo Args e Kwargs

O uso do *args e **kwargs é muito comum quando não sabemos a quantidade de argumentos que receberemos, mais informações sobre este parametro podemos ver na documentação oficial: https://docs.python.org/3/tutorial/controlflow.html#unpacking-argument-lists

Anotações:

Args e Kwargs

Args

Para criar uma função utilizando o args, podemos fazer:

```
#!/usr/bin/python3

def nome_da_funcao(*args):

    comandos
```

Desta forma, todos os argumentos que forem passados serão convertidos em uma lista.

Anotações:

Args e Kwargs

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 utilizar o Args e receber os argumentos em formato de lista, devemos tratar a lista em nossa aplicação paraconseguir o resultado desejado.
Por exemplo, se o chamar uma função eu passo os seguintes parametros: 4linux e python, para utilizar isto em minha aplicação eu uso o conceito de listas e irei usar args[0] para exibir 4lilnux e args[1] para exibir python (pois as listas são acessadas através de indice).
No exemplo abaixo vamos calcular a área do quadado e do retangulo utilizando o *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)
```

Args e Kwargs

Kwargs

Para criar uma função utilizando o Kwargs, podemos fazer:

```
#!/usr/bin/python3

def nome_da_funcao(**kwargs):

    comandos
```

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

Anotações:

Args e Kwargs

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 o Kwargs os argumentos serão recebidas em forma de dicionário (chave-valor), para usar dicionário na aplicação podemos acessar a sua chave e assim obtemos o seu valor.
Por exemplo, se o chamar uma função eu passo os seguintes parametros: sistema=linux e linguagem=python, para utilizar isto em minha aplicação eu uso o conceito de dicionário e 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 abaixo vamos ver todas as chaves e os valores do dicionário passado.

```
#!/usr/bin/python3

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


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

Funções Anônimas

Objetivos da Aula

- ✔ Entender funções anônimas
- ✔ Utilizar funções anonimas.

Funções Anônimas

Nesta aula iremos apresentar as funções anônimas, entender como elas funcionam e criar alguns exemplos.

Anotações:

Funções Anônimas

Entendendo Funções Anônimas

- Funções anônimas são aquelas que não estão vinculadas a um nome, em Python podem ser chamadas também como “expressões lambda”.
- São muito utilizadas no meio acadêmico pra a 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 palavraa reservada “lambda” para definir funções anonimas, utilizando esta palavra o retorno da expressão será um objeto de função, por este motivo são chamadas de funções anônimas. Pode ser usada para cálculos matemáticos ou quando precisamos de uma função que não seja tão complexa que possa ser executada em uma única linha.

Anotações:

Funções Anônimas

Sintaxe

Ao definirmos uma função anônima, utilizamos a seguinte sintaxe:

```
lambda argumentos: expressão
```

É importante lembrar que “lambda” é uma palavra reservada em Python, então não podemos definir nenhuma variável com este nome para que não ocorra nenhum erro.

Anotações:

Funções Anônimas

Utilizando Funções Anônimas

```
#!/usr/bin/python3

var = lambda x: x*2

print (var(2))
```

No exemplo acima, a função espera receber um argumento (valor de x) e depois utiliza este número para fazer a multiplicação.

Utilizando Funções Anônimas

Uma função lambda é muio útil para resolvermos cálculos matemáticos. Um exemplo do mundo real pode ser quando acontece a black friday e um produto terá 50% de desconto, para isto podemos fazer da seguinte forma:

```
#!/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 ("=-=-=-=-=-=-=-=-=-=-=")
```

Funções Anônimas

Passar mais de um número

Podemos passar mais de um número para fazer o cálculo, como por exemplo:

```
#!/usr/bin/python3

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

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


Anotações:

Funções Anônimas

Criando mais de uma função

Podemos criar mais de uma função:

```
#!/usr/bin/python3

anonimas = [lambda x: x** 2,

            lambda x: x** 3,

            lambda x: x** 4]

for a in anonimas:

    print (a(10))
```


Anotações:

Decoradores

Objetivos da Aula

- ✔ Entender Decoradores;
- ✔ Entender o funcionamento de um Decorador;
- ✔ Utilizar Decoradores.

Decoradores

Nesta aula iremos apresentar o conceito de Decoradores, entender como podemos criar um decorador e como vamoms utilizar decoradores já prontos.

Anotações:

Decoradores

Entendendo Decoradores

- Um decorador é uma forma de adicionarmos novas funcionalidades as nossas funções sem precisarmos alterar o código delas.
- Em Python uma função pode ser passada como valor para uma segunda função e também pode ser retornada como valor desta segunda função.
- Quando fazemos desta forma, temos um **“decorator”**.

Entendendo Decoradores

Em Python um função/método podem ser passadas como valores para outras funções e o retorno de uma função também pode ser uma outra função.

Podemos ver maiores informação na documentação oficial:

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

Anotações:

Decoradores

Sintaxe

Para criar um decorador, devemos fazer desta forma:

```
def func1():
    def func2():
        return "Sintaxe"
    return func2
```

Temos uma função e seu conteúdo é outra função. O retorno de func1 é func2, ou seja, uma função.

Anotações:

Decoradores

Usando decoradores

Após termos criados um decorador, podemos adicioná-lo em nossas funções e fazermos as modificações necessárias.

Uma vez definido, existem duas formas de chamar um decorador:

```
@nome_do_decorator
def func_nova( ):
    comandos
```

```
d = func(parametro)
d(parametro)
```


Anotações:

Decoradores

Usando Decoradores

```
def func(x):
    def decorador(y):
        return x+y
    return decorador
d = func(10)
print (d(15))
```


Anotações:

Decoradores

Usando Decoradores

A forma mais utilizada:

```
def meu_decorador(linguagem):
    def func():
        print ('A linguagem é: %s' %linguagem())
        return linguagem()
    return func

@meu_decorador
def linguagem():
    return ('Python')

linguagem()
```


Usando decoradores

Normalmente para chamar um decorador utilizamos @nome_do_decorador, esta é a foma mais utilizada pois torna o seu código mais simples e legível. Imagine que vamos criar um decorador para gravar log, podemos defini-lo em todas as funções (e assim ter log de execução em cada função que for executada).

```
def meu_decorador(linguagem):
    def wrapper():
        with open('teste.txt','a') as f:
            f.write('gerando log da função %s \n' % linguagem.__name__)
        print ('Gravando log \n%s' %linguagem())
        return linguagem()
    return wrapper

@meu_decorador
def log1():
    return ('Log1 OK')
@meu_decorador
def log2():
    return ('Log2 OK')

log1()
log2()
```

Decoradores

Usando Decoradores

Quando temos uma função e executamos:

```
print (func.__name__)
```

O retorno será o nome da função, porém a forma como fizemos o decorator até agora, ele não retornará o nome da função e sim do decorator.

Isto acontece pois devemos usar o módulo “wraps” para evitar que isto ocorra.

Anotações:

Decoradores

Usando Decoradores

Definindo o **wraps**:

```
from functools import wraps

def meu_decorador(linguagem):
    @wraps(linguagem)

...

