# Fundamentos de Data Science para Finanças

Prof. Dr. Daniel Bergmann

# Fundamentos de Python

## Introdução

Olá, sou o Daniel Bergmann e serei seu professor no curso de Machine Learning e Data Science com Python. Neste tópico, aprenderemos os fundamentos da linguagem: como trabalhamos com variáveis, listas, tipos, laços condicionais, métodos, conversões e mais. Seguiremos sempre boas práticas de programação e principais convenções da linguagem.

Criaremos nossos primeiros códigos e aprenderemos conteúdos fundamentais para trabalhar com data science. Sugerimos que o curso de Métodos Quantitativos aplicados a Negócios do LIT seja um pré-requisito para aqueles que não estão tão familiarizados com tema.

Este curso é direcionado para pessoas que nunca fizeram nenhuma linha de código em Python e querem aprender a linguagem com foco em data science.

Espero que seja uma experiência de aprendizado incrível e que possamos vencer todos os desafios juntos. Neste módulo, vamos dar nossos primeiros passos na linguagem Python, aprendendo seus fundamentos e principais convenções.

## Preparando o Ambiente

Para realizar este curso, não é necessário realizar o download de nenhum programa ou software. Vamos utilizar uma ferramenta da Google chamada Colaboratory ou Colab.

Para começarmos a criar nosso código, utilizaremos a ferramenta Google Colab. Logo em sua página inicial, teremos uma breve apresentação do programa, e como é simples de utilizá-lo. Não é necessário realizar qualquer tipo de download ou configuração, além do compartilhamento facilitado.

A primeira coisa que precisamos fazer para usar o Google Colab é acessar o seguinte endereço:

**https://colab.research.google.com/notebooks/intro.ipynb**

Após o cadastro de um e-mail Gmail você consegue se inscrever na plataforma Google Colab e implementar seus códigos de Data Science em Python sem a necessidade de instalar qualquer software na sua máquina. Como vantagem, você pode rodar o Google Colab em Tablets e computadores com sistemas operacionais Windows, Apple Mac Os ou Linux.

No painel de ferramentas na parte superior direita da tela, clicaremos sobre "File > New notebook". É importante frisar que devemos estar logados em nossa conta do Google, pois tudo que criarmos será salvo em uma pasta do Google Drive na origem do arquivo. Salientamos que os notebooks das aulas podem ser baixados no endereço do Google Drive que o LIT disponibilizou aos participantes.

]

## Primeiros passos no Python

Ao fazermos uma busca simples no Google sobre este assunto, descobriremos que Python é uma linguagem de programação de alto nível, mas o que isso significa?

As linguagens de alto nível são mais parecidas com as linguagens humanas. As linguagens de baixo nível (Assembly, Fortran, C++ entre outras) são mais semelhantes às estruturas da máquina. Mas qual são as vantagens e desvantagens no uso dessas diferentes linguagens e seus níveis?

A linguagem de alto nível é mais fácil de aprender, afinal é mais intuitiva para o desenvolvedor humano, além de ter sua compreensão facilitada. Contudo, a linguagem de baixo nível gera melhor aproveitamento e desempenho da máquina, afinal o conteúdo não precisará ser interpretado pelo computador.

Primeiramente, abra um notebook em branco com o nome de “Fundamentos de Python LIT.ipynb” ou outro nome da sua escolha. Siga os passos a seguir a fim de reforçar os entendimentos dos principais conceitos vistos no módulo “Fundamentos de Python”.

Como podemos executar um código Python? Nas células poderemos fazer operações matemáticas ao pressionarmos "Shift + Enter":

```
>> 1+1
```

Será devolvido para nós o valor 2. Mas se simplesmente escrevermos Daniel ou qualquer outro nome nas células e acionarmos o "Shift + Enter" teremos uma mensagem de erro, afinal esta não é uma operação válida na linguagem.

Precisamos indicar que o que estamos escrevendo é de fato uma palavra, uma informação textual. Para isso envolveremos o nome "Daniel" entre aspas:

```
>> "Daniel"
```

Podemos utilizar aspas duplas “Daniel” ou aspas simples ‘Daniel’ a fim de representar um texto ou frase. Dentro da comunidade do Python é comumente utilizada aspas simples para escrever textos ou *strings*.