print (linguagem.__name__)
```


Usando decoradores

Temos que definir desta forma pois é importante quando fizermos o Reflection.

O Reflection é descobrir de um objeto o seu tipo, classe, atributos e métodos. Isso ajuda a aplicação em sua execução, se não definirmos o decorator desta forma e deixar que ele assuma o nome de outra função, pode ocorrer algum erro.

Anotações:

Orientação a Objeto

Objetivos da Aula

- ✔ Compreender o que é Orientação a Objetos;
- ✔ Conehcer conceitos básicos da Orientação a Objetos.

Orientação a Objeto

Nesta aula iremos apresentar o conceito de Orientação a Objeto e entender o porque é importante implementar POO (Programação Orientada a Objeto) em nossos projetos.

Anotações:

Orientação a Objeto

A **orientação a objetos** é um paradigma de programação que tenta abstrair a programação para coisas do mundo real. Python é uma linguagem que aceita diversos paradigmas e, neste escopo de paradigmas, possui um suporte bem maduro para o desenvolvimento orientado a objetos.

Em **orientação a objetos**, um software não é composto por um grande bloco de funcionalidades específicas, e sim por vários blocos de funcionalidades distintas e independentes que, juntas, montam um sistema.

Anotações:

Orientação a Objeto

Origem

Um dos criadores desse conceito foi **Alan Kay**, que possuía conhecimento em biologia e 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**, que tornou possível a criação da interface gráfica para os computadores.

Anotações:

Orientação a Objetos

Caracteristicas

Uma linguagem é caracterizada como **Orientada a Objetos** quando atende aos quatro tópicos seguintes:

Anotações:

Orientação a Objetos

A linguagem **Python** foi criada totalmente sobre esse conceito de orientação a objetos e nos possibilita utilizar vários desses recursos nativamente.

A principal função da **Orientação a Objetos** é o reuso do código, mas seu uso nos traz muitos outros benefícios como facilitar a organização do código e a legibilidade do mesmo.

Anotações:

Orientação a Objetos

Vantagens

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

- **Reutilização de código** (poupando tempo, tanto programando quanto debugando);
- **Escalabilidade** (adicionar código mais facilmente);
- **Manutenibilidade** (código mais fácil de manter);
- **Agilidade de desenvolvimento**.

Anotações:

Orientação a Objeto

Conceitos básicos

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

Conceitos Básicos de Orientação a Objeto

Classes: Uma classe é a abstração de alguma coisa do mundo real em forma de código. Exemplo: Quando pensamos no mundo DevOps, sabemos que temos um um servidor. Este servidor será a nossa classe Principal.

Objetos: Um objeto é uma instância dessa classe. Exemplo: Um Servidor tem as suas características como endereço de IP, usuários, serviços, etc.

Atributos/Propriedades: Os atributos ou 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 de IP, Serviços sendo executados, Memória, Disco, CPU e etc.

Anotações:

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

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

Assinatura: Significa o retorno do método. Exemplo: Uma função que faz uma 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 (endereco = None).

Retorno: O retorno é quando queremos resgatar e trazer alguma informação. Usamos a palavra reservada *return* para resgatar o valor esperado.

Anotações:

Classes, propriedades e métodos

Objetivos da Aula

- ✔ Conhecer Classes e Objetos;
- ✔ Aprender Propriedades e Métodos.

Classes, propriedades e métodos

Nesta aula iremos conhecer Classes, Objetos e trabalhar com o conceito de propriedades e métodos.

Anotações:

Classes, propriedades e métodos

Entender o que são Classes e Objetos

Antes de começarmos a ver na prática o que é orientação a objetos, precisamos entender algo que confunde muitos iniciantes neste novo mundo chamado de **OO**:

As classes definem as características e o comportamento dos seus objetos. Cada característica é representada por um atributo e cada comportamento é definido por um método. Porém, é importante 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? Isso dependerá do sistema! Não existe uma receita de bolo, pois cada programador tem a sua visão e experiência. Dessa forma, julga os objetos de um determinado sistema de modo diferente. Quanto mais complexo o sistema, mais difícil e diferente é essa análise, mas podemos observar que para sistemas mais simples o julgamento é mais uniforme. Vamos ver um exemplo de um programa para clínicas médicas:

*"O sistema deve cadastrar **pacientes**, **médicos**, e **consultas**. Deve ser possível anexar uma ou mais **receita**s na consulta e também deverá ser possível relacionar pacientes com **exame**s e **resultado**s dos exames."*

As palavras em negrito no texto se referenciam às entidades observadas no sistema. Note que mesmo em casos bem simplificados há divergências: resultado do exame é ou não é um objeto? Isso vai depender de você, programador. Se você observar que o resultado do exame está desempenhando um papel importante no sistema e/ou que ele por si só possui atributos vinculados e um comportamento separado, podemos estruturá-lo como um objeto. Se ele é meramente uma propriedade do objeto exame, então podemos deixá-lo de fora.

Classes, propriedades e métodos

Classes são como a planta baixa de uma casa, a especificação técnica de uma cadeira ou o projeto de um veículo.

Ou seja, **a classe apenas define como será o objeto, mas ela não é o objeto em si**.

Voltando ao exemplo da casa: podemos construir 15 casas a partir de uma mesma planta baixa, não podemos?

Anotações:

Classes, propriedades e métodos

Pode-se dizer, então, que todas essas “casas” são objetos produzidos a partir de uma classe. Dito de outra forma, esses objetos “casas” possuem todas as características e comportamentos definidos na especificação da classe “planta”.

Seguindo este raciocínio, é importante frisar que, ainda que a planta baixa defina uma casa, ela não é a casa em si.

Anotações:

Classes, propriedades e métodos

Portanto, **uma classe NÃO é um objeto** e sim uma abstração de sua estrutura, na qual podemos definir as características que vão compor um objeto. Mas, esses são apenas conceitos. Veremos como isso funciona no **Python**.

Anotações:

Classes, propriedades e métodos

Estrutura de uma classe

Para criar classes em Python, existe uma palavra reservada **"class":**

```
class NomedaClasse:

    Atributo


def metodo(self):

    codigo
```


Anotações:

Classes, propriedades e métodos

Método construtor

O Método construtor serve para executar algo quando instanciamos uma classe, por exemplo, setar os valor default, ou exibir alguma mensagem na tela.

```
class NomedaClasse:

    def __init__(self):

        pass
```


Entendendo o método construtor

O método construtor será executado toda vez que instanciarmos a classe, então para utilizarmos devemos pensar no que é necessário executar quando chamarmos a classe.
Imagine o seguinte exemplo, temos uma conexão SSH e alguns métodos para executar determinados comandos. Independente de qual método for instanciado, a conexão precisará ser realizada. Desta forma, podemos colocar a conexão no método construtor.
Abaixo temos um exemplo onde vemos o médoto construtor sendo iniciado:

```
#!/usr/bin/python3


class NomeClasse:


   def __init__(self):
       self.teste = print ("Executando o método construtor")



t = NomeClasse()
```

Anotações:

Na programação, antes de efetivamente termos um objeto, precisamos criar sua estrutura: descrever e detalhar o máximo que aquele objeto fará, suas características e etc.

Imagine que criaremos um sistema para levantar o inventário do nosso parque de servidores. Para isso precisamos trazer informações como: CPU, Memória, Espaço em Disco, etc.

Na OO, essa estrutura que estamos falando que descreve como será objeto é chamada de **classe**.

```
#!/usr/bin/python3

class Servidor:
    pass
```

Para criarmos um objeto a partir dessa classe, fazemos da mesma forma que variáveis:

```
#!/usr/bin/python3

class Servidor:
    pass

dns = Servidor()
```

Usualmente armazenamos esse objeto numa variável para que possamos trabalhar com o mesmo objeto no resto do programa.

Você pode criar vários objetos a partir da mesma classe:

```
#!/usr/bin/python

class Servidor:
    pass

dns = Servidor()
web = Servidor()
database = Servidor()
ldap = Servidor()
```

A classe que montamos até aqui possui apenas um nome: Servidor. Até agora essa classe não tem estrutura alguma, já que o bloco da classe está vazio. Vamos começar a melhorar isso.

Classes, propriedades e métodos

Propriedades e métodos

Uma classe é composta de **propriedades** e **métodos** que, juntos, criam a funcionalidade de um objeto.

- ✔ Atributos e métodos possuem o que chamamos de visibilidade;
- ✔ Métodos podem ou não receber parâmetros, que, por sua vez, 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:
    cpu = None
    memoria = None
    disco = None
```

Para criarmos uma propriedade numa classe, basta, logo no começo do bloco, enumerá-las (uma por linha por padrão), como se fossem variáveis.

Anotações:

Classes, propriedades e métodos

Criando classe com atributos

```
class NomedaClasse:

    atributo = “valor”

    def metodo(self):

        pass
```


Anotações:

Classes, propriedades e métodos

Acessando Métodos e Atributos

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

```
objeto = Classe()
```

- Dentro da classe usamos **self** para acessar os recursos.

A sintaxe para criação de propriedades 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 %s, Memoria: %s, Disco %s GB "%(dns.cpu,dns.memoria,dns.disco))
```

Nas linhas destacadas, estamos atribuindo valores aos atributos. De tal modo que podemos resgatar o valor desses atributos posteriormente. A saída deste código é: "O servidor tem as seguintes configuracoes:
CPU 2, Memoria: 2048, Disco 50 GB ".
Como você também pode imaginar, os valores atribuídos à essas propriedades podem ser de qualquer tipo, igual fazemos com as variáveis normais. Podemos atribuir um array, uma string, um número, etc.

Classes, propriedades e métodos

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, também devemos colocar nessa estrutura as ações que aquele objeto vai executar, ou como passaremos a chamar daqui para frente, seus métodos.

Essas ações, diferentemente das propriedades, precisam ser declaradas e implementadas. Ou seja, precisamos enumerar todas as ações do nosso objeto e codificar o que cada uma vai fazer *(existe 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 configuracoes:
       CPU %s, Memoria: %s, Disco %s GB "
       %(dns.cpu,dns.memoria,dns.disco))
```

Os métodos são chamados como as funções, porém, precisamo, do nome do objeto na frente seguido por ponto. O primeiro parâmetro **self** é obrigatório, pois só com ele o python consegue diferenciar os métodos das funções .

Herança e Polimorfismo

Objetivos da Aula

- ✔ Conhecer o conceito de Herança;
- ✔ Conhecer Herança Múltipla;
- ✔ Trabalhar com Polimorfismo.

Herança e Polimorfismo

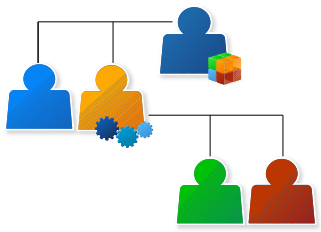
Nesta aula iremos apresentar o conceito de Herança e Polimorfismo, criar exemplos e colocar em prática os conceitos criando uma aplicação.

Anotações:

Herança e Polimorfismo

Um dos conceitos fundamentais da orientação a objetos é a herança de classes.

Através dela é que podemos implementar uma série de outros conceitos, bem como ver com mais clareza a reusabilidade de código.



A herança permite que uma classe estenda outra. Ao fazer isso, **a classe filha vai herdar todos os atributos e métodos da classe pai**. Ela pode, então, tanto possuir novos métodos e atributos como reescrever métodos já existentes.

Entendendo Herança

Herança é um dos recursos mais interessantes da OO. Mas não se engane, o que falaremos aqui sobre classes mães e filhas não tem nada a ver com o que acontece na biologia.

Imagine que você está modelando seu sistema e observa que alguns objetos possuem comportamentos parecidos ou até mesmo iguais. Uma das coisas a se fazer para que a manutenção nesse código parecido não seja muito complicada, seria centralizar esse código num único lugar, fazendo com que a manutenção seja feita, ao invés de em vários arquivos, num único ponto.

Este é o conceito de generalização. Imagine que você criou um objeto Professor e um objeto Aluno. Se você modelou com o básico de informações, você notará que ambos possuem certas características e comportamentos iguais. Dito isto, podemos pegar o que for igual e colocar tudo numa nova classe, digamos Pessoa.

Para que o funcionamento do sistema permaneça o mesmo, em ambas as classes, Aluno e Professor, precisamos colocar uma nova notação, dizendo que ambas tem como base a classe Pessoa.

Herança e Polimorfismo

A Herança é definida colocando-se o nome da classe pai como parâmetro da classe filha.

```
class NomedaClasse(ClassePai):
```

- Todos os métodos e atributos agora existem na classe filha;
- O Python possui herança múltipla;
- É possível estender de uma classe filha.

Entendendo Herança

Imagine que você modelou um servidor cloud e um servidor físico como objetos do seu sistema. Ambos possuem diversas características e ações em comum, que podem ser abstraídas para uma classe mãe (ou pai, como preferir), centralizando o código comum num único ponto de alteração, facilitando assim a manutenibilidade do código, organização, etc.

Crie um arquivo chamado “Servidor.py” e coloque o seguinte código:

```
#!/usr/bin/python3

class Servidor:

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

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

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

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

Crie um arquivo chamado “Cloud.py” e coloque o seguinte código:

```
from servidor import Servidor

class Cloud(Servidor):
    def __init__(self):
        self.memoria = 512
        self.cpu = 1
        self.disco = 50


dns = Cloud()

print ("Memoria Inicial",dns.memoria)
print ("Disco Inicial ",dns.disco)
print ("CPU Inicial ",dns.cpu)

dns.contratarDisco(50)

print ("Disco Total ",dns.disco)
```


Crie um terceiro arquivo, chamado Fisico.py

```
#!/usr/bin/python

from servidor import Servidor

class Fisico(Servidor):
    def __init__(self):
        self.memoria = 4096
        self.cpu = 4
        self.disco = 1024

dns = Fisico()

print ("Memoria Inicial",dns.memoria)
print ("Disco Inicial ",dns.disco)
print ("CPU Inicial ",dns.cpu)

dns.contratarDisco(1024)

print ("Disco Total ",dns.disco)
```

Note que agora temos 3 classes: uma classe Pai chamada Servidor e duas Classes Filhas chamadas Cloud e Físico. Por mais que os servidores sejam diferentes, eles tem aspectos em comum:

- Disco
- Memória
- CPU

Ambos podem ser contratados adicionais, porém de maneiras diferentes.

Exemplo:

Em um servidor Cloud, nós podemos contratar quantos discos quisermos, pois é simplesmente um volume entregue por um Storage, porém no servidor Físico, ficamos dependentes da quantidade de slots que temos nas máquinas.

Exemplo: Se uma máquina tem slot pra 4 discos, não conseguimos colocar outro disco. É preciso substituir os já existentes e, além do mais, precisamos também limitar o disco por tamanho, já que atualmente não temos como colocar um disco de 10Gb em um servidor físico.

Para resolver este problema, vamos entrar agora no conceito de **Polimorfismo**.

Herança e Polimorfismo

Polimorfismo: significa muitas formas. Com esses recursos podemos ter métodos com nomes iguais, porém fazendo coisas diferentes.

Como por exemplo caso de contratação de disco adicional pois tanto servidores Cloud quanto servidores Físicos precisam de espaço, porém, eles agem de formas diferentes.

Anotações:

Herança e Polimorfismo

Vamos adicionar um novo atributo chamado **Slots** ao servidor Físico, que vai armazenar a quantidade de **slots** da máquina.

Faremos uma regra para disponibilizar apenas discos de 500Gb e 1 Tb.

```
class Fisico(Servidor):
   def __init__(self):
      self.total_slots = 4
      self.slots_ocupados = 1
```


Entendendo Polimorfismo

Agora alteramos a classe Físico, que tem as características e atributos específicos dela. Assim, aplicamos perfeitamente o conceito de polimorfismo. Poderíamos, por exemplo, criar agora uma nova classe Servidor Mainframe e estendê-la da classe servidor.

A Classe Servidor também serviu para nos mostrar o conceito de classe abstrata, já que ela nunca será instanciada diretamente, só a usamos como base para criar outros servidores.

Na próxima página segue o código fonte da classe Físico já alterada.

Anotações:


```
#!/usr/bin/python3

from servidor import Servidor

class Fisico(Servidor):
    def __init__(self):
        self.memoria = 4096
        self.cpu = 4
        self.disco = 1024
        self.total_slots = 4
        self.slots_ocupados = 1

    def contratarDisco(self,disco):
        if disco == 500 or disco == 1024:
            if self.total_slots > self.slots_ocupados:
                self.slots_ocupados += 1
                self.disco += disco
            else:
                print ("Voce nao tem mais slots disponiveis")
                print ("Total de Slots ",self.total_slots
                print ("Total Ocupados ",self.slots_ocupados)
        else:
            print ("Tamanho de disco nao valido")

dns = Fisico()

print ("Memoria Inicial",dns.memoria)
print ("Disco Inicial ",dns.disco)
print ("CPU Inicial ",dns.cpu)

dns.contratarDisco(1024)
dns.contratarDisco(1024)
dns.contratarDisco(1024)
dns.contratarDisco(1024)
print ("Disco Total ",dns.disco)
```

Anotações:

Herança e Polimorfismo

Herança Múltipla

A **herança múltipla** é utilizada quando uma classe precisa herdar os atributos e características de mais uma classe.

Imagine que nos dois tipos de servidores que temos definidos até agora (cloud e físicos). Ambos precisam de acesso, porém, na classe Cloud, pode-se ter um tipo de acesso a mais, que seria o **VNC** e, no caso do Físco, o acesso seria o **IPMI**.

Vamos criar um novo arquivo chamado **"acesso.py"** e adicionar as seguintes linhas para conseguir fazer o método de acesso:

```
#!/usr/bin/python
# acesso.py
class Acesso:
    def __init__(self):
        pass

    def acessarConsole(self):
        print ("Acesso por VNC")
```

Agora edite o arquivo **"cloud.py"** e adicione as seguintes linhas:

```
from servidor import Servidor
from acesso import Acesso

class Cloud(Servidor,Acesso):
    def __init__(self):
        self.memoria = 512
        self.cpu = 1
        self.disco = 50


dns = Cloud()
dns.acessarConsole()
```

Vamos usar neste exemplo o conceito de Polimorfismo e fazer com que a classe Físico tenha um método com o mesmo nome, porém execute algo diferente.

Para isto, edite o arquivo "**fisico.py"** e adicione as seguintes linhas:

```
#!/usr/bin/python3


from servidor import Servidor
from acesso import Acesso

class Fisico(Servidor,Acesso):
    def __init__(self):
        self.memoria = 4096
        self.cpu = 4
        self.disco = 1024
        self.total_slots = 4
        self.slots_ocupados = 1

    def contratarDisco(self,disco):
        if disco == 500 or disco == 1024:
            if self.total_slots > self.slots_ocupados:
                self.slots_ocupados += 1
                self.disco += disco
            else:
                print ("Voce nao tem mais slots disponiveis")
                print ("Total de Slots ",self.total_slots)
                print ("Total Ocupados ",self.slots_ocupados)
        else:
            print ("Tamanho de disco nao valido")

    def acessarConsole(self):
        print ("Acesso por IPMI")

dns = Fisico()
dns.acessarConsole()
```

Anotações:

Erros e Exceções

Objetivos da Aula

- ✔ Compreender o que é são Erros e Exceções
- ✔ Trabalhar com Try / Except.

Erros e Exceções

Nesta aula iremos apresentar o conceito de erros e exceções, ver quais as diferenças, como o Python trabalha com cada tipo e vamos implementar o tratamento de erro em nossa aplicação.

Anotações:

Erros e Exceções

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

Os **erros de sintaxe** (ou erros de análise) são os mais comum quando estamos aprendendo, são aqueles que ao executar o script o interpretador exibe uma linha indicando “SyntaxError” e uma “seta” apontando onde o erro foi encontrado.