Sabemos que nosso nome é tal por repetição, nossos pais e amigos nos chamam pelo nome desde pequenos e então passamos a aprender e reter essa informação. Como podemos fazer com o que o programa aprenda algum específico? A resposta é: nomeação por meio de variáveis.

```
>> nome = 'Daniel'
```

E posteriormente, escrevermos em outra célula:

```
>> nome
```

Teremos como retorno "Daniel". Conseguimos armazenar na memória do computador uma informação. Podemos inclusive adicionar a variável idade:

```
>> idade = 38
```

Como estamos lidando com números, não é necessário o uso das aspas. E se escrevermos novamente idade em uma célula, teremos o retorno 38. Em suma, conseguimos guardar dados importantes para nós na memória do computador.

O que faremos na sequência é exibir uma mensagem usando uma das variáveis criadas, tanto nome quanto idade. Existe uma função em Python que nos possibilita a exibição de conteúdos na tela, função print(), sendo que dentro dos parênteses teremos as informações a serem exibidas como resposta.

```
>> print('O nome é Daniel e sua idade é 38 anos')
```

Ao executarmos essa função por meio de "Shift + Enter", teremos a mensagem exibida integralmente na tela. Mas e se quisermos exibir o conteúdo armazenado na variável, e não apenas "Daniel" ou a idade de 38 anos? Para isso, acrescentaremos um "f" e deixar as funções entre chaves.

```
>> print(f'O nome é {nome} e sua idade é {idade} anos')
```

Podemos modicar o valor das variáveis de acordo com nosso interesse, então o valor da idade pode se tornar 39 ou 40, por exemplo.

Existe uma maneira de criarmos nossas próprias funções. Adicionaremos um novo texto de título que chamaremos "Criando minha primeira função". Para tal, clique no menu Insert > Text e digite:

```
# Criando nossa primeira função
```

Queremos criar uma função chamada "saudacao", perceba que não utilizamos qualquer tipo de acentuação, afinal para trechos de código não usamos caracteres especiais. O código utilizado para criarmos funções é o def.

```
>> def saudacao():
```

Ao iniciarmos a escrita da função, logo após os dois pontos, será criado um espaço na célula para organizar a orientação da escrita. Essa orientação é muito importante, pois ela ditará o funcionamento do nosso código.

Nossa função se dará da seguinte maneira: a partir do input "qual é o seu nome" deveremos retornar com o valor da variável nome, que configuramos anteriormente.

```
>> def saudacao():
       nome = input('Qual é o seu nome?')
       print(f'olá {nome}')
```

A função foi criada para ser executada conforme o código abaixo.

```
>> saudacao()
```

Teremos como input, o seguinte texto:

```
Qual é o seu nome?
```

E então, haverá um espaço para digitar "Daniel". Como resposta, obteremos:

```
Olá Daniel
```

Poderemos escrever outros nomes, tais como: "Luis" ou "Maria". Desta forma, criamos nossas próprias funções conforme surgem as necessidades do nosso dia a dia.

Criamos uma função em que coletamos um nome e imprimimos seu valor. Porém, o nome que criamos está na função saudacao()

```
>> def saudacao():
        nome = input('Qual é o seu nome? ')
        print(f'Olá {nome}')
```

Nossa tarefa neste momento será criar uma função cujo valor do nome não esteja em seu interior. Primeiramente, criaremos o nome Luiza.

```
>> nome = 'Luiza'
```

Posteriormente, trabalharemos em nossa função, que chamaremos de 'saudacao ()', com alguns parâmetros de interesse. Dentro dos () conseguimos incluir valores a serem utilizados para o funcionamento desta função.

```
>> def saudacao_com_parametros(nome_da_pessoa):
        print(f'Olá {nome_da_pessoa}')
```

A função foi gerada para que possamos evocar como parâmetro a variável nome_da_pessoa. Parâmetros são argumentos localizados fora do escopo da função, mas que podem ser utilizados por ela.

## Condicionais

Compreendemos na seção anterior o que são as funções, parâmetros e variáveis. Nesta parte, aprenderemos o conceito de "condicionais". Criaremos uma mensagem de texto para deixar nosso notebook organizado, intitulado de "Condicional". Para tanto, insira um texto e digite:

```
# Condicional
```

Nossa proposta é armazenar a idade de alguém no computador.

```
>> idade = 15
```

Faremos uma função que verifica se o usuário tem a idade correta para dirigir um carro ou não. Ou seja, se a idade for maior ou igual a 18 anos, o usuário pode ter sua habilitação válida e dirigir veículos, caso o contrário a permissão será negada.

A nova função será chamada de: verifica_se_pode_dirigir(), ao qual receberá somente idade como parâmetro. Então, precisaremos verificar se a idade é maior ou menor que 18 anos. Caso esta condição seja respeitada, imprimiremos a mensagem “Tem permissão para dirigir”.

```
>> idade = 20
   def verica_se_pode_dirigir(idade):
      if idade >= 18:
         print('Tem permissão para dirigir')
```

Ao executarmos o código acima com “Shift + Enter”, aparecerá na tela a seguinte informação: “Tem permissão para dirigir”.