```
if 'Mariana' == nome
                    ^

SyntaxError: invalid syntax
```


Anotações:

Erros e Exceções

Exceções

As **exceções** são aquelas que a aplicação irá gerar mesmo que a sintaxe esteja correta, porém está ocorrendo algum erro ao tentar executá-la.

Estas **exceções** porem ser ou fatais, ou seja, mesmo que ocorra alguma exceção em sua aplicação ela pode continuar funcionando, porém sempre devemos tratá-las.

Anotações:

Erros e Exceções

As exceções não são tratadas automaticamente, porém exibem mensagens de erro que podem nos ajudar, como por exemplo:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>

TypeError: Can't convert 'int' object to str implicitly
```


Exceções

Um exemplo que irá gerar uma exceção é tentar fazer uma comparação com uma variavel porém não definirmos o que está variavel será, como por exemplo:

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

Quando execurtarmos 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 o que ocorreu, ou seja, o tipo da exceção é “NameError” e ela indica que a variavel “nome” não foi definida.

Erros e Exceções

Tratando exceções:

Existem diversos tipos de exceções e nós conseguimos manupilar para que, caso ocorra aquela exceção um trecho de código seja executado e o nosso script não 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 Exception é ignorado e a execução é concluída;
3 - Caso ocorra alguma exceção durante a execução do Try, o resto do código é ignorado e caso o Exception Type corresponda com o que ocorreu, o Except é executado;
4 - Caso ocorrer erro e o Exception Type não for correspondente, então a execução do script irá parar.

O principal objetivo de colocarmos o nosso código dentro de um try / except é fazer com que o código não pare de executar caso ocorra algum erro.

Anotações:

Erros e Exceções

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:

Erros e Exceções

Finally

O try/except possui uma opção que é destinada a executar independente se ocorreu exception ou não.

Para implementar o finally, podemos adicionar abaixo da Exception:

```
finally:
        comandos
```


O **"finally"** será executado independente se ocorreu alguma exceção no seu código, ou seja, independente do 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 por exemplo, pois independente se ocorreu erro ou executou com sucesso ele irá gravar o log e assim teremos um controle dos horários em que foi executado ou que ocorreu erro.

Raise

Objetivos da Aula

- ✔ Compreender o que são Raise Exception;
- ✔ Trabalhar com Raise.

Raise

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

Anotações:

Raise

No Python nós podemos criar as nossas próprias exceções e fazer com que a aplicação ou script trate determinada situação e não pare a execução.

A declaração **raise** nos permite criar estas exceções, ou seja, eu estou forçando 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

O raise aceita qualquer tipo de Type Error (semelhante ao Except). Para que ele funcione da forma que desejamos, o mesmo Type Error definido no raise deverá ser definido no Except.

Normalmente quando vamos utilizar o raise, colocamos ele dentro de um bloco de código de if/else. Pois caso o retorno não seja o esperado, ele apenas irá forçar uma exceção e o meu script irá continuar rodando.

Anotações:

Raise

Sintaxe

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

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


Anotações:

Raise

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)
```


Anotações:

Raise

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:
    raise
```


Anotações:

Exception Types

Objetivos da Aula

- ✔ Conhecer os Exception Types;
- ✔ Trabalhar com Exception Types.

Exception Types

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

Anotações:

Exception Types

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 pois assim conseguimos ter um maior controle da nossa aplicação.

Anotações:

Exception Types

Trabalhando com Exception Types

Para definir uma exceção genérica, podemos fazer da seguinte forma:

```
try:
    x = 5
    y = 0
    z = x/y
except Exception as e:
    print ('Erro: %s' %e)
```


Anotações:

Exception Types

Trabalhando com Exception Types

Podemos utilizar Exception Types especificas, como por exemplo:

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

TypeError: quando o tipo do objeto é inapropriado.

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

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

Anotações:

Exception Types

NameError

```
#!/usr/bin/python3

try:
    print ('A Linguagem é: %s' %linguagem)

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


Anotações:

Exception Types

TypeError

```
#!/usr/bin/python3

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

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


Anotações:

Exception Types

IndexError

```
#!/usr/bin/python3

try:
    linguagem = ['python']
    print ('A Linguagem e: %s' %linguagem[2])

except IndexError as e:
    print ('Erro: %s' %e)
```


Anotações:

Exception Types

KeyError

```
#!/usr/bin/python3

try:
    linguagem = {'curso':'fundamentals'}
    print ('A Linguagem é: %s' %linguagem["nome"])

except KeyError as e:
    print ('Erro: %s'%e)
```


Anotações:

Exception Types

Definindo mais de um Exception Type:

Podemos definir mais de um Exception Type, para isto basta inserir no código da seguinte forma:

```
except NameError as e:
    print ('Erro: %s' %e)
except KeyError as e:
    print ('Erro: %s' %e)
```


Anotações:

Modulos e Instaladores de Pacotes

Objetivos da Aula

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

Instaladores de Pacotes

Nesta aula iremos apresentar o conceito de módulos, ver como podemos criar um módulo, instalar módulos externos via easy_install e via pip.

Anotações:

Modulos e Instaladores de Pacotes

Um módulo nada mais é do que um arquivo com uma série de funções que podem ser utilizados em outra parte do código.

No Python podemos utilizar alguns módulos que já estão prontos ou podemos criar os nossos próprios módulos.

É extremamente importante fazermos a modularização de aplicações pois desta forma conseguimos reutilizar os módulos em várias partes da aplicação e porque é mais fácil de controlar e manter.

Anotações:

Modulos e Instaladores de Pacotes

Criando um módulo

Vamos criar um arquivo com o nome **module.py** e definir duas funções:

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

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


Anotações:

Modulos e Instaladores de Pacotes

Utilizando o módulo

Após a criação do módulo, vamos entender como podemos usá-lo em nossa aplicação.

O primeiro passo é importar o módulo:

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


Anotações:

Modulos e Instaladores de Pacotes

Utilizando o módulo

Vendo informações sobre o módulo:

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

Chamando o módulo:

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


Anotações:

Modulos e Instaladores de Pacotes

Utilizando o módulo

Podemos importar um módulo específico:

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


Utilizando Módulo

Os módulos em python são separados por diretórios dentro do seu projeto.

É necessário que exista um arquivo __init__.py dentro dos diretórios dos pacotes para que o python entenda que os arquivos podem ser importados como um módulo.

Um exemplo de estrutura de aplicação poderia ser assim:

Projeto/
/model
/ __init__.py
/ model.py

Para fazermos o import do model.py seria necessário fazer desta forma:

```
from model.model import nome_da_funcao
```

Modulos e Instaladores de Pacotes

Instalando Módulos Externos

O Python tem um repositório oficial chamado PuPI (Python Package Index), onde são disponibilizados módulos prontos que podem ser usados em sua aplicação.

```
https://pypi.python.org/pypi
```

O PyPI (Python Package Index) tem uma lista enorme de pacotes que podemos utilizar. Atualmente existem mais de 100 mil pacotes disponíveis.

Nele 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

E módulos para trabalhar com testes também:

Teste Funcional
Selenium: https://pypi.python.org/pypi/selenium

Teste Unitário
Pytest: https://pypi.python.org/pypi/pytest

Modulos e Instaladores de Pacotes

Instalando Módulos Externos

Uma maneira de instalar um módulo externo é utilizar o pacote python-setuptools que disponibilizará o comando easy_install usado para a instalação dos módulos .

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

O comando **easy_install** vem do pacote setuptools que foi desenvolvido em 2004 como a primeira ferramenta para instalação de pacotes do python. Ele logo foi notado devido a facilidade em conectar no repositório PyPI, resolvendo as dependências.

Antigamente a instalação era feita baixando o código fonte do pacote e utilizando os seguintes comandos para efetuar a instalação:

```
python setup.py install
```

Anotações:

Modulos e Instaladores de Pacotes

Instalando Módulos Externos

Outra maneira de instalar um módulo externo é utilizar o pacote **python-pip** que disponibilizará o comando **pip** usado para a instalação dos módulos .

```
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 que fornece o comando easy_install, a instalação do pip pode ser feita por ele.

O **pip** foi desenvolvido em 2008 como alternativa ao easy_install e se tornou bem popular devido a facilidade de uso. Além de ter mais opções que o comando easy_install, o **pip** é mais rápido pois não faz a instalação de um pacote a partir do zero, ele provém apenas os metadados com os códigos fonte essenciais para o funcionamento do módulo.

Anotações:

Módulos Nativos

Objetivos da Aula

- ✔ Compreender o que são Módulos Nativos;
- ✔ Conhece os módulos os, sys, datetime, json e csv.

Módulos Nativos

Nesta aula iremos entender o que são módulos Nativos, vamos conhecer alguns destes módulos e utilizar e nossas aplicações.

Anotações:

Módulos Nativos

Sintaxe

Em Python podemos criar novos módulos, utilizar módulos externos (que já estão prontos) ou usar os módulos nativos.

Os módulos nativos são aqueles que já estão embutidos na linguagem, ou seja, não é necessário instalar basta faze o “import” e utiliza-los conforme a sintaxe do módulo.

Anotações:

Módulos Nativos

Vamos aprender um pouco mais sobre os seguintes módulos:

Os: Permite usar funcionalidades do sistema operacional

Sys: Podemos acessar parâmetros e funções específicas do Python.

Datetime: Módulo que nos trás alguns tipos de data e hora.

Json: Codificar e decodificar no formato JSON

Csv: Ler e escreer arquivos no forrmato CSV (comum em planilhas)

Anotações:

Módulos Nativos

Módulo “os”

Sempre que for necessário criar um script que precise executar algum comando no linux, posso utilizar o módulo os.

Como por exemplo criar diretórios, ver informações do sistema listar arquivos, ver informações de usuário, entre outras coisas.

Anotações:

Módulos Nativos

Algumas funcionalidades:

```
import os
#ver qual o usuário logado
print (os.getlogin())
#listar o conteúdo de um diretorio
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

Basicamente quando vamos executar alguma coisa no sistema operacional Linux utilizamos o módulo os, ele é o módulo do Python que fará as chamadas de sistema e executará o comando.

Podemos executar o comando utilizando “os.system('comando') porém o mais indicado é que veja na documentação a melhor forma de fazer a execução, pois pode existir alguma forma específica para fazer o que deseja.

Para maiores informações sobre o módulo os, podemos verificar a documentação oficial:

https://docs.python.org/3/library/os.html#module-os

Anotações:

Módulos Nativos

Módulo “sys”

O Módulo Sys nos fornece acesso a algumas variáveis ou funções que são executadas pelo Python.

Com ele podemos trabalhar com o interpretador, ou seja, conseguimos ver a versão do Python, em que sistema ele está sendo executado, passar parametros, entre outras coisas.

Anotações:

Módulos Nativos

Algumas funcionalidades:

```
import sys
#ver em qual plataforma está executando o script
print (sys.platform)
#ver os módulos que estão sendo executados pelo Python
print (sys.builtin_module_names)
#passar argumentos
print (sys.argv)
#finalizar o script
sys.exit()
```


Modulo Sys

O módulo sys nos tratá informações sobre o interpretador (python). Uma funcionalidade muito interessante é a passagem de argumentos, como por exemplo:

```
#!/usr/bin/python3
import sys

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

O argv irá retornar os argumentos como uma lista, então ao passarmos 3 argumentos, para acessa-los seria: sys.argv[1], sys.argv[2] e sys.argv[3], o sys.argv[0], corresponde ao nome do Script.
Para maiores informações podemos ver a documentação ificial:
https://docs.python.org/3/library/sys.html#module-sys

Módulos Nativos

Módulo “Datetime”

O Módulo Datetime nos fornece formas de manipular data e hora, de maneira simples e até mais complexas.

É muito importante sabermos trabalhar com data e hora em Python e a linguagem oferece um módulo nativo para isto.

Anotações:

Módulos Nativos

Algumas funcionalidades:

```
import datetime
#mostrar 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

O Modulo Datetime permite a manipulação de data e horário, podemos fazer a expiraçao de token com validade de 1h:

```
#!/usr/bin/python3
from datetime import datetime
acesso = datetime(2016,01,22,00,00,00)
atual = datetime(2016,01,22,01,01,00)

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

Neste caso, temos um exemplo prático para calcular o tempo de expiração de um token de acesso. Fizemos a subtração de 2 objetos Datetime, e essa subtração também nos retorna um outro objeto Datetime com a diferença das datas. Nele podemos usar o método total_seconds, que vai converter o resultado em segundos e compara com 3600 segundos que é equivalente a uma hora.

Então, se o total de segundos for maior que 3600 o token é expirado.

Podemos ver maiores informações sobre o módulo datetime na documentação oficial: https://docs.python.org/3/library/datetime.html#module-datetime

Módulos Nativos

Módulo “JSON”

O Módulo Json permite a alteração de determinados caracteres para o formato JSON e vice-versa.

Este é um módulo muito importante pois quando trabalhamos com API's normalmente usamos este formato e temos que fazer a conversão para conseguir utilizar os dados em nossa aplicação.

Anotações:

Módulos Nativos

Algumas funcionalidades:

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


Modulo Json

Quando fazemos integração com outras ferramentas, é normal utilizarmos API para o envio e recebimento de alguns dados. É comum que estes dados venham no formado JSON, ou seja, como um dicionário e as vezes a nossa aplicação não se encaixa com este tipo de informação e precisamos converter este JSON para string e vice-versa.

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

print type((json.loads('{"nome":"python",
                         "modulo":"fundamentals"}')))
```

No código acima podemos ver o tipo de cada um.
Podemos ver maiores informações sobre o módulo json na documentação oficial:
https://docs.python.org/3/library/json.html#module-json

Módulos Nativos

Módulo “CSV”

O módulo CSV permite que nosso script consiga trabalhar com planilhas, é possível fazer a importação e expotação.

Podemos ler e escrever arquivos deste tipo e usar os dados em nossos scripts.

Anotações:

Módulos Nativos

Ver as informações em um arquivo:

```
import csv

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


Modulo Json

Quando fazemos integração com outras ferramentas, é normal utilizarmos API para o envio e recebimento de alguns dados. É comum que estes dados venham no formado JSON, ou seja, como um dicionário e as vezes a nossa aplicação não se encaixa com este tipo de informação e precisamos converter este JSON para string e vice-versa.

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

print type((json.loads('{"nome":"python",
                         "modulo":"fundamentals"}')))
```

No código acima podemos ver o tipo de cada um.
Podemos ver maiores informações sobre o módulo json na documentação oficial:
https://docs.python.org/3/library/json.html#module-json

Módulos Nativos

Gravar as informações em um arquivo:

```
import csv

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


Modulo Json

Quando fazemos integração com outras ferramentas, é normal utilizarmos API para o envio e recebimento de alguns dados. É comum que estes dados venham no formado JSON, ou seja, como um dicionário e as vezes a nossa aplicação não se encaixa com este tipo de informação e precisamos converter este JSON para string e vice-versa.

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

print type((json.loads('{"nome":"python",
                         "modulo":"fundamentals"}')))
```

No código acima podemos ver o tipo de cada um.
Podemos ver maiores informações sobre o módulo json na documentação oficial:
https://docs.python.org/3/library/json.html#module-json

SQL

Objetivos da Aula

- ✔ Entender o que são Banco de Dados;
- ✔ Fazer uma introdução à linguagem SQL.

SQL

Nesta aula iremos apresentar alguns conceitose características da linguagem SQL e mostrar como podemos trabalhar com ela.

Anotações:

SQL

Introdução ao SQL

Um **banco de dados** é um local destinado a guardar dados 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 é utilizada para interagir com um SGBD (Sistema Gerenciador de Banco de Dados) e executar tarefas como: inserir registros, gerenciar usuários, criar consultas, entre outras.

Introdução ao SQL

Temos que ter de forma bem clara o que é um banco de dados e o porquê desse nome.

Por que você acha que um Banco de Dados armazena somente dados como textos, números, datas, etc, sem um contexto em específico?

Por exemplo, o que quer dizer 09/06/2017 ?

09/06/2017 é somente um dado, pois não tem nenhum significado. Porém, se mudarmos o contexto para:

O Cliente João foi cadastrado na data de 09/06/2017.

Desta maneira, demos um contexto para o dado extraído e conseguimos, a partir deste dado, trazer uma informação e trabalhar a partir dela.

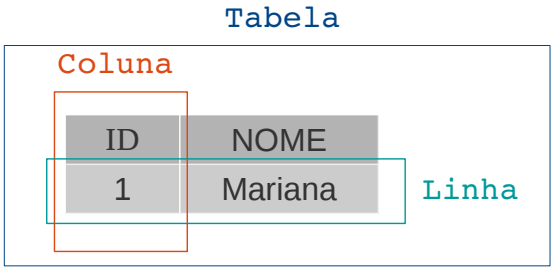
Sendo assim, utilizamos o banco de dados para conservar os dados até que possamos recuperá-los em um determinado contexto e gerar informações que possam trazer valor para o negócio.

SQL

Introdução ao SQL

Na computação utilizamos ferramentas **SGBD (Sistemas Gerenciadores de Bancos de Dados)** para fazer o armazenamento e o manuseio desses dados.

Nos SGBD's, os dados são organizados em **tabelas, colunas, linhas**.