Ao invés de 20 anos, se colocarmos 15 anos na idade, teremos o seguinte código:

```
>> idade = 15
   def verica_se_pode_dirigir(idade):
      if idade >= 18:
         print('Tem permissão para dirigir')
```

Ao executarmos o código, nada acontece. Não ditamos qualquer tipo de comportamento para a função caso a idade em questão seja menor de 18 anos. Então, precisaremos inserir o comando else dentro da função no intuito de indicar se uma determinada condição pode ou não ser atendida.

```
>> idade = 15
   def verica_se_pode_dirigir(idade):
      if idade >= 18:
         print('Tem permissão para dirigir')
       else:
         print('Não tem permissão para dirigir')
```

A função se tornou executável e, desta forma, mensagens diferentes serão exibidas de acordo com a idade fornecida.

Ainda no contexto das condicionais, queremos receber o valor da idade dentro da própria função, ou seja, sem a exigência dos parâmetros. Nossa nova função será descrita pelo seguinte script: ‘verifica_se_pode_dirigir_sem_parametro()’. Primeiramente, devemos coletar a idade para que a condição da habilitação seja adequadamente testada.

```
>> idade = 15
   def verica_se_pode_dirigir_sem_parametro():
      idade = input('Qual sua idade')
      if idade >= 18:
         print('Tem permissão para dirigir')
       else:
         print('Não tem permissão para dirigir')
```

Ao executarmos nossa nova função, será pedido a idade do usuário, e então inseriremos o valor 10. Notaremos um erro, pois o termo ">=" não é suportado entre instâncias de 'str'. Mas, o que isso quer dizer?

O termo 'str’ se refere a uma string, ou seja, uma informação textual com caracteres, enquanto 'int' se refere a um número inteiro. Isso quer dizer que o input da idade está sendo lido como um texto e não

como um número. Realizaremos uma conversão para que a idade seja do tipo inteiro como se demonstra no código a seguir:

```
>> idade = 15
   def verica_se_pode_dirigir_sem_parametro():
       idade = input('Qual sua idade')
       idade = int(idade)
      if idade >= 18:
        print('Tem permissão para dirigir')
       else:
        print('Não tem permissão para dirigir')
```

Dessa maneira será coletada a string para que a conversão ao tipo inteiro seja executada no Python. Agora, ao executarmos nossa função e inserirmos a variável idade como input, receberemos a mensagem que o código foi executado corretamente.

Por isso é importante saber se estamos trabalhando com números inteiros ou palavras, pois existem operações que não irão funcionar corretamente caso não realizemos as conversões apropriadas. Dessa maneira, temos duas funções com formatos diferentes: uma que utiliza parâmetro e outra que se abstém da sua utilização.

## Lista, repetições tipos booleanos

Aprenderemos outras maneiras de armazenar valores por meio da linguagem Python. Existe uma forma de armazenarmos uma série de elementos de uma única vez por meio do objeto conhecido por listas no Python. Como exemplo:

```
>> idade = 22
```

Sabemos que estamos lidando com um número inteiro, mas há maneiras de confirmar qual é o tipo de conteúdo a partir da função type().

```
>> type (idade)
```

Será retornado int como esperávamos. O mesmo acontece se escrevermos strings, como:

```
>> nome = "Luiza"
```

Como retorno do método type( ), obteremos o termo str. Podemos guardar uma lista de elementos que contemplam diferentes idades por meio dos colchetes.

```
>> idades = [18, 22, 15, 50 ]
```

Este é um elemento do tipo list, como poderemos verificar por meio do método type( ). Mas como poderemos coletar elementos específicos, como o segundo ou terceiro número desta lista de idades?

Poderemos escrever:

```
>> idades[2]
```

Estranhamente, teremos como retorno o valor 15, que está na terceira posição na lista. Porque será que isso ocorreu? Sempre que criamos uma lista, nosso primeiro elemento é o índice 0, ainda que não esteja claramente visível, observe:

```
>> idade = [18, 22, 15, 50]
            0  , 1,  2,  3
```

Se quisermos coletar idades específicas como, por exemplo, do índice 0 ao 2, isto é, os valores 18, 22 e 15, poderemos escrever:

```
>> idades[0:2]
```

Como resultado teremos os valores 18 e 22. Não era isso que esperávamos! Para exibirmos os valores que desejamos, devemos escrever:

```
>> idades[0:3]
```

Isso ocorre porque o último número da lista é excluído na leitura, então precisamos adicionar um índice para coletar os três valores. Se quisermos coletar os números 22, 15 e 50 devemos escerever:

```
>> idades[1:]
```

Assim, serão coletadas todas as idades com excessão da primeira. Se quisermos coletar apenas o último número, podemos digitar da seguinte forma:

```
>> idades[-1]
```

Se escrevermos idades[-2], por exemplo, teremos como resultado o valor 15. Existem algumas formas diferentes de coletar elementos de uma lista, como pudemos demonstrar. Vamos dar preferência sempre para a leitura que se inicia do índice 0 até um determinado elemento.