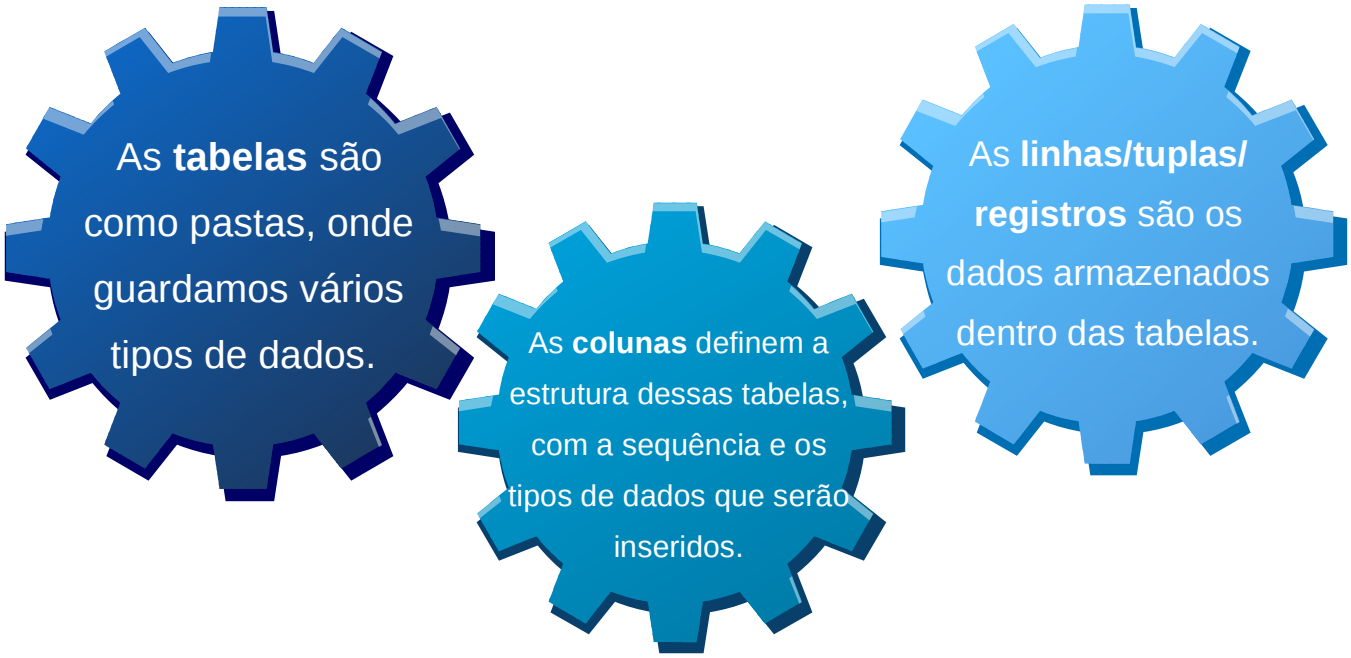


Durante a maior parte do tempo como um desenvolvedor, você irá criar ou encontrar aplicações que utilizam alguma forma de armazenamento de dados. Estes dados podem estar sendo armazenados em simples arquivos no sistema de arquivos de uma máquina ou podem estar sendo armazenados em estruturas mais complexas,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

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

Anotações:

SQL



Anotações:

SQL

Quando utilizamos SGBD existem vários tipos de comandos separados por categoria. Os principais são:

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

DML: Data Manipulation Language, com ele podemos manipular os dados, com comandos como: INSERT, UPDATE e DELETE.

DQL: Data Query Language, com ele fazemos a consulta dos dados, com comandos como: SELECT, SHOW.

Anotações:

PostgreSQL

Objetivos da Aula

- ✔ Conhecer o PostgreSQL;
- ✔ Trabalhar com PostgreSQL.

PostgreSQL

Nesta aula iremos apresentar o PostgreSQL e mostrar como podemos trabalhar com ele, criando tabelas, inserindo dados e fazendo as consultas.

Anotações:

PostgreSQL

Introdução a PostgreSQL

O **PostgreSQL** é um Sistema Gerenciador de Banco de Dados (SGBD) de código aberto.
Ele é otimizado para aplicações mais complexas, ou seja, aplicações que trabalham informações mais críticas ou grandes volumes de dados.
Podemos utilizar como exemplo um comércio eletrônico de porte médio, o PostgreSQL é mais indicado pois é 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. Ele é 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, tem vários tipos de índices para diversos tipos de aplicações.

Ele é utilizado em larga escala por empresas como Skype, Caixa Econômica Federal e Datasus.

O Python trabalha muito bem com o PostgreSQL e, assim como o MySQL, possui módulos para trabalhar com ele.

Anotações:

PostgreSQL

Trabalhando com PostgreSQL

Quando ainda não temos nenhum usuário criado, o sistema disponibiliza um usuário para podermos gerenciar o banco.

1 Logar com o usuário root

```
$ su - root
```

2 Logar com o usuário postgres e acessar o banco

```
# su - postgres
# psql
```


Anotações:

PostgreSQL

Trabalhando com PostgreSQL

1 Criar um usuário para acesso a aplicação

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

2 Criar um banco de dados para aplicação

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

3 Deslogar com o usuário postgres

```
=# \q
```

4 Logar no banco 'projeto' com o usuario 'admin'

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


Anotações:

PostgreSQL

Trabalhando com PostgreSQL

1	Verificando as tabelas criadas `=# \dt`
2	Criar uma tabela para inserir scripts `=#CREATE TABLE scripts(id SERIAL, nome VARCHAR(50), conteudo TEXT);`
3	Ver informações sobre a tabela `=# \d scripts`
4	Ver o conteúdo da tabela `=# SELECT * FFROM scripts;`

Anotações:

PostgreSQL

Trabalhando com PostgreSQL

1 Inserir dados na tabela scripts

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

2 Fazer o update

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

3 Deletar um registro

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

4 Apagar todos os registros

```
=# TRUNCATE scripts;
```


Anotações:

PostgreSQL

Trabalhando com PostgreSQL

1 Contar dados especificos

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

2 Consultar dados por um intervalo

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

3 Consultar dados por parte do campo

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

4 Ordenar dados de forma decrescente

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


Anotações:

MySQL

Objetivos da Aula

- ✔ Conhecer o MySQL;
- ✔ Trabalhar com MySQL.

MySQL

Nesta aula iremos apresentar o MySQL e mostrar como podemos trabalhar com ele, criando tabelas, inserindo dados e fazendo as consultas.

Anotações:

MySQL

Introdução ao MySQL

O **MySQL** é um dos SGBD's mais populares que existem. É amplamente utilizado para serviços de hospedagem de sites. Este banco é focado na agilidade, então é adequado se a aplicação necessita de retornos rápidos e não envolve operações mais complexas.

Podemos utilizar como exemplo o armazenamento do conteúdo de um site ou um fórum.

Anotações:

MySQL

Trabalhando com MySQL

1 Logar no MySQL
```
$ mysql -u root -p
```

2 Verificar os bancos existentes
```
> SHOW DATABASES;
```

3 Criar um banco de dados
```
> CREATE DATABASE projetos;
```

4 Acessar a base
```
> USE projetos;
```


Anotações:

MySQL

Trabalhando com MySQL

5 Inserir um usuario para administrar o banco

```
> GRANT ALL PRIVILEGES on projetos.* to admin@'localhost'
  IDENTIFIED BY '4linux' WITH GRANT OPTION;
```

6 Atualizar os acessos

```
> FLUSH PRIVILEGES;
```

7 Sair do MySQL

```
> EXIT;
```

6 Acesse com o novo usuario

```
# mysql -u admin -p
```


Anotações:

MySQL

Trabalhando com MySQL

1 Verificando as tabelas criadas

```
> SHOW TABLES;
```

2 Criar uma tabela para inserir clientes

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

3 Ver informações sobre a tabela

```
> DESCRIBE clientes;
```

4 Ver o conteúdo da tabela

```
> SELECT * FROM clientes;
```


Anotações:

MySQL

Trabalhando com MySQL

1 Inserir dados na tabela scripts

```
> INSERT INTO clientes(nome, endereco) VALUES ('Mariana', 'Rua
  Vergueiro');
```

2 Fazer o update

```
> UPDATE clientes SET endereco='Paulista' WHERE nome = 'Mariana';
```

3 Deletar um registro

```
> DELETE FROM clientes WHERE id = 1;
```

4 Apagar todos os registros

```
> TRUNCATE clientes;
```


Anotações:

MySQL

Trabalhando com MySQL

1. Vamos criar mais uma tabela chamada projetos

```
> CREATE TABLE projetos(id INTEGER PRIMARY KEY AUTO_INCREMENT NOT NULL, nome VARCHAR(255) NOT NULL, cliente VARCHAR(255));
```

2. Inserir 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 com PostgreSQL

Objetivos da Aula

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

Python com PostgreSQL

Nesta aula iremos apresentar o módulo de conexão com o PostgreSQL e mostrar como podemos trabalhar com ele, inserindo dados e fazendo as consultas.

Anotações:

Python com PostgreSQL

Instalação do módulo

A primeira coisa que precisamos fazer é instalar o módulo do Python que dá suporte ao PostgreSQL.

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

No caso deste módulo em específico, instalamos pelo **apt-get** porque o **postgres** precisa de outras bibliotecas do sistema operacional que o **pip** não consegue resolver, já que ele foi feito para resolver dependências somente do python e não de Linux.

Anotações:

Python com PostgreSQL

Criando conexão

```
import psycopg2

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

Na primeira linha é importado o módulo do postgres e, logo em seguida, abrimos uma conexão com o banco e vamos utilizá-la para executar os comandos dentro do banco.

Anotações:

Python com PostgreSQL

Executar operações

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

A variável **cur** recebe o cursor do banco de dados a partir da conexão. É com este cursor que vamos efetuar as operações CRUD (Create, Retrieve, Update, Delete).

Anotações:

Python com PostgreSQL

Sintaxe

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

Nesta parte utilizamos a variável **cur** que recebeu o cursor para navegar no banco e executamos um comando **insert**.

Logo abaixo, há uma linha utilizando a variável **con** para fazer o **commit**. Enquanto não for efetuado o **commit**, não será gravado nenhum dado no banco.

Segue abaixo 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: %s"%e)
    print ("Fazendo rollback")
    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 .

Python com PostgreSQL

Sintaxe

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

O execute também é utilizado para atualizar e deletar registros ainda dependendo do **commit** para efetuar a modificação no banco.

Segue abaixo um 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: %s"%e)
    print ("Fazendo rollback")
    con.rollback()

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

Da mesma forma que usamos a DML Insert, o update e o delete também ficarão gravados na memória até que seja efetuado o commit. Se o python não encontrar este comando, não haverá a atualização desses dados ao final da execução do script.

Python com PostgreSQL

Sintaxe

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

O **cur.rollback()** é utilizado para voltar a última transação caso ocorra algum erro. O **cur.close()** e **con.close()** são utilizados para finalizar a conexão com o banco de dados.

Segue abaixo um 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: %s"%e)
    print ("Fazendo rollback")
    con.rollback()

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

Neste exemplo foi digitado errado o nome de uma tabela.

Python com PostgreSQL

Sintaxe

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

Ao realizar uma busca, podemos retornar só o primeiro registro do banco utilizando o **fetchone()** ou todos eles utilizando o **fetchall()**.

Abaixo temos um exemplo buscando o primeiro registro de uma tabela e 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: %s"%e)
    print ("Fazendo rollback")

finally:
    print ("Finalizando conexao 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 com MySQL

Objetivos da Aula

- ✔ Conhecer o módulo para conexão;
- ✔ Fazer inserção, atualização, exclusão e consulta no banco de dados.

Python com MySQL

Nesta aula iremos conhecer o módulo de conexão e mostrar como podemos trabalhar com ele, criando tabelas, inserindo dados e fazendo as consultas através do Python.

Anotações:

Python com MySQL

Instalação do Módulo

A primeira coisa que precisamos fazer é instalar módulo do python que dá suporte ao MySQL.

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

No caso deste módulo em específico, instalamos pelo **apt-get** porque o **MySQL** precisa de outras bibliotecas do sistema operacional que o **pip** não consegue resolver, já que ele foi feito para resolver dependências somente do python e não de Linux.

Anotações:

Python com MySQL

Criando conexão

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")
```


O Python não tem uma função nativa para fazer a conexão com o MySQL. Por isso, instalamos o módulo MySQLdb, ele nos fornece todos os comandos necessário para interagir com o banco de dados.

Na aula de funções foi explicado o conceito de tratamento de exceções, que são 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 consiga efetuar 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 elas não sejam fechadas após o seu uso e o seu script tiver muitas conexões de banco, pode ocasionar o erro: MySQL server has gone away, causando a indisponibilidade do banco de dados.

Anotações:

Python com MySQL

Executando comandos

Da mesma maneira que precisamos de um cursor no **PostgreSQL** para interagir com o banco de dados, no **MySQL** também é necessário.

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

Agora podemos utilizar a variável **cur** que recebeu o cursor para navegar no banco e executamos um comando **insert**.

O Python não tem uma função nativa para fazer a conexão com o MySQL. Por isso, instalamos o módulo MySQLdb, ele nos fornece todos os comandos necessário para interagir com o banco de dados.

Na aula de funções foi explicado o conceito de tratamento de exceções, que são 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 consiga efetuar 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 elas não sejam fechadas após o seu uso e o seu script tiver muitas conexões de banco, pode ocasionar o erro: MySQL server has gone away, causando a indisponibilidade do banco de dados.

Anotações:

Python com MySQL

Executando comandos

Da mesma maneira que precisamos de um cursor no **PostgreSQL** para interagir com o banco de dados, no **MySQL** também é necessário.

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

Para gravar os dados no banco devemos executar o **con.commit()**.

Segue abaixo 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: %s"%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 .

Python com MySQL

Executando comandos

Podemos fazer um update e um 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 abaixo um 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: %s"%e)
    con.rollback()

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

Python com MySQL

Sintaxe

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

O **cur.rollback()** é utilizado para voltar a última transação caso ocorra algum erro. O **cur.close()** e **con.close()** são utilizados para finalizar a conexão com o banco de dados.

Segue abaixo um 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: %s"%e)
    con.rollback()

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

Neste exemplo foi digitado errado o nome da tabela.

Python com MySQL

Sintaxe

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

Ao realizar uma busca, podemos retornar só o primeiro registro do banco utilizando o **fetchone()** ou todos eles utilizando o **fetchall()**.

Segue abaixo um 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: %s"%e)

finally:
    print ("Finalizando conexao 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.

Estrutura de Dados do MongoDB

Objetivos da Aula

- ✔ Conhecer o MongoDB;
- ✔ Compreender sua estrutura de dados.

Estrutura de Dados no MongoDB

Nesta aula iremos apresentar o MongoDB e ver a sua estrutura de dados.

Anotações:

Estrutura de Dados do MongoDB

MongoDB

Este é um banco de dados do tipo noSQL (Not Only SQL), os dados são armazenados no formato JSON e tem o objetivo de ter alta performance.

Os bancos de dados noSQL normalmente ocupam mais espaço em disco pois não tem o recurso de JOIN entre os dados, mas a sua leitra e escrita são bem superiores.

Estrutura de Dados no MongoDB

O MongoDB não é um substituto dos bancos relacionais, apesar de ser possível trabalhar somente com o MongoDB sem a necessidade de outros bancos de dados.

Mas quando usar o MongoDB?
Quando você precisa de velocidade em leitura e escrita de dados.

No site do MongoDB existem benchmarks mostrando que esee banco de dados consegue efetuar cerca de 270 mil operações de leitura e escrita por segundo.

O MongoDB 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:

Estrutura de Dados do MongoDB

MongoDB

O MongoDB trabalha com documentos, estes documentos são do tipo Chave – Valor.

Nele podemos armazenar documentos do tipo JSON que é o formato mais usado atualmente para enviar dados na web através de API's que utilizam o modelo REST.

Anotações:

Estrutura de Dados do MongoDB

MongoDB

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

Collections: Onde armazenamos os documentos, são como as tabelas nos bancos SQL.

Documentos: Onde armazenamos os dados (no formato JSON), no SQL são como os registros.

No MongoDB um registro é um documento, este documento é uma estrutua de dados composta por chave e valor (como um dicionário em Python). Uma chave pode ter qualquer valor (até outro dicionário ou uma lista).

Como pode exemplo:

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

}
```

Podemos ver maiores informações sobre collections na documentação oficial:

https://docs.mongodb.com/v3.2/core/databases-and-collections/

Anotações:

Estrutura de Dados do MongoDB

Exemplo de Documento

```
{
   nome :  "Mariana Albano" ,
   projetos : [
                {nome :  "Automação Python" ,
                 descricao : "Projeto de automação"},

                {nome :  "Deploy" ,
                  descricao :  "Projeto de deploy"}
              ]
 }
```


Anotações:

Estrutura de Dados do MongoDB

Instalação

A instalação pode ser feita da seguinte forma:

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


Anotações:

Trabalhar com comandos no MongoDB

Objetivos da Aula

- ✔ Criar e excluir bases;
- ✔ Fazer inserção, atualização, exclusão e consulta no MongoDB.

Trabalhar com comandos no MongoDB

Nesta aula iremos apresentar como podemos trabalhar com o MongoDB, criando collections, inserindo dados e fazendo as consultas.

Anotações:

Trabalhar com comandos no MongoDB

Utilizando o MongoDB

1 Acessar o MongoDB
```
# mongo
```

2 Ver os bancos que estão criados
```
> show dbs ou show databases
```

3 Acessar ou criar um banco
```
> use dexterops
```

4 Deletar m banco
```
> db.runCommand({dropDatabase:1})
```


No console do MongoDB, existem alguns comandos que foram criados para compatibilidade com usuários do MySQL, como por 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.

Precisamos tomar cuidado na hora de digitar o nome do banco de dados, pois como é mostrado, o MongoDB cria um banco de dados novo cada vez que for digitado um nome errado.

Sendo assim, é importante verificar os bancos de dados existentes e apagar sempre que for criado um banco de dados erroneamente evitando que exista o que chamamos de “sujeira” no banco de dados.

Trabalhar com comandos no MongoDB

Utilizando o MongoDB

1. Verificar as collections
   > show collections OU show tables

2. Insetir um registro:
   > db.fila.insert({“_id”:1, “servico”:”Intranet”,”status”:0})

3. Remover um documento
   > db.fila.remove({“_id”:1})

4. Verificar os documentos
   > db.fila.find()

Além desses comandos mostrados também existem os seguintes comandos:

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().

Trabalhar com comandos no MongoDB

Utilizando o MongoDB

1	Verificar os documentos de uma forma mais fácil de ler `> db.file.find().pretty()`
2	Fazer uma pesquisa com filtro `> db.fila.find({“_id”:2}).pretty()`
3	Fazer uma alteração no documento e apagar os dados `> db.fila.update({“_id”:1},{“status”:1})`
4	Fazer uma alteração no documento e permanecer os dados `> db.fila.update({“_id”:1},{“$set”:{“status”:1}})`

O parâmetro **_id** não é obrigatório, porém, caso ele não seja informado, o MongoDB irá gerar um id parecido com esse:

```
{ "_id" : ObjectId("569946c973fdfb8dacf877f3")
```

Isso serve para garantir a unicidade dos documentos, você pode inserir o mesmo documento várias vezes sem informar o _id, e o MongoDB irá inserir esses documentos, fazendo com que pelo menos um parâmetro garanta a diferença entre os documentos.

Ao utilizar o **update**, nunca podemos esquecer de usar a chave **$set** para atualizar um valor em específico. Caso essa chave não seja informada, o documento que você quer 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}
```

É importante também ter cuidado ao utilizar o método remove, pois se não forem especificados parâmetros, ele irá remover todos os seus documentos.