O que faremos nesta aula é juntar os conhecimentos que já temos. Em idades, temos uma lista de valores armazenados no computador, e nosso objetivo é criar uma função para verificar se são idades válidas para a habilitação de um veículo.

Temos nossa lista de idades, mas queremos saber para cada idade da lista se o valor é maior ou igual a 18. Para isso, utilizaremos algo como "para cada idade...", sendo este "para" o comando for em linguagem Python.

```
>> def verifica_se_pode_dirigir(idade):
     if idade >= 18:
       print(f'{idade} anos de idade, TEM permissão para dirigir')
     else:
       print(f'{idade} anos de idade, NÃO TEM permissão para dirigir')

   for idade in idades:
     verifica_se_pode_dirigir(idade)
```

Ao executarmos nossa função teremos o seguinte resultado:

```
>> 18 anos de idade, TEM permissão para dirigir
   22 anos de idade, TEM permissão para dirigir
   15 anos de idade, NÃO TEM permissão para dirigir
   50 anos de idade, TEM permissão para dirigir
```

Nossa lista funcionou, e conseguimos verificar cada idade. Notemos que o for que utilizamos está fora do escopo da função, mas como poderíamos incluí-lo?

```
>> def verifica_se_pode_dirigir(idades):
     for idade in idades:
       if idade >= 18:
          print(f'{idade} anos de idade, TEM permissão para dirigir')
       else:
          print(f'{idade} anos de idade, NÃO TEM permissão para dirigir')
```

Com o 'for' dentro da função, passamos uma lista como parâmetro e para cada idade da lista será executada a verificação. Ao executarmos o código, teremos o texto que acompanha os valores da idade após sua verificação. Os resultados obtidos são exatamente os mesmos do exemplo anterior.

Além da string, inteiros e listas também temos o tipo booleano. Criaremos uma nova sessão em nosso notebook com o título "Booleano", e então começaremos a estudar. Quando afirmamos que a idade é maior ou menor do que 18 anos, escrevemos:

```
>> idade = 18
   idade >= 18
```

Receberemos como resposta para aquela pergunta um 'True', isto é, verdadeiro. Implicando que nossa afirmação anterior é de fato real, o valor de idade é maior ou igual a 18. Caso afirmação seja considerada falsa, receberemos um 'False' como resposta, como no exemplo a seguir:

```
>> idade = 15
idade >= 18
```

Nossa tarefa será verificar em uma lista de idades quais estão aptas para ter habilitação e, a seguir, armazenar esses dados em uma nova lista que chamaremos de 'permissoes', que inicialmente estará vazia.

```
>> permissoes = []
idades = [20, 14, 40]
```

Queremos realizar uma verificação lógica a fim de adicionar os valores 'True'na lista denominada 'permissoes', e para tal propósito utilizaremos o método do Python conhecido como append(). Caso a idade não atenda os requisitos para o estado 'True', os valores serão armazenados com o rótulo 'False'. Segue o código desenvolvido a seguir:

```
>> def verifica_se_pode_dirigir(idades, permissoes):
    for idade in idades:
      if idade >= 18:
          permissoes.append(True)
      else:
          permissoes.append(False)
```

Como resultado, teremos:

```
>> [True, False, True]
```

Iremos refinar um pouco mais: se houver permissão para dirigir, queremos exibir uma mensagem afirmativa, caso contrário exibirá outra negativa. Para cada permissão da lista de 'permissoes, faremos tanto uma verificação como uma comparação lógica. Para realizar o procedimento comparativo, utilizamos o famoso duplo sinal de igualdade ==.

```
>> for permissao in permissoes:
  if permissao == True:
    print('Tem permissão para dirigir')
  else:
    print('Não tem permissão para dirigir')
```

Receberemos como resposta

```
Tem permissão para dirigir
Não tem permissão para dirigir
Tem permissão para dirigir
```

Já avançamos em nosso estudo com tipos booleanos, utilização dos laços com 'for'e realização de verificações comparativas, além de atribuir valores dentro de uma lista por meio do append().