Trabalhar com comandos no MongoDB

Criando Documento com Subdocumento

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


Anotações:

Trabalhar com comandos no MongoDB

Alterando um Subdocumento

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


Anotações:

Trabalhar com comandos no MongoDB

Removendo um Subdocumento

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


Anotações:

Python com MongoDB

Objetivos da Aula

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

Python com MongoDB

Nesta aula iremos apresentar o módulo que fará a conexão com o MongoDB e integrar com o Python parar fazer as consultas, inserções, alterações e exclusões de documentos e subdocumentos.

Anotações:

Trabalhar com comandos no MongoDB

Instalação

Para conectar o **MongoDB** com seus scripts em **Python**, precisamos instalar um módulo chamado **PyMongo**.

```
pip install pymongo
```

Poucas coisas irão mudar da sintaxe do shell do MongoDB para a sintaxe do Python.

Anotações:

Trabalhar com comandos no MongoDB

Conexão

Vamos criar um script e nele instanciar um 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:

Trabalhar com comandos no MongoDB

Para inserir um novo documento, basta executar:

```
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 da mesma forma que fizemos no MongoDB, para isto, basta adicionar no script:

```
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: %s" %e)

inserir_dados()
```

Trabalhar com comandos no MongoDB

Para fazer a busca de dados utilizamos:

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

Esta busca nos retornará um objeto, então para conseguir exibir as informações, devemos fazer desta forma:

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


Busca de Dados

Para fazer a busca de dados utilizando subdocumentos, podemos fazer desta forma:

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


def buscar_dados():

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

buscar_dados()
```

Trabalhar com comandos no MongoDB

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, podemos fazer desta forma:

```
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()
```

Trabalhar com comandos no MongoDB

Fazer um update em um subdocumento:

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


Update em subdocumentos

Para fazer um 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()
```


Trabalhar com comandos no MongoDB

Fazer um update em 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 desta forma:

```
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()
```


Entendendo o Projeto

Objetivos da Aula

- ✔ Entender o projeto;
- ✔ Criar a estrutura do projeto;

Entendendo o Projeto

Nesta aula iremos entender o projeto e montar a sua estrutura.

Anotações:

Entendendo o Projeto

Projeto

Iremos criar um chat para comunicação interna da empresa Dexter Courier. Devemos atender os seguintes requisitos:

- ✔ As mensagens devem ser armazenadas no banco de dados;
- ✔ O padrão de armazenamento deve ser: nome, data, mensagem;
- ✔ Não devemos nos preocupar com autenticação.

Anotações:

Entendendo o projeto

Estrutura do projeto

Para armazenar as mensagens iremos utilizar o mongodb. Ele é o ideal para o projeto pois não existe a necessidade de criar relacionamento entre as tabelas e sua leitura/escrita são mais rápidas.

O armazenamento no banco será feito da seguinte forma:

{'nome':'Cliente','data':'28/07/2017 – 15:00', 'mensagem':'olá'}

Anotações:

Entendendo o projeto

Criando a estrutura do projeto

A estrutura do projeto deverá conter os seguintes arquivos:

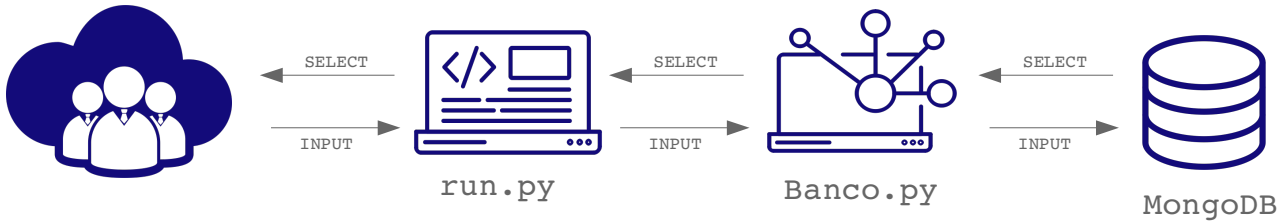
/run.py: será a interface e responsável pela execução do chat;

/modules/Banco.py: será responsável por consultar e gravar inforações no banco.

Anotações:

Entendendo o projeto

Fluxo da Aplicação



Fluxo da aplicação

O acesso do usuário a aplicação será através do arquivo “run.py” onde ele poderá escrever novas mensagens (input). Este input deve ser recebido pela nossa aplicação e enviado ao arquivo “ banco.py” para que ele envie a gravação da mensagem no banco.
Após este fluxo terminar, a aplicação deve automaticamente fazer com que as novas mensagens apareçam no chat.

Anotações:

Construindo o Projeto

Objetivos da Aula

- ✔ Criar o arquivo de conexão com o banco;
- ✔ Criar a classe e os métodos para interação do usuário.

Construindo o Projeto

Nesta aula iremos criar o arquivo responsável pela conexão com o MongoDB e o arquivo que fará a interação com o usuário.

Anotações:

Construindo o Projeto

Entendendo o módulo

Para construir o projeto iremos utilizar uma biblioteca chamada **“Curses”.**

Esta biblioteca permite a manipulação das exibições do modo texto, ou seja, conseguimos utilizar o nosso terminal para exibir as informações que forem necessárias.

Anotações:

Construindo o Projeto

Entendendo o módulo

Para iniciar o módulo podemos utilizar:

```
import curses

    tela = curses.initscr()
```


Anotações:

Construindo o Projeto

Entendendo o módulo

Podemos adicionar mensagens na tela usando o “addstr”:

```
while True:

    tela = curses.initscr()

    for i in range(0, 10, 2):

        tela.addstr(i, 0,"%s" %i)
```

Passamos como argumento a posição onde a mensagem será exibida.

Anotações:

Construindo o Projeto

Entendendo o módulo

Para digitar mensagens:

```
while True:

    tela = curses.initscr()

    input = tela.getstr()

    tela.addstr(input)
```


Anotações:

Construindo o Projeto

Entendendo o módulo

Ainda existem alguns métodos importantes que podemos utilizar:

```
#limpar a tela

tela.clear()

#sair do curses

curses.endwin()
```


Anotações:

Construindo o Projeto

Criando o arquivo de acesso ao banco

Este arquivo deverá conter: o método construtor com a conexão do banco.

```
from pymongo import MongoClient, DESCENDING
import time
class Banco:
    def __init__(self):
        self.client = MongoClient('127.0.0.1')
        self.db = self.client['chat']
```


Anotações:

Construindo o Projeto

Criando o arquivo de inserção

Um método que fará a inserção de mensagens:

```
def add_message(self,**kwargs):
    self.db.message.insert({
        "name":kwargs["name"],
        "message":kwargs["message"],
        "hora":str(time.strftime('%d-%m-%Y %H:%M:%S'))})
```


Anotações:

Construindo o Projeto

Criando o arquivo de busca

Um método que fará a busca das mensagens:

```
def filter(self):
    message = []
    for t in self.db.message.find().sort("hora",
                                         DESCENDING).
                                         limit(30):
            message.append(t)
    message.reverse()
    return message
```


Anotações:

Construindo o Projeto

Criando a interface

O método construtor para iniciar a tela:

```
import curses
import sys
from modules.Banco import Banco

class Chat:

    def __init__(self):
        self.tela = curses.initscr()
```


Anotações:

Construindo o Projeto

Criando a interface

O método para fazer a busca das mensagens:

```
def select_message(self):
      try:

          message = Banco()
          buscar_mensagem = message.filter()
          self.tela.clear()
          for t in buscar_mensagem:
              self.tela.addstr(("[%s] %s : %s \n")%(t["hora"],
                                                    str(t["name"]),
                                                    t["message"]))
      except KeyboardInterrupt as e:
          curses.endwin()
          sys.exit()
```


Anotações:

Construindo o Projeto

Criando a interface

O método para adiciona novas mensagens:

```
    def add_message(self, input):
        try:
            self.tela.addstr("Mensagem: ")
            input = self.tela.getstr().decode(encoding="utf-8")
            self.tela.clear()
            message = Banco()
            message.add_message(name=nome, message=str(input))

        except KeyboardInterrupt as e:
            curses.endwin()
            sys.exit()
```


Anotações:

Construindo o Projeto

Criando a interface

Vamos instanciar os métodos e iniciar o chat:

```
if __name__ == '__main__':
    try:
        nome = input("Digite o seu nome: ")
        iniciar = Chat()
        while True:
            iniciar.select_message()
            iniciar.add_message(nome)
    except KeyboardInterrupt as e:
        curses.endwin()
        sys.exit()
```


Anotações:

